

Podstawowy podręcznik do nauki algorytmiki

- Przystępne wprowadzenie do algorytmiki
- Bez zbędnej teorii
- Gotowe rozwiązania w C++

ALGORYTMY

STRUKTURY DANYCH

I TECHNIKI PROGRAMOWANIA

WYDANIE IV



PIOTR WRÓBLEWSKI



Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redakcja: Michał Mrowiec

Projekt okładki: Marcin Pasek

Fotografia na okładce została wykorzystana za zgodą iStockPhoto Inc.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?algo4_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-5377-5

Copyright © Helion 2010

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

- Przedmowa 9**
- Rozdział 1. Zanim wystartujemy 17**
 - Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych 18
 - Jak to się niedawno odbyło,
czyli o tym, kto „wymyślił” metodologię programowania 21
 - Proces koncepcji programów 22
 - Poziomy abstrakcji opisu i wybór języka 23
 - Poprawność algorytmów 24
- Rozdział 2. Rekurencja 27**
 - Definicja rekurencji 27
 - Ilustracja pojęcia rekurencji 28
 - Jak wykonują się programy rekurencyjne? 30
 - Niebezpieczeństwa rekurencji 31
 - Ciąg Fibonacciego 31
 - Stack overflow! 33
 - Pułapek ciąg dalszy 34
 - Stąd do wieczności 34
 - Definicja poprawna, ale... 35
 - Typy programów rekurencyjnych 36
 - Myślenie rekurencyjne 38
 - Przykład 1.: Spirala 38
 - Przykład 2.: Kwadraty „parzyste” 40
 - Uwagi praktyczne na temat technik rekurencyjnych 41
 - Zadania 42
 - Rozwiązania i wskazówki do zadań 44
- Rozdział 3. Analiza złożoności algorytmów 49**
 - Definicje i przykłady 50
 - Jeszcze raz funkcja silnia 53
 - Zerowanie fragmentu tablicy 57
 - Wpadamy w pułapkę 58
 - Różne typy złożoności obliczeniowej 59
 - Nowe zadanie: uprościć obliczenia! 61

Analiza programów rekurencyjnych	62
Terminologia i definicje	62
Ilustracja metody na przykładzie	63
Rozkład logarytmiczny	64
Zamiana dziedziny równania rekurencyjnego	66
Funkcja Ackermanna, czyli coś dla smakoszy	66
Złożoność obliczeniowa to nie religia!	68
Techniki optymalizacji programów	68
Zadania	69
Rozwiązania i wskazówki do zadań	70
Rozdział 4. Algorytmy sortowania	73
Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	74
Sortowanie bąbelkowe, algorytm klasy $O(N^2)$	75
Quicksort, algorytm klasy $O(N \log N)$	77
Heap Sort — sortowanie przez kopcowanie	80
Scalanie zbiorów posortowanych	82
Sortowanie przez scalanie	83
Sortowanie zewnętrzne	84
Uwagi praktyczne	87
Rozdział 5. Typy i struktury danych	89
Typy podstawowe i złożone	89
Ciągi znaków i napisy w C++	90
Abstrakcyjne struktury danych	92
Listy jednokierunkowe	93
Tablicowa implementacja list	115
Stos	119
Kolejki FIFO	123
Stery i kolejki priorytetowe	125
Drzewa i ich reprezentacje	131
Zbiory	143
Zadania	145
Rozwiązania zadań	146
Rozdział 6. Derekursywacja i optymalizacja algorytmów	147
Jak pracuje kompilator?	148
Odrobina formalizmu nie zaszkodzi!	150
Kilka przykładów derekursywacji algorytmów	151
Derekursywacja z wykorzystaniem stosu	154
Eliminacja zmiennych lokalnych	154
Metoda funkcji przeciwnych	156
Klasyczne schematy derekursywacji	158
Schemat typu while	159
Schemat typu if-else	160
Schemat z podwójnym wywołaniem rekurencyjnym	162
Podsumowanie	163
Rozdział 7. Algorytmy przeszukiwania	165
Przeszukiwanie liniowe	165
Przeszukiwanie binarne	166
Transformacja kluczowa (hashing)	167
W poszukiwaniu funkcji H	169
Najbardziej znane funkcje H	169
Obsługa konfliktów dostępu	171

Powrót do źródeł	172
Jeszcze raz tablice!	173
Próbkowanie liniowe	173
Podwójne kluczowanie	175
Zastosowania transformacji kluczowej	176
Podsumowanie metod transformacji kluczowej	176
Rozdział 8. Przeszukiwanie tekstów	179
Algorytm typu brute-force	179
Nowe algorytmy poszukiwań	181
Algorytm K-M-P	182
Algorytm Boyera i Moore'a	185
Algorytm Rabina i Karpa	187
Rozdział 9. Zaawansowane techniki programowania	191
Programowanie typu „dziel i zwyciężaj”	192
Odszukiwanie minimum i maksimum w tablicy liczb	193
Mnożenie macierzy o rozmiarze $N \times N$	195
Mnożenie liczb całkowitych	197
Inne znane algorytmy „dziel i zwyciężaj”	198
Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas	199
Problem plecakowy, czyli niełatwe jest życie turysty piechura	200
Programowanie dynamiczne	202
Ciąg Fibonacciego	203
Równania z wieloma zmiennymi	204
Najdłuższa wspólna podsekwencja	205
Inne techniki programowania	208
Uwagi bibliograficzne	210
Rozdział 10. Elementy algorytmiki grafów	211
Definicje i pojęcia podstawowe	212
Cykle w grafach	214
Sposoby reprezentacji grafów	217
Reprezentacja tablicowa	217
Słowniki węzłów	218
Listy kontra zbiory	219
Podstawowe operacje na grafach	220
Suma grafów	220
Kompozycja grafów	220
Potęga grafu	220
Algorytm Roy-Warshalla	221
Algorytm Floyd-Warshalla	224
Algorytm Dijkstry	227
Algorytm Bellmana-Forda	228
Drzewo rozpinające minimalne	228
Algorytm Kruskala	229
Algorytm Prima	230
Przeszukiwanie grafów	230
Strategia „w głąb” (przeszukiwanie zstępujące)	231
Strategia „wszerz”	232
Inne strategie przeszukiwania	234
Problem właściwego doboru	235
Podsumowanie	239
Zadania	239

Rozdział 11. Algorytmy numeryczne	241
Poszukiwanie miejsc zerowych funkcji	241
Iteracyjne obliczanie wartości funkcji	243
Interpolacja funkcji metodą Lagrange'a	244
Różniczkowanie funkcji	245
Całkowanie funkcji metodą Simpsona	247
Rozwiązywanie układów równań liniowych metodą Gaussa	248
Uwagi końcowe	251
Rozdział 12. Czy komputery mogą myśleć?	253
Przegląd obszarów zainteresowań Sztucznej Inteligencji	254
Systemy eksperckie	255
Sieci neuronowe	256
Reprezentacja problemów	257
Ćwiczenie 1.	258
Gry dwuosobowe i drzewa gier	259
Algorytm mini-max	260
Rozdział 13. Kodowanie i kompresja danych	265
Kodowanie danych i arytmetyka dużych liczb	267
Kodowanie symetryczne	267
Kodowanie asymetryczne	268
Metody prymitywne	274
Łamanie szyfrów	275
Techniki kompresji danych	275
Kompresja poprzez modelowanie matematyczne	277
Kompresja metodą RLE	278
Kompresja danych metodą Huffmana	279
Kodowanie LZW	283
Idea kodowania słownikowego na przykładach	284
Opis formatu GIF	286
Rozdział 14. Zadania różne	289
Teksty zadań	289
Rozwiązania	291
Dodatek A Poznaj C++ w pięć minut!	295
Elementy języka C++ na przykładach	295
Pierwszy program	295
Dyrektywa #include	296
Podprogramy	296
Procedury	296
Funkcje	297
Operacje arytmetyczne	298
Operacje logiczne	298
Wskaźniki i zmienne dynamiczne	299
Referencje	300
Typy złożone	300
Tablice	300
Rekordy	301
Instrukcja switch	301
Iteracje	302
Struktury rekurencyjne	303
Parametry programu main()	303
Operacje na plikach w C++	303

Programowanie obiektowe w C++	304
Terminologia	304
Obiekty na przykładzie	305
Składowe statyczne klas	308
Metody stałe klas	308
Dziedziczenie własności	308
Kod warunkowy w C++	311
Dodatek B Systemy obliczeniowe w pigułce	313
Kilka definicji	313
System dwójkowy	313
Operacje arytmetyczne na liczbach dwójkowych	315
Operacje logiczne na liczbach dwójkowych	315
System ósemkowy	317
System szesnastkowy	317
Zmienne w pamięci komputera	317
Kodowanie znaków	318
Dodatek C Kompilowanie programów przykładowych	321
Zawartość archiwum ZIP na ftp	321
Darmowe kompilatory C++	321
Kompilacja i uruchamianie	322
GCC	322
Visual C++ Express Edition	323
Dev C++	327
Literatura	329
Spis tabel	331
Spis ilustracji	333
Skorowidz	339

Przedmowa

Algorytmika stanowi gałąź wiedzy, która w ciągu ostatnich kilkudziesięciu lat dostarczyła wielu efektywnych narzędzi wspomagających rozwiązywanie różnorodnych zagadnień za pomocą komputera. Dla jednych jest to tylko swego rodzaju książka kucharska, do której sięga się w razie potrzeby po wybrane „przepisy”. Dla innych algorytmika stanowi okazję do rozwinięcia umiejętności skutecznego rozwiązywania problemów i szkołę niestandardowego myślenia. Moją intencją w trakcie pisania niniejszej książki było połączenie tych dwóch perspektyw poprzez prezentację w miarę szerokiego, ale zarazem pogłębionego wachlarza tematów z tej dziedziny. Chciałem przy okazji opisywanych zagadnień ukazać Czytelnikowi odpowiednią perspektywę możliwych zastosowań komputerów, wychodząc poza wzory matematyczne i suchy kod programów przykładowych.

Co odróżnia tę książkę od innych podręczników?

Poprzednie wydania tej książki okazały się dużym sukcesem wydawniczym i zyskały wielu wiernych odbiorców. Z perspektywy lat widzę, że głównym powodem tak życzliwego przyjęcia był prosty język publikacji (unikalem w miarę możliwości zagłębiania się w złożony aparat matematyczny, ilustrując jednocześnie materiał teoretyczny przykładami i prawdziwym kodem w języku C++). Zwłaszcza to ostatnie posunięcie okazało się przysłowiowym „strzałem w dziesiątkę”, gdyż Czytelnicy mieli już dość podręczników nasyconych pseudokodem, niedającym się łatwo przełożyć na realia kompilatorów i wymogów systemów operacyjnych. W tym wydaniu postanowiłem odświeżyć kody źródłowe programów przykładowych, tak aby bez problemu dały się uruchomić pod popularnymi, darmowymi kompilatorami (Microsoft Visual C++ Express Edition i GCC/Dev C++), podałem także szereg porad praktycznych dotyczących samego procesu kompilacji, co powinno pomóc zupełnie początkującym adeptom sztuki programowania (dodatek C).

Mam oczywiście świadomość, że nie jest możliwe zaprezentowanie wszystkiego, co najważniejsze w dziedzinie algorytmiki, w obrębie jednej książki. Jest to niewykonalne z uwagi na rozpiętość dziedziny, z jaką mamy do czynienia. Może się więc okazać, że to, co zostało pomyślane jako logicznie skonstruowana całość, jednych rozczaruje, innych zaś przytłoczy ogromem poruszanych zagadnień. Moim pragnieniem było stworzenie w miarę reprezentatywnego przeglądu zagadnień algorytmicznych, przydatnego dla tych Czytelników, którzy programowanie mają zamiar potraktować w sposób profesjonalny. Po przeczytaniu tej książki być może odczujesz potrzebę przejrzania innej podobnej literatury (spis znajdziesz na końcu książki), ale nie zdziw się, jeśli spotkasz pozycje przeładowane matematyką lub zawierające masę pseudokodu zamiast

prawdziwych programów, które dają się łatwo kompilować i testować. Mam jednak nadzieję, że po lekturze tego podręcznika będzie Ci łatwiej zmagać się z bardziej opasłymi tomami, niestety zazwyczaj pisanymi dość hermetycznym językiem.

Niniejszy podręcznik polecam szczególnie osobom zainteresowanym programowaniem, a nie mającym do tego solidnych podstaw teoretycznych. Ponieważ grupuje on dość obszerną klasę zagadnień z dziedziny informatyki, będzie również użyteczny jako repetytorium dla tych, którzy zajmują się programowaniem zawodowo. Jest to jednak książka dla osób, które przynajmniej częściowo zetknęły się z programowaniem i rozumieją podstawowe pojęcia, takie jak: *zmienna*, *program*, *kompilacja* itd. — tego typu terminy będą bowiem stanowiły podstawę języka używanego w tej książce i nie będę się zagłębiał w ich wyjaśnianie.

Dlaczego C++?

Niewątpliwie kilka słów wyjaśnienia należy poświęcić problemowi języka programowania, w którym są prezentowane algorytmy w książce. Wybór padł na nowoczesny i modny język C++, którego precyzja zapisu i modularność przemawiają za używaniem go do programowania nowoczesnych aplikacji. C++ posiada oczywiście szereg wad, z których główną jest brak mechanizmów narzucających dobry styl programowania (programista ma czasami zbyt dużą swobodę wyboru). Z drugiej jednak strony żaden inny język nie pozwala tak swobodnie programować obiektowo, a gdy ktoś ma ochotę, może także wykorzystać mechanizmy niskopoziomowe. Taka swoboda jest przydatna np. w programowaniu systemowym lub na poziomie sprzętu (sterowniki), o grach już nie wspominając.

Warto jednak przy okazji podkreślić, że sam język prezentacji algorytmu nie ma istotnego znaczenia dla jego działania — jest to tylko narzędzie i stanowi wyłącznie zewnętrzną powłokę, która ulega zmianom w zależności od aktualnie panujących mód. Ponieważ C++ zdobył sobie olbrzymią popularność, został wybrany dla potrzeb tej książki. Dla kogoś, kto niedawno poznał C++, może to być doskonała okazja do przestudiowania potencjalnych zastosowań tego języka. Dla programujących dotychczas tylko w Pascalu został przygotowany minikurs języka C++, który powinien umożliwić im błyskawiczne opanowanie podstawowych różnic między C++ i Pascaliem. Składnia języka C++ jest na tyle podobna do innych języków, że na pewno osoby preferujące Javę lub niedawne „wynałazki” podobne do C# (w mojej opinii ten język, oparty na składni kilku innych języków, wprowadza tylko zbędne zamieszanie) poradzą sobie z dostosowaniem algorytmów do własnych potrzeb.

Oczywiście niemożliwe jest szczegółowe nauczanie się tak obszernego pojęciowo języka, jakim jest C++, dysponując objętością zaledwie krótkiego dodatku — bo tyle miejsca zostało przeznaczone na ten cel. Zamiarem moim było jedynie przełamanie bariery składniowej, tak aby prezentowane listingi były zrozumiałe. Czytelnik pragnący poszerzyć znajomość zasad programowania w C++ powinien sięgnąć po jeden z podręczników podanych w bibliografii; szczególnie polecam — ze względu na prostotę opisu — książkę [Eck02], ambitni powinni obojętnie przestudiować [Str04] — dzieło napisane przez samego twórcę języka, Bjarne Stroustrupa i stanowiące bardzo kompletne źródło wiedzy na temat C++.

Jak należy czytać tę książkę?

Czytelnik, który zetknął się wcześniej z tematyką podejmowaną w tej książce, może czytać ją w dość dowolnej kolejności, wynikającej z bieżących potrzeb.

Dla początkujących zalecane jest trzymanie się porządku narzuconego przez układ rozdziałów. Książka zawiera szczegółowy skorowidz i spis ilustracji — powinny one ułatwić odszukiwanie potrzebnych informacji.

Wiele rozdziałów zawiera na końcu zestaw zadań związanych tematycznie z aktualnie opisywanymi zagadnieniami. W dużej części zadania te są rozwiązane, ewentualnie podane są szczegółowe wskazówki do nich.

Oprócz zadań tematycznych ostatni rozdział zawiera zestaw różnorodnych zadań, które nie zmieściły się w toku wykładu. Przed ich rozwiązaniem zaleca się dokładne przestudiowanie całego materiału, który obejmują pozostałe rozdziały.

Co zostało opisane w tej książce?

Aby ułatwić nieco nawigację po różnorodnych tematach poruszanych w książce, postanowiłem zaprezentować główną tematykę kolejnych rozdziałów. Poniższy opis jest w pewnym sensie powtórzeniem spisu treści, z dodatkowym komentarzem dotyczącym zawartości.

Rozdział 1. Zanim wystartujemy

Rozbudowany wstęp pozwalający wziąć głęboki oddech przed przystąpieniem do pracy przy klawiaturze. W rozdziale tym poznasz kilka niezbędnych faktów historycznych dotyczących przeszłości algorytmiki i zrozumiesz, skąd wziął się obecny postępowanie w tej dziedzinie.

Rozdział 2. Rekurencja

Rozdział ten jest poświęcony jednemu z najważniejszych mechanizmów używanych w procesie programowania — rekurencji. Uświadamia zarówno oczywiste zalety, jak i nie zawsze widoczne wady tej techniki programowania. Zagadnienia poznasz na prostych i trudnych przykładach oraz będziesz miał szansę sprawdzić się, rozwiązując ciekawe zadania graficzne.

Rozdział 3. Analiza złożoności algorytmów

Przegląd najpopularniejszych i najprostszych metod służących do obliczania sprawności obliczeniowej algorytmów i porównywania ich ze sobą w celu wybrania najefektywniejszego. Rozdział przeznaczony raczej dla studentów informatyki, choć osoby ogólnie zainteresowane programowaniem powinny nań rzucić choćby pobieżnie okiem.

Rozdział 4. Algorytmy sortowania

Prezentuje najpopularniejsze i najbardziej znane procedury sortujące. Oczywiście rozdział nie wyczerpuje tematu — zakładam, że stanie się zachętą do dalszego studiowania arcyciekawej dziedziny algorytmów sortujących, mającej na dodatek duże walory dydaktyczne. Szczegółowo przedstawione są zarówno proste metody, np. sortowania przez wstawianie, bąbelkowe, jak i złożone, ze szczególnym naciskiem na szczegółowy opis metody *Quicksort*.

Rozdział 5. Typy i struktury danych

Omawia popularne struktury danych (listy, kolejki, drzewa binarne, etc.) i ich implementację programową w C++. Szczególną uwagę poświęcono ukazaniu możliwych zastosowań nowo poznanych struktur danych.

Rozdział 6. Derekursywacja i optymalizacja algorytmów

Prezentuje sposoby przekształcania programów rekurencyjnych na ich wersje iteracyjne. Rozdział ten ma charakter bardzo techniczny i jest przeznaczony dla programistów zainteresowanych problematyką optymalizacji programów.

Rozdział 7. Algorytmy przeszukiwania

Rozdział ten wykorzystuje kilka poznanych już wcześniej metod w stosunku do zagadnienia wyszukiwania elementów w słowniku, a następnie szczegółowo omawia metodę transformacji kluczowej (ang. *hashing*).

Rozdział 8. Przeszukiwanie tekstów

Ze względu na wagę tematu algorytmy przeszukiwania tekstów zostały zgrupowane w osobnym rozdziale. Szczegółowo omówiono metody *brute-force*, *K-M-P*, *Boyera i Moore'a*, *Rabina i Karpa*.

Rozdział 9. Zaawansowane techniki programowania

Wieloletnie poszukiwania w dziedzinie algorytmiki zaowocowały wynalezieniem pewnej grupy metod o charakterze generalnym: programowania dynamicznego, „dziel i zwyciężaj”, algorytmów „żarłoczych” (ang. *greedy*). Te *metaalgorytmy* znacznie rozszerzają zakres możliwych zastosowań komputerów do rozwiązywania problemów.

Rozdział 10. Elementy algorytmiki grafów

Opis jednej z najciekawszych struktur danych występujących w informatyce. Grafy ułatwiają (a czasami po prostu umożliwiają) rozwiązanie wielu problemów, które traktowane za pomocą innych struktur danych wydają się nie do rozwiązania.

Rozdział 11. Algorytmy numeryczne

Rozdział prezentuje kilka ciekawych problemów natury obliczeniowej, ukazujących zastosowanie komputerów w matematyce, a konkretnie do wykonywania obliczeń przybliżonych, jakimi są: miejsca zerowe funkcji, interpolacje, różniczkowanie, całkowanie, metoda *Gaussa* itp.

Rozdział 12. Czy komputery mogą myśleć?

Wstęp do bardzo rozległej dziedziny tzw. *sztucznej inteligencji*. Omówienie obszarów zainteresowań tej dziedziny, przykład implementacji programowej popularnego w teorii gier algorytmu *Mini-Max*.

Rozdział 13. Kodowanie i kompresja danych

Obszerne omówienie popularnych metod szyfrowania i kompresji danych. W rozdziale tym zapoznasz się z pojęciem kodowania symetrycznego i asymetrycznego, omówię też szczegółowo system kryptograficzny z kluczem publicznym (*RSA*), przy tej okazji poznasz także sposób wykonywania operacji arytmetycznych na bardzo dużych liczbach całkowitych. Zagadnienia kompresji danych poznasz, poczynawszy od podstaw teoretycznych i prostych metod, opiszę także szczegółowo słynne algorytmy kompresji metodą *Huffmana* i *LZW*.

Rozdział 14. Zadania różne

Zestaw różnorodnych zadań, które nie zmieściły się w tekście głównym książki. Oczywiście z rozwiązaniami!

Dodatki

W dodatkach opisałem podstawową składnię języka C++ (w formie porównania z językiem Pascal, często używanym do opisywania algorytmów) oraz omówiłem popularne systemy kodowania (dwójkowy i szesnastkowy) przydatne każdej osobie zainteresowanej programowaniem.

Dodatek C zawiera szczegóły dotyczące kompilowania i uruchamiania programów przykładowych z użyciem kompilatora C++ zawartego w GCC (GNU Compiler Collection) i przy wykorzystaniu środowiska Microsoft Visual C++ Express Edition.

Programy przykładowe

Programy opisane w książce zostały umieszczone na serwerze *ftp* wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/algo4.zip*. Wszystkie pliki źródłowe są zazwyczaj pełniejsze i bardziej rozbudowane niż warianty prezentowane w wersji drukowanej, dla oszczędności miejsca pomijałem często szczegółowe komentarze umieszczone w kodzie oraz standardowe pliki nagłówkowe. Można zatem założyć, że jeśli w trakcie wykładu jest prezentowana jakaś funkcja bez podania explicite sposobu jej użycia, to na pewno wersja źródłowa zawiera reprezentacyjny przykład jej zastosowania (przykładowa funkcja *main* i komplet funkcji nagłówkowych). Warto zatem podczas lektury porównywać wersje umieszczone na *ftp* z tymi, które zostały omówione na kartach książki!

Programy zostały przetestowane na kompilatorze GCC (sprawdzone w systemach Windows Vista i Linux — odmiana Ubuntu) oraz Microsoft Visual C++ Express Edition (świetne darmowe środowisko programowania dla systemu Windows). Wszystkie programy powinny też działać na kompilatorze Dev C++, gdyż tak naprawdę jest on tylko nakładką (IDE) na GCC.

Programy graficzne są zgodne wyłącznie z Windows, ale zostały dokładnie opisane, tak że ich przeniesienie na inny graficzny system operacyjny nie powinno sprawić kłopotu sprawnemu programiście (w tekście znajduje się tabelka wyjaśniająca działanie użytych instrukcji graficznych, tak więc nawet osoby, które nigdy nie pracowały z kompilatorem Visual C++, poradzą sobie bez problemu z analizą programów przykładowych).

Konwencje typograficzne i oznaczenia

Poniżej znajduje się kilka typowych oznaczeń i konwencji, które można napotkać na kartach tej książki.

W szczególności regułą jest, że wszystkie listingi i teksty ukazujące się na ekranie zostały odróżnione od zasadniczej treści książki czcionką o stałej szerokości znaków, to samo dotyczy komend systemu operacyjnego, jeśli takie są opisywane.



prog.cpp

Listing

Taki symbol oznacza, że tekst programu znajduje się w archiwum ZIP, zawierającym programy przykładowe, umieszczonym na serwerze *ftp* w pliku **prog.cpp**.



Ważna uwaga — materiał istotny dla zrozumienia działania omawianego zagadnienia.



Ostrzeżenie — rzeczy, których nie powinieneś robić, jeśli chcesz uniknąć kłopotów.



Odwołanie — w miejscu, na które wskazuje, znajdziesz dodatkowe informacje dotyczące omawianego zagadnienia.



Definicja lub twierdzenie.

Patrz
[Odn93]

Konwencja dotycząca odnośników bibliograficznych — oznacza odnośnik do pozycji bibliograficznej oznaczonej [Odn93] ze spisu na końcu książki.

Podziękowania na marginesie

Pierwsze wydanie tej książki powstało jako owoc mojego kilkuletniego pobytu we Francji, gdzie miałem niepowtarzalną okazję korzystania z interesujących zasobów bibliograficznych kilku bibliotek technicznych. Większość tytułów, których lektura zainspirowała mnie do napisania tej książki, była w tamtym czasie w Polsce dość trudno (jeśli w ogóle) dostępna i głównym celem książki było wypełnienie pewnej luki na naszym rynku wydawniczym.

Wstępny wydruk (jeszcze w pieluchach) pierwszego wydania książki został przejrany i opatrzone wieloma cennymi uwagami przez Zbyszka Chamskiego. Wydanie drugie zostało gruntownie zrecenzowane przez Pana Tomasza Zielińskiego, jego ogromna korekta została w znacznej mierze uwzględniona w wydaniu trzecim. Bieżące, czwarte wydanie uwzględnia liczne komentarze Czytelników, w szczególności w zakresie jakości kodu C++ i jego zgodności z popularnymi kompilatorami. Wszystkim osobom, które poświęciły swój czas na recenzje, serdecznie dziękuję!

Uwagi do wydania IV

W bieżącej edycji książki całość tekstu została gruntownie przejrana i poprawiona, starałem się usunąć zgłoszone niejasności, przededagowałem nieczytelne fragmenty, poprawiłem niektóre ilustracje oraz usunąłem błędy w treści i listingach, które znaleźli czujni Czytelnicy. Same listingi zostały poprawione pod kątem czystości języka C++, tak aby umożliwić bezproblemową kompilację na różnych kompilatorach i systemach operacyjnych. Część skorygowanych błędów wynikała z zaszłości historycznych (pierwsze wydanie ukazało się w 1995 roku), a część była ewidentnie moją winą lub błędami składu (wydanie drugie było konwertowane z formatu LaTeX na Word, co doprowadziło do błędów podczas przenoszenia danych). Za wszystkie kłopoty z kodami źródłowymi serdecznie przepraszam, a w ramach drobnej rekompensaty w obecnym wydaniu postanowiłem też pomóc Czytelnikom w samym procesie programowania, pokazując, jak użyć kompilatora GCC lub Visual C++ Express Edition.

Choć struktura książki nie uległa zmianie, to uzupełniona została treść wielu rozdziałów; największe zmiany zasły w: rozdziale 3. (analiza złożoności algorytmów), 4. (doszło sortowanie zewnętrzne i systemowe), 5. (nieco praktycznych uwag na temat języka C++ użytego w kontekście struktur danych), 10. (nowe algorytmy grafowe).

Wzbogacona została także zawartość programów przykładowych dostępnych na *ftp*, m.in. programy graficzne z rozdziału 2. zostały przygotowane w wersjach dla systemu Windows, doszły nowe programy jako efekt rozszerzenia treści książki: *ftp://ftp.helion.pl/przyklady/algo4.zip*.

Licząc, że dokonane poprawki wpłyną pozytywnie na jakość publikacji, życzę przyjemnej i pożytecznej lektury.

Piotr Wróblewski

Maj 2009

Rozdział 1.

Zanim wystartujemy

Zanim na dobre rozpoczniemy operowanie takimi pojęciami, jak wspomniany we wstępie „algorytm”, warto przedyskutować dokładnie, co przez nie rozumiemy¹.



Definicja

Algorytm to:

- ◆ skończony ciąg/sekwencja reguł, które aplikuje się na skończonej liczbie danych, pozwalający rozwiązywać zbliżone do siebie klasy problemów;
- ◆ zespół reguł charakterystycznych dla pewnych obliczeń lub czynności informatycznych.

Cóż, definicje powyższe wydają się klarowne i jasne, jednak obejmują na tyle rozległe obszary działalności ludzkiej, że daleko im do precyzji. Pomijając chwilowo znaczenie, samo pochodzenie terminu algorytm nie zawsze było do końca jasne. Dopiero specjaliści zajmujący się historią matematyki odnaleźli najbardziej prawdopodobny źródłosłów: termin ten pochodzi od nazwiska perskiego pisarza matematyka Muhammada ibn Musa al-Chuwarizmi² (żył w IX wieku n.e.). Jego zasługą jest dostarczenie klarownych reguł wyjaśniających krok po kroku zasady operacji arytmetycznych wykonywanych na liczbach dziesiętnych.

Słowo algorytm jest często łączone z imieniem greckiego matematyka Euklidesa (365 — 300 p.n.e.) i jego słynnym przepisem na obliczanie największego wspólnego dzielnika dwóch liczb a i b (NWD):

```
dane wejściowe:  $a$  i  $b$ , zmienne pomocnicze:  $c$ ,  $res$ 
dopóki  $a > 0$  wykonuj:
  podstaw za  $c$  resztę z dzielenia  $a$  przez  $b$ ;
  podstaw za  $b$  liczbę  $a$ ;
  podstaw za  $a$  liczbę  $c$ ;
  podstaw za  $res$  liczbę  $b$ ;
rezultat:  $res$ .
```

Oczywiście Euklides nie proponował swojego algorytmu dokładnie w ten sposób (w miejsce funkcji reszty z dzielenia stosowane były sukcesywne odejmowania), ale jego pomysł można zapisać w powyższy sposób bez szkody dla wyniku, który w każdym przypadku będzie taki sam. Nie jest to, rzecz jasna, jedyny algorytm, z którym mieliśmy w swoim życiu do czynienia. Każdy z nas z pewnością umie zaparzyć kawę:

¹ Definicja pochodzi ze słownika *Le Nouveau Petit Robert* (Dictionnaires le Robert — Paris 1994) — tłumaczenie własne.

² Jego nazwisko pisane było po łacinie jako *Algorismus*, pisownia w tym wydaniu książki jest zgodna z encyklopedią PWN.

- ◆ włączyć gaz;
- ◆ zagotować niezbędną ilość wody;
- ◆ wsypać zmieloną kawę do szklanki;
- ◆ zalać kawę wrzącą wodą;
- ◆ osłodzić do smaku;
- ◆ poczekać, aż odpowiednio naciągnie.

Powyższy przepis działa, ale zawiera kilka słabych punktów: co to znaczy „odpowiednia ilość wody”? Co dokładnie oznacza stwierdzenie „osłodzić do smaku”? Przepis przygotowania kawy ma cechy algorytmu (rozumianego w sensie zacytowanych wyżej definicji słownikowych), ale brak mu precyzji niezbędnej do wpisania go do jakiejś maszyny, tak aby w każdej sytuacji umiała ona sobie poradzić z poleceniem „przygotuj mi małą kawę”. (Np. jak w praktyce określić warunek, że kawa „odpowiednio naciągnęła”?).

Jakie w związku z tym cechy powinny być przypisane algorytmowi rozumianemu w kontekście informatycznym? Dyskusję na ten temat można by prowadzić dość długo, ale przyjmując pewne uproszczenia, można zadowolić się następującymi wymogami wyszczególnionymi poniżej.

Każdy algorytm:

- ◆ posiada dane wejściowe (w ilości większej lub równej zero) pochodzące z dobrze zdefiniowanego zbioru (np. algorytm Euklidesa operuje na dwóch liczbach całkowitych);
- ◆ produkuje pewien wynik (niekoniecznie numeryczny);
- ◆ jest precyzyjnie zdefiniowany (każdy krok algorytmu musi być jednoznacznie określony);
- ◆ jest skończony (wynik algorytmu musi zostać kiedyś dostarczony — mając algorytm A i dane wejściowe D , powinno być możliwe precyzyjne określenie czasu wykonania $T(A)$);
- ◆ daje się zastosować do rozwiązywania całej klasy zagadnień, a nie tylko jednego konkretnego zadania.

Ponadto niecierpliwość każe nam szukać algorytmów efektywnych, tzn. wykonujących swoje zadanie w jak najkrótszym czasie i wykorzystujących jak najmniejszą ilość pamięci (do tej tematyki powrócimy jeszcze w rozdziale 3.). Zanim jednak pospieszymy do klawiatury, aby wpisywać do pamięci komputera programy spełniające powyższe założenia, popatrzymy na algorytmikę z perspektywy historycznej.

Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych

Cytowane na samym początku tego rozdziału imiona matematyków kojarzonych z algorytmiką rozdzielone są ponad tysiącem lat i mogą łatwo zasugerować, że ta gałąź wiedzy przeżywała w ciągu wieków istnienia ludzkości burzliwy i błyskotliwy rozwój. Oczywiście nijak się to ma do rzeczywistego postępu tej dziedziny, który był i ciągle jest ściśle związany z rewolucją techniczną dokonującą się na przestrzeni zaledwie ostatnich dwustu lat. Owszem, jeśli zechcemy traktować informatykę i algorytmikę jako pewną całość wywodzącą się naturalnie z systemów obliczeniowych, to warto wspomnieć o osiągnięciach ludów sumeryjskich, wynalazców tabliczek obliczeniowych, własnego kalendarza i sześćdziesiątego systemu pomiarowego (24-godzinna doba to ich wynalazek). Znani są też Chińczycy, wynalazcy Abakusa, czyli najśłynniejszego liczydła w historii ludzkości, choć mało kto ma świadomość, że praktycznie każdy w miarę cywilizowany lud dopracowywał się jakiegoś systemu wspomagającego obliczenia i trudno tu oddawać komuś palmę pierwszeństwa. Ponadto, licytując się tego typu faktami, łatwo cofniemy się do okresu datowanego na 2 – 3 tysiące lat p.n.e., tylko czy to ma obecnie wartość inną niż ciekawostka?

Aby nie zamieniać tego rozdziału w podręcznik historii, pominię rozważania na temat maszyny do dodawania Blaise’a Pascala (ok. 1645) lub jej anglosaskiego odpowiednika, skonstruowanego w niemal tym samym czasie przez G. W. Leibniza. Popatrzmy jedynie na kilka charakterystycznych wydarzeń związanych z wynalazkami, które nie tylko ułatwiały obliczanie, ale i pozwalały na elementarne programowanie, czyli coś, co już jednoznacznie kojarzy nam się z komputerami i algorytmami.

— 1801 —

Francuz Joseph Marie Jacquard wynajduje krosno tkackie, w którym wzorec tkaniny był „programowany” na swego rodzaju kartach perforowanych. Proces tkania był kontrolowany przez algorytm zakodowany w postaci sekwencji otworów wybitych w karcie. Sam pomysł był wynikiem wielu lat pracy Jacquarda i mógł ujrzeć światło dzienne dzięki przypadkowi, jakim było uczestnictwo w konkursie państwowym, na którym przedstawił maszynę do robienia sieci rybackich.

Koncepcja Jacquarda zainteresowała francuskiego matematyka, L.M. Carnota, który ściągnął go do Paryża w celu kontynuowania badań i pomógł w uzyskaniu stypendium rządowego. Pierwsze prace omal nie doprowadziły do śmierci wynalazcy, rozwścieczeni tkacze niemal utopili go w Rodanie, przeczuwając, że jego maszyna stanowi zagrożenie dla ich zatrudnienia (a dokładniej: dla zatrudnienia ich dzieci, które do tej pory służyły za pomocników podnoszących nitki, aby umożliwić utkanie lub nie odpowiedniego wzoru przez przesuwającą się poprzecznie cewkę z nitką — wynalazek Jacquarda eliminował pięć stanowisk pracy przy jednym krośnie!). Pomysł Jacquarda był dopasowany do ówczesnych możliwości technicznych, ale warto zauważyć, że karta perforowana z zakodowaną logiką dwustanową (dziurka lub jej brak oznaczał dla maszyny tkackiej podjęcie lub nie odpowiedniej akcji natury mechanicznej) jest prekursorem współczesnych pamięci, w których za pomocą liczb binarnych koduje się odpowiednie akcje algorytmu!

— 1833 —

Anglik Charles Babbage częściowo buduje maszynę do wyliczania niektórych formuł matematycznych. W czasach, w których żył Babbage, nastąpiła eksplozja zastosowań matematyki (astronomia, nawigacja), a nie istniały metody wspomagające obliczenia w sposób automatyczny. Babbage był autorem koncepcji tzw. *maszyny analitycznej*, zbliżonej do swego poprzedniego dzieła, ale wyposażonej w możliwość przeprogramowywania, jak w przypadku maszyny Jacquarda.

W pewnym uproszczeniu maszyna ta miała składać się z magazynu (odpowiednik pamięci realizowanej jako kolumny kół, później zastąpionej bębnem), młyna (jednostka obliczeniowa wykonująca operacje dzięki obrotom kół i przekładni) i mechanizmu sterującego wykorzystującego karty perforowane (Jacquard!). Czyż nie przypomina to schematu komputera?

Opisy maszyny Babbage’a były na tyle dokładne, że matematyczka Ada Lovelace, córka lorda Byrona, opracowała pierwsze teoretyczne „programy” dla tej nieistniejącej jeszcze maszyny, stając się pierwszą uznaną... programistką w historii informatyki³!

Wymagania natury mechanicznej, jakie stawiała ta maszyna, pozwoliły na skonstruowanie jej pierwszego prototypu dopiero w dwadzieścia lat od narodzin samej idei, a sama maszyna powstała dopiero w roku... 1992, oczywiście bardziej jako ciekawostka niż potrzeba naukowa.

— 1890 —

W zasadzie pierwsze publiczne i na dużą skalę użycie maszyny bazującej na kartach perforowanych. Chodzi o maszynę do opracowywania danych statystycznych, dzieło Amerykanina Hermana Holleritha, użyte przy dokonywaniu spisu ludności. (Na marginesie warto dodać, że

³ Od jej imienia pochodzi nazwa języka programowania ADA.

przedsiębiorstwo Holleritha przekształciło się w 1911 roku w *International Business Machines Corp.*, bardziej znane jako IBM, będące do dziś czołowym producentem komputerów).

— lata trzydzieste —

Rozwój badań nad teorią algorytmów (plejada znanych matematyków: Turing, Gödel, Markow). Z tego okresu wywodzi się słynne zagadnienie postawione przez pruskiego⁴ matematyka Dawida Hilberta, który w 1928 roku na międzynarodowym kongresie matematyków publicznie postawił pytanie, czy istnieje metoda pozwalająca rozstrzygnąć prawdziwość dowolnego twierdzenia matematycznego, w wyniku li tylko mechanicznych operacji na symbolach. Studentom informatyki bliskie będzie pojęcie tzw. maszyny Turinga, abstrakcyjnej maszyny obliczeniowej złożonej z głowicy czytająco-piszącej oraz nieskończonej taśmy zawierającej symbole (np. liczby lub operatory działań). Ten abstrakcyjny model matematyczny stworzył podwaliny pod współczesne komputery. W ramach tej książki Czytelnik powinien zapamiętać tylko, że to, co określa się nieco myląc terminem *maszyna*, jest wyłącznie *modelem* schematu działania wg zadanego algorytmu.

— lata czterdzieste —

Budowa pierwszych komputerów ogólnego przeznaczenia (głównie dla potrzeb obliczeniowych wynikłych w tym wojennym okresie: badania nad łamaniem kodów, początek „kariery” bomby atomowej).

Pierwszym urządzeniem, które można określić jako „komputer”, był automatyczny kalkulator MARK 1, skonstruowany w 1944 roku (jeszcze na przekątnikach, czyli jako urządzenie elektromechaniczne). Jego twórcą był Amerykanin Howard Aiken z uniwersytetu Harvarda. Aiken bazował na idei Babbage’a, która musiała czekać 100 lat na swoją praktyczną realizację! W dwa lata później powstaje pierwszy elektroniczny komputer ENIAC⁵ (jego wynalazcy: J. P. Eckert i J. W. Mauchly z uniwersytetu Pensylwania), który miał oryginalnie wspomagać obliczenia balistyczne.

Powszechnie jednak za pierwszy komputer w pełnym tego słowa znaczeniu uważa się EDVAC⁶ zbudowany na uniwersytecie w Princeton. Jego wyjątkowość polegała na umieszczeniu programu wykonywanego przez komputer całkowicie w jego pamięci, podobnie jak i pamięci do przechowywania wyników obliczeń. Autorem tej przełomowej idei był matematyk Johannes von Neumann (Amerykanin węgierskiego pochodzenia)⁷.

— okres powojenny —

Prace nad komputerami prowadzone są w wielu krajach równolegle. W grę zaczyna wchodzić wkroczenie na nowo powstały obiecujący rynek komputerów (kończy się bowiem era budowania unikalnych uniwersyteckich prototypów). Na rynku pojawiają się kalkulatory IBM 604 i BULL Gamma 3, a następnie duże komputery naukowe, np. UNIVAC 1 i IBM 650. Zaczynającej się zarysowywać dominacji niektórych producentów usiłują przeciwdziałać badania prowadzone w wielu krajach (mniej lub bardziej systematycznie i z różnorakim poparciem polityków), ale... to już jest temat na osobną książkę!

⁴ Urodzony w Königsbergu, obecnie zwanym Kaliningradem.

⁵ Ang. *Electronic Numerical Interpreter And Calculator*.

⁶ Ang. *Electronic Discrete Variable Automatic Computer*.

⁷ Koncepcja komputera została opracowana w 1946 roku, jednak jego pierwsza realizacja praktyczna powstała dopiero w roku 1956.

— teraz —

Burzliwy rozwój elektroniki powoduje masową, do dziś trwającą komputeryzację wszelkich dziedzin życia. Komputery stają się czymś powszechnym i niezbędnym, wykonując tak różnorodne zadania, jak tylko każe im to wyobraźnia ludzka.

Jak to się niedawno odbyło, czyli o tym, kto „wymyślił” metodologię programowania

Zamieszczone w poprzednim paragrafie kalendarium zostało doprowadzone do momentu, w którym programiści zaczęli mieć do dyspozycji komputery z prawdziwego zdarzenia. Olbrzymi nacisk, jaki był kładziony na rozwój sprzętu, w istocie doprowadził do znakomitych rezultatów — efekt jest widoczny dzisiaj w praktycznie każdym biurze i w coraz większej liczbie domów prywatnych.

W latach sześćdziesiątych zaczęto konstruować pierwsze naprawdę duże systemy informatyczne — w sensie ilości kodu, głównie assemblerowego, wyprodukowanego na poczet danej aplikacji. Ponieważ jednak programowanie było ciągle traktowane jako działalność polegająca głównie na intuicji i wyczuciu, zdarzały się całkiem poważne wpadki w konstrukcji oprogramowania: albo szybko tworzone były systemy o małej wiarygodności, albo też nakład pieniędzy włożonych w rozwój produktu znacznie przewyższał szacowane wydatki i stawał pod znakiem zapytania sens podjętego przedsięwzięcia. Brak było zarówno metod, jak i narzędzi umożliwiających sprawdzanie poprawności programowania. Powszechną metodą programowania było testowanie programu aż do momentu jego całkowitego „odpluskwienia”⁸. Zwróćmy jeszcze uwagę, że oba wspomniane czynniki: wiarygodność systemów i poziom nakładów są niezmiernie ważne w praktyce; informatyczny system bankowy musi albo działać stuprocentowo dobrze, albo nie powinien być w ogóle oddany do użytku! Z drugiej strony poziom nakładów przeznaczonych na rozwój oprogramowania nie powinien odbić się niekorzystnie na kondycji finansowej przedsiębiorstwa.

W pewnym momencie sytuacja stała się tak krytyczna, że zaczęto wręcz mówić o kryzysie w rozwoju oprogramowania! W 1968 roku została nawet zwołana konferencja NATO (Garmisch, Niemcy) poświęcona przedyskutowaniu zaistniałej sytuacji. W rok później w ramach IFIP (ang. *International Federation for Information Processing*) została utworzona specjalna grupa robocza pracująca nad tzw. metodologią programowania.

Z historycznego punktu widzenia dyskusja na temat udowadniania poprawności algorytmów zaczęła się jednak od artykułu Johna McCarthy’ego „A basis for a mathematical theory of computation”, gdzie padło zdanie: „zamiast sprawdzania programów komputerowych metodą prób i błędów aż do momentu ich całkowitego odpluskwienia powinniśmy udowadniać, że posiadają one pożądane własności”. Nazwiska ludzi, którzy zajmowali się teoretycznymi pracami na temat metodologii programowania, nie znikły bynajmniej z horyzontu: Dijkstra, Hoare, Floyd, Wirth itd. (będą oni jeszcze nieraz cytowani w tej książce!).

Krótką prezentacją, której dokonaliśmy w poprzednich dwóch paragrafach, ukazuje dość zaskakującą młodość algorytmiki jako dziedziny wiedzy. Warto również zauważyć, że nie jest to nauka, która powstała samorodnie. O ile obecnie należy ją odróżniać jako odrębną gałąź wiedzy, to nie sposób nie docenić wielowiekowej pracy matematyków, którzy dostarczyli algorytmice zarówno narzędzi opisu zagadnień, jak i wielu użytecznych teoretycznych rezultatów. (Powyższa uwaga dotyczy się również wielu innych dziedzin wiedzy).

Teraz, gdy już zdefiniowaliśmy sobie głównego bohatera tej książki (bohatera zbiorowego: chodzi bowiem o algorytmy!), przejrzymy kilka sposobów używanych do jego opisu.

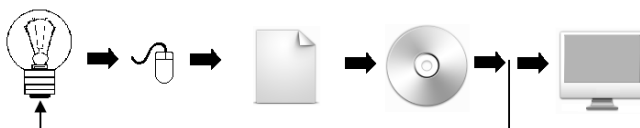
⁸ Żargonowe określenie procesu usuwania błędów z programu.

Proces koncepcji programów

W poprzednim paragrafie wyszczególniliśmy kilka cech charakterystycznych, które powinien posiadać algorytm rozumiany jako pojęcie informatyczne. Szczególny nacisk położony został na precyzję zapisu. Wymóg ten jest wynikiem ograniczeń narzuconych przez współcześnie istniejące komputery i kompilatory — nie są one bowiem w stanie rozumieć poleceń nieprecyzyjnie sformułowanych, zbudowanych niezgodnie z wbudowanymi w nie wymogami syntaktycznymi.

Rysunek 1.1 w sposób uproszczony obrazuje etapy procesu programowania komputerów. Olbrzymia żarówka symbolizuje etap, który jest od czasu do czasu pomijany przez programistów (dodajmy, że zwykle z opłakanymi skutkami) — REFLEKSJĘ.

Rysunek 1.1.
Etapy konstrukcji programu



Następnie jest tworzony tzw. tekst źródłowy nowego programu, mający postać pliku tekstowego, wprowadzanego do komputera za pomocą zwykłego edytora tekstowego. Większość istniejących obecnie kompilatorów posiada taki edytor już wbudowany, więc użytkownik w praktyce nie opuszcza tzw. środowiska zintegrowanego, grupującego programy niezbędne w procesie programowania. Ponadto niektóre środowiska zintegrowane zawierają zaawansowane edytory graficzne umożliwiające przygotowanie zewnętrznego interfejsu użytkownika praktycznie bez pisania ani jednej linii kodu. Pomijając już jednak tego typu szczegóły, generalnie efektem pracy programisty jest plik lub zespół plików opisujących w formie symbolicznej sposób zachowania się programu wynikowego. Opis ten jest kodowany w tzw. języku programowania, który stanowi na ogół podzbiór języka naturalnego⁹. Kompilator dokonuje mniej lub bardziej zaawansowanej analizy poprawności i, jeśli wszystko jest w porządku, produkuje tzw. *kod wykonywalny*, zapisany w postaci zrozumiałej dla komputera. Plik zawierający kod wykonywalny może być następnie wykonywany pod kontrolą systemu operacyjnego komputera (który to system notabene jest także złożony z wielu pojedynczych programów). Kod wykonywalny jest specyficzny dla danego systemu operacyjnego. W ostatnich latach rozpowszechnił się język Java, który pozwala budować programy niezależne od systemów operacyjnych, ale dzieje się to na zasadzie pewnego „oszustwa”: kod wykonywalny nie jest uruchamiany bezpośrednio przez system operacyjny, tylko poprzez specjalne środowisko uruchomieniowe, które izoluje go od sprzętu i systemu, wprowadzając jako warstwę pośrednią opóźnienia niezbędne dla dokonania translacji kodu pośredniego na kod finalny.

Gdzie w tym procesie umiejscowione jest to, co stanowi tematykę książki, którą trzymasz, Czytelniku, w ręku? Otóż z całego skomplikowanego procesu tworzenia oprogramowania zajmujemy się tym, co do tej pory nie jest (jeszcze?) zautomatyzowane: koncepcją algorytmów, ich jakością i technikami programowania aktualnie używanymi w informatyce. Będziemy anonsować pewne problemy dające się rozwiązać za pomocą komputera, a następnie omówimy sobie, jak to zadanie wykonać w sposób efektywny. Tworzenie zewnętrznej otoczki programów, czyli tzw. *interfejsu użytkownika*, nie wchodzi w zakres tematyczny tej książki.

⁹ W praktyce jest to język angielski.

Poziomy abstrakcji opisu i wybór języka

Jednym z delikatniejszych problemów związanych z opisem algorytmów jest sposób ich prezentacji zewnętrznej. Można w tym celu przyjąć dwie skrajne pozycje:

- ♦ zbliżyć się do maszyny (język assemblera: nieczytelny dla nieprzygotowanego odbiorcy);
- ♦ zbliżyć się do człowieka (opis słowny: maksymalny poziom abstrakcji zakładający poziom inteligencji odbiorcy niemożliwy aktualnie do wbudowania w maszynę¹⁰).

Wybór języka assemblera do prezentacji algorytmów wymagałby w zasadzie związania się z określonym typem maszyny, co zlikwidowałoby jakąkolwiek ogólność rozważań i uczyniłoby opis trudnym do analizy. Z drugiej zaś strony opis słowny wprowadza ryzyko niejednoznaczności, która może być kosztowna: program, po przetłumaczeniu go na postać zrozumiałą dla komputera, może nie zadziałać!

Aby zaradzić zaanonsowanemu wyżej problemom, zwyczajowo przyjęło się prezentowanie algorytmów w dwojaki sposób:

- ♦ za pomocą istniejącego języka programowania;
- ♦ używając pseudojęzyka programowania (mieszanki języka naturalnego i form składniowych pochodzących z kilku reprezentatywnych języków programowania).

W niniejszym podręczniku można napotkać obie te formy i wybór któregoś z nich został podyktowany kontekstem omawianych zagadnień. Przykładowo: jeśli dany algorytm jest możliwy do czytelnego zaprezentowania za pomocą języka programowania, wybór będzie oczywisty! Od czasu do czasu jednak napotkamy sytuacje, w których prezentacja kodu w pełnej postaci, gotowej do wprowadzenia do komputera, byłaby zbędna (np. zbliżony materiał był już przedstawiony wcześniej) lub nieczytelna (liczba linii kodu przekracza objętość jednej strony). W każdym jednak przypadku ewentualne przejście z jednej formy w drugą nie powinno stanowić dla Czytelnika większego problemu.

Już we wstępie zostało zdradzone, iż językiem prezentacji programów będzie C++. Pora zatem dokładniej wyjaśnić powody, które przemawiały za tym wyborem. C++ jest językiem programowania określanym jako *strukturalny*, co z założenia ułatwia pisanie w nim w sposób czytelny i zrozumiały. Związek tego języka z klasycznym C umożliwia oczywiście tworzenie absolutnie nieczytelnych listingów, będziemy tego jednak starannie unikać. W istocie częstokroć będą omijane pewne możliwe mechanizmy optymalizacyjne, aby nie zatracić prostoty zapisu. Najważniejszym jednak powodem użycia C++ jest fakt, iż ułatwia on programowanie na wielu poziomach abstrakcji. Istnienie klas i wszelkie obiektowe cechy tego języka powodują, że zarówno ukrywanie szczegółów implementacyjnych, jak i rozszerzanie już zdefiniowanych modułów (bez ich kosztownego „przepisywania”) jest bardzo łatwe, a są to właściwości, którymi nie można pogardzić.

Być może cenne będzie podkreślenie usługowej roli, jaką w procesie programowania pełni wybrany do tego celu język. Wiele osób pasjonuje się wykazywaniem wyższości jednego języka nad drugim, co jest sporem tak samo jałowym, jak wykazywanie „wyższości świąt Wielkiej Nocy nad świętami Bożego Narodzenia” (choć zapewne mniej zabawnym...). Język programowania jest w końcu tylko narzędziem, ulegającym zresztą znacznej (r)ewolucji na przestrzeni ostatnich lat. Pracując nad pewnymi partiami tej książki, musiałem od czasu do czasu zwalczać silną pokusę prezentowania niektórych algorytmów w takich językach jak LISP czy PROLOG.

Uprościłoby to znacznie wszelkie rozważania o listach i rekurencji — niestety ograniczyłoby również potencjalny krąg odbiorców książki do ludzi profesjonalnie związanych wyłącznie z informatyką.

¹⁰ Niemowładzi sobie bez trudu z problemami, nad którymi biedzą się specjaliści od tzw. sztucznej inteligencji, usiłujący je rozwiązywać za pomocą komputerów! (Chodzi o efektywność uczenia się, rozpoznawanie form, etc.).

Zdając sobie sprawę, że C++ może być pewnej grupie Czytelników nieznany, przygotowałem dla nich w dodatku A minikurs tego języka. Polega on na równoległej prezentacji struktur składniowych w C++ i Pascalu, tak aby poprzez porównywanie fragmentów kodu nauczyć się czytania listingów prezentowanych w tej książce. Kilkustronicowy dodatek nie zastąpi oczywiście podręcznika poświęconego tylko i wyłącznie C++, umożliwi jednak lekturę książki osobom pragnącym z niej skorzystać bez konieczności poznawania nowego języka.

Poprawność algorytmów

Wpisanie programu do komputera, skompilowanie go i uruchomienie jeszcze nie gwarantują, że kiedyś nie nastąpi jego załamanie (cokolwiek by to miało znaczyć w praktyce). O ile jednak w przypadku niewinnych domowych aplikacji nie ma to specjalnego znaczenia (w tym sensie, że tylko my ucierpimy), to w momencie zamierzonej komercjalizacji programu sprawa znacznie się komplikuje. W grę zaczyna wchodzić nie tylko kompromitacja programisty, ale i jego odpowiedzialność za ewentualne szkody poniesione przez użytkowników programu.

Od błędów w swoich produktach nie ustrzegają się nawet wielkie koncerny programistyczne — w miesiąc po kampanii reklamowej produktu X pojawiają się po cichu „darmowe” (dla legalnych użytkowników) uaktualnione wersje, które nie mają wcześniej niezauważonych błędów. Popularne systemy operacyjne, takie jak np. Windows lub Mac OS, posiadają wbudowane mechanizmy automatycznej aktualizacji przez Internet, które służą naprawianiu wadliwych funkcji systemu lub bieżącej reakcji na zagrożenia (np. wirusy).

Mamy tu do czynienia z pośpiechem, którego celem jest wyprzedzenie konkurencji, co usprawiedliwia wypuszczanie przez dyrekcje firm niedopracowanych produktów — ze szkodą dla użytkowników niemających żadnych możliwości obrony przed tego typu praktykami. Z drugiej jednak strony uniknięcie błędów w programach wcale nie jest problemem banalnym i stanowi temat poważnych badań naukowych¹¹!

Zajmijmy się jednak czymś bliższym rzeczywistości typowego programisty: pisze on program i chce uzyskać odpowiedź na pytanie: „Czy będzie on działał poprawnie w każdej sytuacji, dla każdej możliwej konfiguracji danych wejściowych?”. Odpowiedź jest tym trudniejsza, im bardziej skomplikowane są procedury, które zamierzamy badać. Nawet w przypadku pozornie krótkich w zapisie programów liczba sytuacji, które mogą zaistnieć w praktyce, wyklucza ręczne przetestowanie programu. Pozostaje więc stosowanie dowodów natury matematycznej, zazwyczaj dość skomplikowanych. Jedną z możliwych ścieżek, którymi można dojść do stwierdzenia formalnej poprawności algorytmu, jest stosowanie metody *niezmienników* (zwanej niekiedy metodą *Floyda*). Mając dany algorytm, możemy łatwo wyróżnić w nim pewne kluczowe punkty, w których dzieją się interesujące dla danego algorytmu rzeczy. Ich znalezienie nie jest zazwyczaj trudne: ważne są momenty inicjalizacji zmiennych, którymi będzie operować procedura, testy zakończenia algorytmu, pętla główna itd. W każdym z tych punktów możliwe jest określenie pewnych zawsze prawdziwych warunków — tzw. *niezmienników*. Można sobie zatem wyobrazić, że dowód formalnej poprawności algorytmu może być uproszczony do stwierdzenia zachowania prawdziwości niezmienników dla dowolnych danych wejściowych.

¹¹ Formalne badanie poprawności systemów algorytmicznych jest możliwe przy użyciu specjalistycznych języków stworzonych do tego celu.

Dwa typowe sposoby stosowane w praktyce to:

- ♦ sprawdzanie stanu punktów kontrolnych za pomocą *debuggera* (odczytujemy wartości pewnych ważnych zmiennych i sprawdzamy, czy zachowują się poprawnie dla pewnych reprezentacyjnych danych wejściowych¹²).
- ♦ formalne udowodnienie (np. przez indukcję matematyczną) zachowania niezmienników dla dowolnych danych wejściowych.

Zasadniczą wadą powyższych zabiegów jest to, że są one nużące i potrafią łatwo zabić całą przyjemność związaną z efektywnym rozwiązywaniem problemów za pomocą komputera. Tym niemniej Czytelnik powinien być świadom istnienia również i tej strony programowania. Jedną z prostszych (i bardzo kompletnych) książek, którą można polecić Czytelnikowi zainteresowanemu formalną teorią programowania, metodami generowania algorytmów i sprawdzania ich własności, jest [Gri84] — entuzjastyczny wstęp do niej napisał sam Dijkstra¹³, co jest chyba najlepszą rekomendacją dla tego typu pracy. Inny tytuł o podobnym charakterze, [Kal90], można polecić miłośnikom formalnych dowodów i myślenia matematycznego. Metody matematycznego dowodzenia poprawności algorytmów są prezentowane w tych książkach w pewnym sensie niejawnie; zasadniczym celem jest dostarczenie narzędzi, które umożliwią quasi-automatyczne generowanie algorytmów.

Każdy program wyprodukowany za pomocą tych metod jest automatycznie poprawny — pod warunkiem, że nie został po drodze popełniony jakiś błąd. Wygenerowanie algorytmu jest możliwe dopiero po jego poprawnym zapisaniu wg schematu:

{warunki wstępne¹⁴} **poszukiwany program** {warunki końcowe}

Przy pewnej dozie doświadczenia możliwe jest wyprodukowanie ciągu instrukcji, które powodują przejście z „warunków wstępnych” do „warunków końcowych” — wówczas formalny dowód poprawności algorytmu jest zbędny. Można też podejść do problemu z innej strony, mając dany zespół warunków wstępnych i pewien program: czy jego wykonanie zapewnia ustawienie pożądanym warunków końcowych?

Czytelnik może nieco się obruszyć na ogólnikowość powyższego wywodu, ale jest ona wymuszona przez rozmiar tematu, który wymaga w zasadzie osobnej książki! Pozostaje zatem tylko ponowić zaproszenie do lektury niektórych zacytowanych wyżej pozycji bibliograficznych — niestety w momencie pisania tej książki niedostępnych w polskich wersjach językowych.

¹² Stwierdzenia: „ważne zmienne”, „poprawne” zachowanie programu, „reprezentatywne” dane wejściowe, etc. należą do gatunku bardzo nieprecyzyjnych i są ściśle związane z konkretnym programem, którego analizą się zajmujemy.

¹³ Jeśli już jesteśmy przy tym autorze, to warto polecić przynajmniej pobieżną lekturę [DF89], która stanowi dość dobry wstęp do metodologii programowania.

¹⁴ Wartości zmiennych, pewne warunki logiczne je wiążące, etc.

Rozdział 2.

Rekurencja

Tematem niniejszego rozdziału jest jeden z najważniejszych mechanizmów używanych w informatyce — *rekurencja*, zwana również *rekursją*¹. Mimo iż użycie rekurencji nie jest obowiązkowe, jej zalety są oczywiste dla każdego, kto choć raz spróbował tego stylu programowania. Wbrew pozorom nie jest to wcale mechanizm prosty i szereg jego aspektów wymaga dogłębnej analizy. Rekurencja upraszcza jednak opis wielu zagadnień (np. pozwala na łatwe definiowanie struktur opartych na fraktalach), a w wielu zagadnieniach informatycznych, np. w strukturach „drzewiastych”, jest wręcz niezbędna z uwagi na ich charakter.

Niniejszy rozdział ma kluczowe znaczenie dla pozostałej części książki — o ile jej lektura może być dość swobodna i nieograniczona naturalną kolejnością rozdziałów, o tyle bez dobrego zrozumienia samej istoty rekurencji nie będzie możliwe swobodne czytanie wielu zaprezentowanych dalej algorytmów i metod programowania.

Definicja rekurencji

Rekurencja jest często przeciwstawiana podejściu iteracyjnemu, czyli n -krotnemu wykonywaniu algorytmów w taki sposób, aby wyniki uzyskane podczas poprzednich iteracji (zwanymi też „przebiegami”) mogły służyć jako dane wejściowe do kolejnych. Sterowanie iteracjami zapewniają instrukcje pętli (np. `for` lub `while`). Rekurencja działa podobnie, tylko funkcję zapętlania pełni wywoływanie się tej samej procedury (funkcji) przez siebie samą, z innymi parametrami. Oczywiście w ciele procedury rekurencyjnej też można spotkać klasyczne pętle, ale pełnią one rolę usługową i nie stanowią kryterium klasyfikacji algorytmu.

Warto wiedzieć, że programy zapisane w formie rekurencyjnej mogą być przekształcone — z mniejszym lub większym wysiłkiem — na klasyczną postać iteracyjną. Zagadnieniu temu poświęcę cały rozdział 6., na razie jednak zajmiemy się zrozumieniem mechanizmu rekurencji, co, jak się okazuje, nie jest takie proste, jak się wydaje na pierwszy rzut oka.

Pojęcie rekurencji poznamy na przykładzie. Wyobraźmy sobie małe dziecko w wieku lat — przykładowo — trzech. Dostaje ono od rodziców zadanie zebrania do pudełka wszystkich drewnianych klocków, które „nieumyślnie” zostały rozsypane na podłodze. Klocki są bardzo prymitywne, to zwyczajne drewniane sześcianiki, które doskonale nadają się do wznoszenia nieskomplikowanych budowli. Polecenie jest bardzo proste: „Zbierz to wszystko razem i poukładaj

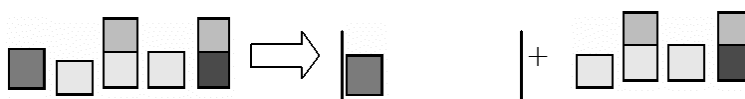
¹ Subtelna różnica między tymi pojęciami w zasadzie już się zatraciła w literaturze, dlatego też nie będziemy się niepotrzebnie zagłębiać w szczegóły terminologiczne.

tak, jak było w pudełku”. Problem wyrażony w ten sposób jest dla dziecka potwornie skomplikowany: klocków jest cała masa i niespecjalnie wiadomo, jak się do tego całościowo zabrać. Mimo ograniczonych umiejętności na pewno nie przerasta go następująca czynność: *wziąć jeden klocek z podłogi i włożyć do pudełka*. Małe dziecko, zamiast przejmować się złożonością problemu, której być może sobie nawet nie uświadamia, bierze się do pracy i rodzice z przyjemnością obserwują, jak strefa porządku na podłodze powiększa się z minuty na minutę.

Zastanówmy się chwilę nad metodą przyjętą przez dziecko: ono wie, że problem postawiony przez rodziców to wcale nie jest „zebrać wszystkie klocki” (bo to de facto jest niewykonalne za jednym zamachem), ale: „wziąć jeden klocek, przełożyć go do pudełka, a następnie zebrać do pudełka pozostałe”. W jaki sposób można zrealizować to drugie? Proste, zupełnie tak, jak poprzednio: „bierzemy jeden klocek...” itd. — postępując tak aż do momentu wyczerpania się klocków.

Spójrzmy na rysunek 2.1, który przedstawia w sposób symboliczny tok rozumowania przyjęty przy rozwiązywaniu problemu „sprzątania rozsypanych klocków”.

Rysunek 2.1.
„Sprzątanie klocków”,
czyli rekurencja
w praktyce



Jest mało prawdopodobne, aby dziecko uświadamiało sobie, że postępuje w sposób rekurencyjny, choć tak jest w istocie! Jeśli uważniej przyjrzymy się opisanemu powyżej problemowi, to zauważymy, że jego rozwiązanie charakteryzuje się następującymi, typowymi dla algorytmów rekurencyjnych, cechami:

- ◆ Zakończenie algorytmu jest jasno określone („w momencie gdy na podłodze nie będzie już klocków, możesz uznać, że zadanie zostało wykonane”).
- ◆ „Duży”, złożony problem został rozłożony na problemy elementarne (które umiemy rozwiązać) i na problemy o mniejszym stopniu skomplikowania niż ten, z którym mieliśmy do czynienia na początku.

Zauważmy, że w sposób dość śmiały użyte zostało określenie „algorytm”. Czy jest sens mówić o opisanym powyżej problemie w kategorii algorytmu? Czy w ogóle możemy przypisywać pięcioletniemu dziecku wiedzę, z której nie zdaje sobie ono sprawy?

Przykład, na podstawie którego zostało wyjaśnione pojęcie algorytmu rekurencyjnego, jest niewątpliwie kontrowersyjny. Prawdopodobnie dowolny specjalista od psychologii zachowań dziecka chwyciłby się za głowę z rozpacz, czytając powyższy wywód. Dlaczego jednak zdecydowałem się na użycie takiego właśnie, a nie innego — może bardziej informatycznego — przykładu? Otóż zasadniczym celem była chęć udowodnienia, iż myślenie w sposób rekurencyjny jest jak najbardziej zgodne z naturą człowieka i duża klasa problemów rozwiązywanych przez umysł ludzki jest traktowana podświadomie w sposób rekurencyjny. Pójdźmy dalej za tym śmiałym stwierdzeniem: jeśli tylko zdecydujemy się na intuicyjne podejście do algorytmów rekurencyjnych, to nie będą one stanowiły dla nas tajemnic, choć być może na początku nie będziemy w pełni świadomi wykorzystywanych w nich mechanizmów.

Powyższe wyjaśnienie pojęcia rekurencji powinno być znacznie czytelniejsze niż typowe podejście zatrzymujące się na niewiele mówiącym stwierdzeniu, że algorytm rekurencyjny jest to algorytm, który odwołuje się do samego siebie.

Ilustracja pojęcia rekurencji

Program, którego analizą będziemy się zajmowali w tym podrozdziale, jest bardzo zbliżony do problemu klocków, z jakim spotkaliśmy się przed chwilą. Schemat rekurencyjny zastosowany w nim jest identyczny, jedynie zagadnienie jest nieco bliższe rzeczywistości informatycznej.

Mamy do rozwiązania następujący problem:

- ♦ Dysponujemy tablicą n liczb całkowitych $tab[n] = tab[0], tab[1], \dots, tab[n-1]$.
- ♦ Czy w tablicy tab występuje liczba x (podana jako parametr)?

Jak postąpiłoby dziecko z przykładu, który posłużył nam za definicję pojęcia rekurencji, zakładając oczywiście, że dysponuje już ono pewną elementarną wiedzą informatyczną? Jest wysoce prawdopodobne, że rozumowałoby w sposób następujący:

- ♦ wziąć pierwszy niezbadany element tablicy n -elementowej;
- ♦ jeśli aktualnie analizowany element tablicy jest równy x , to:

ogłoś „Sukces” i zakończ;

w przeciwnym przypadku:

zbadaj pozostałą część tablicy.

Wyżej podaliśmy warunki pozytywnego zakończenia programu. W przypadku gdy przebadaliśmy całą tablicę i element x nie został znaleziony, należy oczywiście zakończyć program w jakiś umówiony sposób — np. komunikatem o niepowodzeniu.

Proszę spojrzeć na przykładową realizację, jedną z kilku możliwych:



rek1.cpp

```
#include <iostream>
using namespace std;
const int n=10;
int tab[n]={1, 2, 3, 2, -7, 44, 5, 1, 0, -3};

void szukaj(int tab[],int left,int right,int x){
    // left, right = lewa i prawa granica obszaru poszukiwań
    // tab = tablica
    // x = wartość do odnalezienia
    if (left>right)
        cout << "Element " << x << " nie został odnaleziony\n";
    else
        if (tab[left]==x)
            cout << "Znalazłem szukany element "<< x << endl;
        else
            szukaj(tab,left+1,right,x);
}

int main()
{
    szukaj(tab,0,n-1,7);
    szukaj(tab,0,n-1,5);
}
```

Warunkiem zakończenia programu jest albo znalezienie szukanego elementu x , albo też wyjście poza obszar poszukiwań. Mimo swojej prostoty powyższy program dobrze ilustruje podstawowe, wspomniane już wcześniej cechy typowego programu rekurencyjnego. Przypatrzmy się zresztą uważniej:

- ♦ *Zakończenie* programu jest jasno określone:
 - ♦ Element zostaje znaleziony.
 - ♦ Następuje przekroczenie zakresu tablicy.
- ♦ Duży problem zostaje rozbity na problemy elementarne, które umiemy rozwiązać (patrz wyżej), i na analogiczny problem, tylko o mniejszym stopniu skomplikowania (z tablicy o rozmiarze n schodzimy do tablicy o rozmiarze $n-1$).

Na podstawie powyższych obserwacji można pokusić się o wyliczenie dwóch podstawowych błędów popełnianych przy konstruowaniu programów rekurencyjnych:

- ◆ złe określenie warunku zakończenia programu;
- ◆ niewłaściwa (nieefektywna) dekompozycja problemu.

W dalszej części rozdziału postaramy się wspólnie dojść do pewnych „zasad bezpieczeństwa”, niezbędnych przy pisaniu programów rekurencyjnych. Zanim to jednak nastąpi, konieczne będzie dokładne wyjaśnienie schematu ich wykonywania.

Jak wykonują się programy rekurencyjne?

Dociekliwy Czytelnik będzie miał prawo zapytać w tym miejscu: „OK, zobaczyłem na przykładzie, że TO działa, ale mam też chyba prawo poznać nieco bardziej od podszewki, JAK to działa!”. Pozostaje zatem podporządkować się temu słusznemu żądaniu.

Odpowiedzią na nie jest właśnie niniejszy podrozdział. Przykład w nim użyty będzie być może banalny, tym niemniej nadaje się doskonale do zilustrowania sposobu wykonywania programu rekurencyjnego.

Już w szkole średniej (lub może nawet podstawowej?!) na lekcjach matematyki dość często używa się tzw. „*silni* n ”, czyli iloczynu wszystkich liczb naturalnych od 1 do n włącznie. Ten użyteczny symbol zdefiniowany jest w sposób następujący:

$$0! = 1, \\ n! = n * (n - 1)!, \text{ gdzie } n \in N, n \geq 1$$

Podstawiając za n wartości dozwolone (czyli większe od 1), łatwo jest ręcznie wyliczyć początkowe wartości silni, dla dużych n nie jest to już takie proste, ale od czego są komputery — przecież nic nie stoi na przeszkodzie, aby napisać prosty program, który zajmuje się obliczaniem silni w sposób rekurencyjny:



rek2.cpp

```
unsigned long int silnia(int x)
{
    if (x==0)
        return 1;
    else
        return x*silnia(x-1);
}

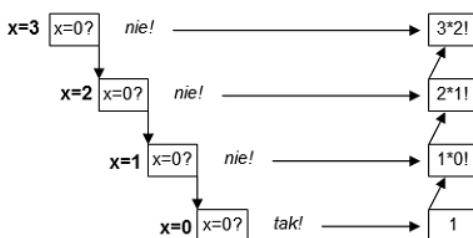
int main()
{
    cout << "silnia(5)=" << silnia(5) << endl;
}
```

Prześledźmy na przykładzie, jak się wykonuje program, który obliczy $3!$. Rysunek 2.2 przedstawia kolejne etapy wywoływania procedury rekurencyjnej i badania warunku na przypadek elementarny.

Konwencje użyte na tym rysunku są następujące:

- ◆ Pionowe strzałki w dół oznaczają zagłębianie się programu z poziomu n na $n-1$ itd. w celu dotarcia do przypadku elementarnego $0!$.
- ◆ Pozioma strzałka oznacza obliczanie wyników cząstkowych.
- ◆ Ukośna strzałka prezentuje proces przekazywania wyniku cząstkowego z poziomu niższego na wyższy.

Rysunek 2.2.
Drzewo wywołań
funkcji silnia(3)



Czymże są jednak owe tajemnicze *poziomy, przekazywanie parametrów*, etc.? Chwilowo te pojęcia mają prawo brzmieć lekko egzotycznie. Scharakteryzujemy zatem nieco dokładniej sposób obliczenia $\text{silnia}(1)$, opisując sposób działania programu:

- ♦ Funkcja silnia otrzymuje liczbę 1 jako parametr wywołania i analizuje: „czy 1 równa się 0?”. Odpowiedź brzmi: „Nie”, zatem funkcja przyjmuje, że jej wynikiem jest $1 * \text{silnia}(0)$, czyli po prostu $\text{silnia}(0)$.
- ♦ Niestety wartość $\text{silnia}(0)$ jest nieznaną. Funkcja wywołuje zatem kolejny swój egzemplarz, który zajmie się obliczeniem wartości $\text{silnia}(0)$, wstrzymując jednocześnie obliczanie wyrażenia $1 * \text{silnia}(0)$.
- ♦ Wywołanie funkcji $\text{silnia}(0)$ zwraca już konkretny, liczbowy wynik cząstkowy (1), który może zostać użyty do obliczenia wyrażenia $1 * \text{silnia}(0)$, czyli $\text{silnia}(1)$.

Technicznie przekazywanie parametrów odbywa się za pośrednictwem tzw. stosu, czyli specjalnego miejsca w pamięci operacyjnej, które jest używane do zapamiętywania informacji potrzebnych podczas wykonywania programów i dynamicznego przydzielania pamięci. Programista ma jednak prawo zupełnie się tym nie przejmować. Fakt, iż parametr zostanie zwrócony za pośrednictwem stosu, niewiele się bowiem różni od przewidywania wyniku przez telefon. Końcowy efekt, wyrażony przez stwierdzenie „*Wynik jest gotowy!*”, jest bowiem dokładnie taki sam w każdym przypadku, niezależnie od realizacji.

Gdzież się jednak znajdują wspomniane poziomy rekurencji? Spójrzmy raz jeszcze na rysunek 2.2. Aktualna wartość parametru x badanego przez funkcję silnia jest zaznaczona z lewej strony reprezentującego ją „pudełka”. Ponieważ dany egzemplarz funkcji silnia czasami wywołuje kolejny swój egzemplarz (dla obliczenia wyniku cząstkowego), wypadłoby jakoś je różnicować. Najprostszą metodą jest dokonywanie tego poprzez wartość x , która jest dla nas punktem odniesienia używanym przy określaniu aktualnej głębokości rekurencji.

Niebezpieczeństwa rekurencji

Z użyciem rekurencji czasami związane są pewne niedogodności. Dwa klasyczne niebezpieczeństwa prezentują poniższe przykłady.

Ciąg Fibonacciego

Naszym pierwszym zadaniem jest napisanie programu, który liczyłby elementy tzw. ciągu Fibonacciego. Występuje on bardzo często w przyrodzie i jest definiowany następująco:

$$\begin{aligned}
 \text{fib}(0) &= 0, \\
 \text{fib}(1) &= 1, \\
 \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2), \text{ gdzie } n \geq 2
 \end{aligned}$$

Elementy tego ciągu stanowią liczby naturalne tworzące ciąg o takiej własności, że kolejny wyraz (z wyjątkiem dwóch pierwszych) jest sumą dwóch poprzednich (tj. 1, 1, 2, 3, 5, 8, 13, ...). Nazwa

pochodzi od imienia Leonarda z Pizy zwanego Fibonaccim, który pierwszy opublikował ten ciąg².

Nawiązanie do zjawisk przyrodniczych nie jest przypadkowe. Wyobraźmy sobie, że hodujemy króliki i mamy gwarantowany wzrost populacji według reguł:

- ◆ Zaczynamy od jednej pary.
- ◆ Każda samica królika wydaje na świat potomstwo w miesiąc po kopulacji — jednego samca i jedną samicę.
- ◆ W miesiąc po urodzeniu królik może przystąpić do reprodukcji.

Jak w takiej sytuacji można obliczyć wzrost rozwoju naszej farmy? Przy końcu pierwszego miesiąca możemy się spodziewać pierwszego zapłodnienia w pierwszej parze królików. Pod koniec drugiego miesiąca samica urodzi parę młodych — na farmie będą już dwie pary. W trzecim miesiącu będziemy mieli już trzy pary, gdyż pierwsza samica wyda na świat kolejne potomstwo, a urodzone wcześniej przystąpi do kopulacji... W łatwy sposób można obliczyć, że liczebność w kolejnych miesiącach będzie wynosić 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... A liczby te stanowią elementy ciągu Fibonacciego³!

Zaprezentowany niżej program jest niemal dokładnym przetłumaczeniem powyższego wzoru i nie powinien stanowić dla nikogo niespodzianki:



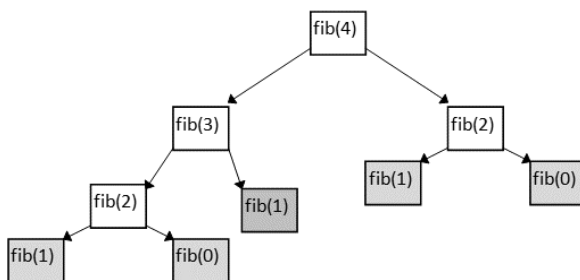
rek3.cpp

```
unsigned long int fib(int x)
{
    if (x<2)
        return x;
    else
        return fib(x-1)+fib(x-2);
}
```

Spróbujmy prześledzić dokładnie wywołania rekurencyjne. Nieskomplikowana analiza prowadzi do następującego drzewa (rysunek 2.3):

Rysunek 2.3.

Obliczanie czwartego
elementu ciągu
Fibonacciego



Każde zacieniowane wyrażenie stanowi problem elementarny; problem o rozmiarze $n \geq 2$ zostaje rozbity na dwa problemy o mniejszym stopniu skomplikowania: $n-1$ i $n-2$, proces dekompozycji zatrzymuje się na przypadkach elementarnych.

² Mniej znane jest to, iż właśnie Fibonacci przywiózł z Azji do Europy wiedzę na temat nowoczesnej arytmetyki, m.in. liczb ujemnych i pozycyjnego systemu zapisu liczb.

³ Gwoli ścisłości podam, że istnieje iteracyjna wersja tego wzoru, wykorzystująca fakt istnienia liczby ϕ (czytaj *fi*). Liczbę tę, zwaną złotym podziałem, otrzymujemy, dzieląc dowolną liczbę ciągu przez liczbę ją poprzedzającą (ϕ wynosi zatem około 1,61804).

Skąd się jednak wziął pesymistyczny tytuł tego podrozdziału? Przypatrzmy się dokładniej rysunkowi 2.3. Już w pierwszej chwili można dostrzec, że znaczna część obliczeń jest wykonywana więcej niż jeden raz (np. cała gałąź zaczynająca się od `fib(2)` jest wręcz zdublowana!). Funkcja `fib` nie ma żadnej możliwości, aby to zauważyć, w końcu jest to tylko program, który wykonuje to, co mu zlecamy. W rozdziale 9. zostanie omówiona ciekawa technika programowania (tzw. *programowanie dynamiczne*) pozwalająca poradzić sobie z powyższą wadą.

Stack overflow!

Tytuł niniejszego podrozdziału oznacza po polsku „przepełnienie stosu”. Jak wykazuje praktyka programowania, pisanie programów podlega regułom raczej świata magii i nieokreśloności niż naszym zachciankom. Ile razy zdarzało się nam zawieszenie się komputera (przez co rozumiemy powszechnie stan, w którym program nie reaguje na nic i trzeba mu zasalutować trzema klawiszami⁴) na skutek działania naszego programu? Zdarza się to nawet najbardziej uważnym programistom i stanowi raczej nieodłączny element pracy programistycznej.

Istnieje kilka typowych przyczyn zawieszania się programów:

- ♦ zachwianie równowagi systemu operacyjnego przez „nielegalne” użycie jego zasobów;
- ♦ „nieskończone” pętle;
- ♦ brak pamięci;
- ♦ nieprawidłowe lub niejasne określenie warunków zakończenia programu;
- ♦ błąd programowania (np. zbyt wolno wykonujący się algorytm).

Programy rekurencyjne są zazwyczaj dość pamięciożerne: z każdym wywołaniem rekurencyjnym wiąże się konieczność zachowania pewnych informacji⁵ niezbędnych do odtworzenia stanu sprzed wywołania, a to zawsze kosztuje trochę cennych bajtów pamięci. Spotyka się programy rekurencyjne, dla których określenie maksymalnego poziomu zagłębienia rekurencji podczas ich wykonywania jest dość łatwe. Analizując program obliczający `3!`, widzimy od razu, że wywoła sam siebie tylko 3 razy; w przypadku funkcji `fib` szybka diagnoza nie przynosi już tak kompletnej informacji.

Przybliżone szacunki nie zawsze należą do najprostszych. Dowodzi tego chyba najlepiej funkcja MacCarthy’ego, zaprezentowana poniżej:



rek4.cpp

```
unsigned long int MacCarthy(int x)
{
    if (x>100)
        return (x-10);
    else
        return MacCarthy(MacCarthy(x+11));
}
```

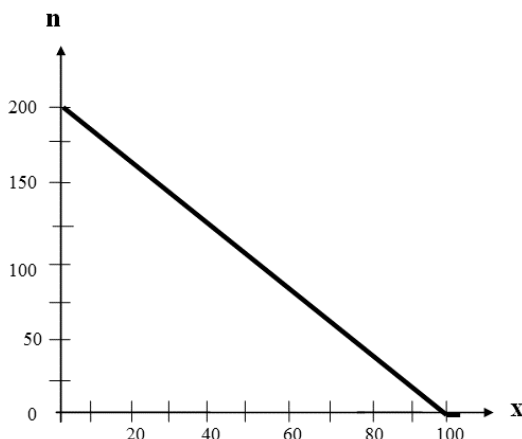
Już na pierwszy rzut oka widać, że funkcja jest jakaś „dziwna”. Kto potrafi powiedzieć w przybliżeniu, jak się przedstawia jej liczba wywołań w zależności od parametru `x` podanego w wywołaniu? Chyba niewielu byłoby w stanie od razu powiedzieć, że zależność ta ma postać przedstawioną na wykresie z rysunku 2.4.

⁴ `Ctrl+Alt+Del` w systemie DOS lub Windows, instrukcja `kill` w systemie Unix, etc.

⁵ W szczegóły wnikać nie będziemy, gdyż tematyka ta nie ma dla nas większego znaczenia w tym miejscu.

Rysunek 2.4.

Liczba wywołań funkcji
MacCarthy'ego w funkcji
parametru wywołania



Bardziej opisowo można zauważyć, że dla wszystkich x większych od 100 funkcja wykona się tylko raz. Uruchom i skompiluj program przykładowy — pozwala on na podawanie parametru wejściowego, jako swój wynik wyświetla wartość funkcji MacCarthy'ego oraz liczbę jej wywołań.

Wyniki działania tej funkcji nie są wcale takie oczywiste, prawda?

Ćwiczenie 2.1.

Zbadaj funkcję MacCarthy'ego w większym przedziale liczbowym niż ten na rysunku. Jakich niebezpieczeństw można się doszukać?

Ćwiczenie 2.2.

Twoim zadaniem będzie narysowanie drzewa wywołań rekurencyjnych funkcji MacCarthy'ego dla granicznej liczby $x = 100$ i dla wartości $x = 99$. Następnie sprawdź wyniki, kompilując i uruchamiając program przykładowy.

Pułapek ciąg dalszy

Jakby nie dość było negatywnych stron programów rekurencyjnych, należy jeszcze dorzucić te, które nie wynikają z samej natury rekurencji, lecz raczej z błędów programisty. Być może warto w tym miejscu podkreślić, iż omawianie „ciemnych stron” rekurencji nie ma na celu zniechęcenia Czytelnika do jej stosowania! Chodzi raczej o wskazanie typowych pułapek i sposobów ich omijania — a te ostatecznie istnieją zawsze (pod warunkiem, że wiemy, CO omijać). Zapraszam zatem do lektury następnych paragrafów.

Stąd do wieczności

W wielu funkcjach rekurencyjnych, pozornie dobrze skonstruowanych, może z łatwością ukryć się błąd polegający na sprowokowaniu nieskończonej liczby wywołań rekurencyjnych. Taki właśnie zwoźniczy przykład jest przedstawiony na następnej stronie (*std.cpp*).

Próba uruchomienia programu *std.cpp* dla wartości $m=2$ doprowadzi, w zależności od systemu operacyjnego, do komunikatu *Stack overflow* lub *Segmentation fault*. W skrócie: zabrakło pamięci i dzieje się tak już po kilku sekundach działania programu!

*std.cpp*

```
int StadDowiecznosc(int n)
{
    if (n==1)
        return 1;
    else
        if ( (n%2)== 0 ) //n parzyste
            return StadDowiecznosc(n-2)*n;
        else
            return StadDowiecznosc(n-1)*n;
}
```

Gdzie jest umiejscowiony problem? Patrząc na ten program, trudno dopatrzeć się szczególnych niebezpieczeństw. W istocie definicja rekurencyjna wydaje się poprawna: mamy przypadek elementarny kończący łańcuch wywołań, problem o rozmiarze n jest upraszczany do problemu o rozmiarze $n-1$ lub $n-2$. Pułapka tkwi właśnie w tej naiwnej wierze, że proces upraszczania doprowadzi do przypadku elementarnego (czyli do $n=1$)! Po dokładniejszej analizie można wszakże zauważyć, że dla $n \geq 2$ wszystkie wywołania rekurencyjne kończą się *parzystą* wartością n . Implikuje to, iż w końcu dojdziemy do przypadku $n=2$, który zostanie zredukowany do $n=0$, który zostanie zredukowany do $n=-2$, który... Można tak kontynuować w nieskończoność, nigdzie po drodze nie ma żadnego przypadku elementarnego!

Wniosek nasuwa się sam: należy zwracać baczną uwagę na to, czy dla wartości parametrów wejściowych należących do dziedziny wartości, które mogą być użyte, rekurencja się kiedyś kończy.

Definicja poprawna, ale...

Rozpatrywany poprzednio przykład służył do zilustrowania problemów związanych ze zbieżnością procesu rekurencyjnego. Wydaje się, że dysponując poprawną definicją rekurencyjną dostarczoną przez matematyka, możemy już być spokojni o to, że analogiczny program rekurencyjny także będzie poprawny (tzn. nie zapętlą się, będzie dostarczać oczekiwane wyniki, etc.). Niestety jest to wiara dość naiwna i niczym nieuzasadniona. Matematyk bowiem jest w stanie zrobić wszystko, co związane z jego dziedziną: określić dziedziny wartości funkcji, udowodnić, że ona się zakończy, wreszcie podać złożoność obliczeniową — jednej jednak rzeczy nie będzie mógł sprawdzić: jak rzeczywisty kompilator wykona tę funkcję! Mimo że większość kompilatorów działa podobnie, to zdarzają się pomiędzy nimi drobne różnice, które powodują, że identyczne programy będą dawać różne wyniki. Nasz kolejny przykład będzie dotyczył właśnie takiego przypadku.

Proszę spojrzeć na następującą funkcję:

```
int N(int n, int p)
{
    if (n==0)
        return 1;
    else
        return N(n-1, N(n-p, p));
}
```

Można przeprowadzić dowód matematyczny⁶, że powyższa definicja jest poprawna w tym sensie, iż dla dowolnych wartości $n \geq 0$ i $p \geq 0$ jej wynik jest określony i wynosi 1. Dowód ten opiera się na założeniu, że wartość argumentu wywołania funkcji jest obliczana tylko wtedy, gdy jest naprawdę niezbędna (co wydaje się dość logiczne). Jak się to zaś ma do typowego kompilatora C++?

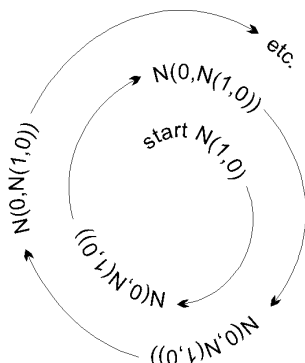
⁶ Patrz [Kro89].

Otóż regułą jest, iż wszystkie parametry funkcji rekurencyjnej są obliczane jako pierwsze, a następnie dokonywane jest wywołanie samej funkcji. (Taki sposób pracy jest zwany *wywołaniem przez wartość*).

Problem może zaistnieć wówczas, gdy w wywołaniu funkcji spróbujemy umieścić ją samą. Zobaczmy, jak to się odbędzie w przypadku naszej funkcji, np. dla $N(1, 0)$ (patrz rysunek 2.5).

Rysunek 2.5.

*Nieskończony
ciąg wywołań
rekurencyjnych*



Zapętlenie jest spowodowane próbą obliczenia parametru p , tymczasem to drugie wywołanie jest w ogóle niepotrzebne do zakończenia funkcji! Istnieje w niej bowiem warunek obejmujący przypadek elementarny: jeśli $n=0$, to zwróć 1. Niestety kompilator o tym nie wie i usiłuje obliczyć ten drugi parametr, powodując zapętlenie programu.

Przykład omówiony w niniejszym paragrafie należy traktować jako swoistą ciekawostkę, niemniej warto go zapamiętać ze względów czysto edukacyjnych.

Typy programów rekurencyjnych

Na podstawie lektury poprzednich paragrafów Czytelnik mógłby wyciągnąć kilka ogólnych wniosków na temat programów używających technik rekurencyjnych: typowo zachłanne w dysponowaniu pamięcią komputera, niekiedy zawieszają system operacyjny... Na szczęście jest to błędne wrażenie! Programy rekurencyjne mają jedną olbrzymią zaletę: są łatwe do zrozumienia i zazwyczaj zajmują mało miejsca, jeśli rozpatrujemy liczbę wierszy kodu użytego do ich realizacji. Z tym ostatnim jest ściśle związana względna łatwość odnajdywania ewentualnych błędów. Wróćmy jednak do tematu.

Zauważyliśmy wspólnie, że program rekurencyjny może być pamięciochłonny i wykonywać się dość wolno. Pytanie brzmi: czy istnieją jakieś techniki programowania pozwalające usunąć (lub co najmniej zredukować) powyższe wady z programu rekurencyjnego? Odpowiedź jest na szczęście pozytywna! Otóż pewna klasa problemów natury rekurencyjnej da się zrealizować na dwa sposoby, dające dokładnie taki sam efekt końcowy, ale różniące się nieco realizacją praktyczną. Podzielmy metody rekurencyjne, tytułem uproszczenia, na dwa podstawowe typy:

- ◆ rekurencja „naturalna”,
- ◆ rekurencja „z parametrem dodatkowym”⁷.

Typ pierwszy mieliśmy okazję zobaczyć podczas analizy dotychczasowych przykładów, teraz zapoznamy się z drugim.

⁷ Pozostaniemy na moment przy tej nieprecyzyjnej nazwie; ten typ rekurencji będzie omawiany jeszcze w rozdziale 6. — w innym jednakże kontekście.

Rozważmy raz jeszcze przykład funkcji obliczającej silnię. Do tej pory znaleźliśmy ją w postaci:

**rek5.cpp**

```
unsigned long int silnia1(unsigned long int x)
{
    if (x==0)
        return 1;
    else
        return x*silnia1(x-1);
}
```

Nie jest to bynajmniej jedyna możliwa realizacja funkcji obliczającej silnię. Spójrzmy dla przykładu na następującą wersję:

```
unsigned long int silnia2(unsigned long int x,
                        unsigned long int tmp=1)
{
    if (x==0)
        return tmp;
    else
        return silnia2(x-1, x*tmp);
}
```

W pierwszym momencie działanie tej drugiej funkcji nie jest być może oczywiste, ale wystarczy wziąć kartkę i ołówek, aby przekonać się na kilku przykładach, że wykonuje ona swoje zadanie tak samo dobrze, jak wersja poprzednia. Osobom nieznającym dobrze C++ należy się niewątpliwie wyjaśnienie konstrukcji funkcji `silnia2`. Otóż dowolna funkcja w C++ może posiadać parametry domyślne.

Dzięki temu funkcja o przykładowym nagłówku:

```
int FunDom(int a, int k=1)
```

może zostać wywołana na dwa sposoby:

- ♦ Poprzez określenie wartości drugiego parametru, np. `FunDom(12,5)`: w tym przypadku `k` przyjmuje wartość 5.
- ♦ Bez określania wartości drugiego parametru, np. `FunDom(12)`: `k` przyjmuje wtedy wartość domyślną równą tej podanej w nagłówku, czyli 1.

Ta użyteczna cecha języka C++ wykorzystana została w drugiej wersji funkcji do obliczania silni. Jakie jednak istotne względy przemawiają za używaniem tej osobliwej z pozoru metody programowania? Argumentem nie jest tu wzrost czytelności programu, bowiem już na pierwszy rzut oka `silnia2` jest o wiele bardziej zagmatwana niż `silnia1`!

Istotna zaleta rekurencji „z parametrem dodatkowym” jest ukryta w sposobie wykonywania programu. Wyobraźmy sobie, że program rekurencyjny „bez parametru dodatkowego” wywołał sam siebie 10-krotnie, aby obliczyć dany wynik. Oznacza to, że wynik cząstkowy z dziesiątego, najgłębszego poziomu rekurencji będzie musiał być przekazany przez kolejne dziesięć poziomów do góry, do swojego pierwszego egzemplarza.

Jednocześnie z każdym „zamrożonym” poziomem, który czeka na nadejście wyniku cząstkowego, wiąże się pewna ilość pamięci, która służy do odtworzenia m.in. wartości zmiennych tego poziomu (tzw. kontekst). Co więcej, odtwarzanie kontekstu już samo w sobie zajmuje cenny czas procesora, który mógłby być wykorzystany np. na inne obliczenia.

Czytelnik już zapewne się domyśla, że program rekurencyjny „z parametrem dodatkowym” robi to wszystko nieco wydajniej. Ponieważ parametr dodatkowy służy do przekazywania elementów wyniku końcowego, dysponując nim, program nie ma potrzeby przekazywania wyniku obliczeń do góry, piętro po piętrze. Po prostu w momencie, w którym program „stwierdzi”, że

obliczenia zostały zakończone, procedura wywołująca zostanie o tym poinformowana wprost z ostatniego aktywnego poziomu rekurencji. Co za tym wszystkim idzie, nie ma absolutnie żadnej potrzeby zachowywania kontekstu poszczególnych poziomów pośrednich, liczy się tylko ostatni aktywny poziom, który dostarczy wynik, i basta!

Myślenie rekurencyjne

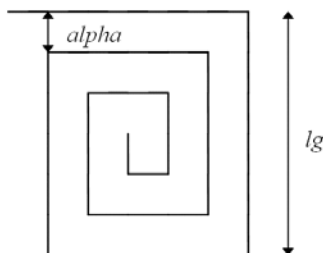
Pomimo oczywistych przykładów na to, że rekurencja jest dla człowieka czymś jak najbardziej naturalnym, niektórzy mają pewne trudności z używaniem jej podczas programowania. Nieumiejętność wyczucia istoty tej techniki programowania może wynikać z braku dobrych i poglądowych przykładów na jej wykorzystanie. Idąc za tym stwierdzeniem, postanowiłem wybrać kilka prostych programów rekurencyjnych, generujących znane motywy graficzne — ich dobre zrozumienie będzie wystarczającym testem na oszacowanie swoich zdolności myślenia rekurencyjnego (ale nawet wówczas wykonanie zadań zamieszczonych pod koniec rozdziału będzie jak najbardziej wskazane).

Przykład 1.: Spirala

Zastanówmy się, jak można narysować spiralę rekurencyjnie jednym pociągnięciem kreski — rysunek 2.6.

Rysunek 2.6.

Spirala narysowana rekurencyjnie



Parametrami programu są :

- ♦ odstęp pomiędzy liniami równoległymi: α ;
- ♦ długość boku rysowanego w pierwszej kolejności: lg .

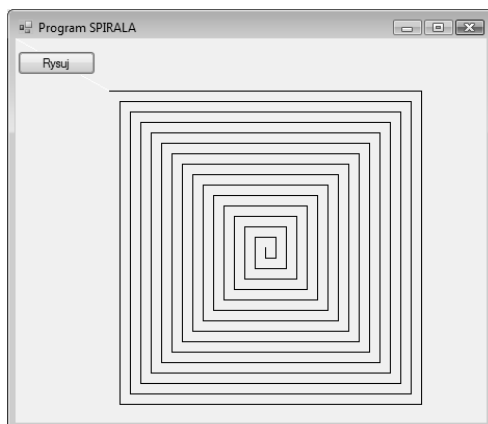
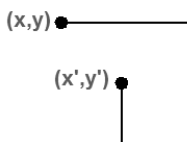
Algorytm iteracyjny byłby również nieskomplikowany (zwykła pętla), ale założmy, że zapomniemy chwilowo o jego istnieniu i wykonamy to samo rekurencyjnie. Istota rekurencji polega głównie na znalezieniu właściwej dekompozycji problemu. Tutaj jest ona przedstawiona na rysunku i w związku z tym ewentualne przetłumaczenie jej na program w C++ powinno być znacznie ułatwione.

Rekurencyjność naszego zadania jest oczywista, bowiem program wynikowy zajmuje się powtarzaniem głównie tych samych czynności (rysuje linie poziome i pionowe, jednakże o różnej długości). Naszym zadaniem będzie odszukanie schematu rekurencyjnego i warunków zakończenia procesu wywołań rekurencyjnych.

Jak rozwiązać to zadanie? Wpierw przybliżmy się nieco do „rzeczywistości ekranowej” i wybierzmy jako punkt startowy pewną parę (x, y) . Idea rozwiązania polega na narysowaniu 4 odcinków zewnętrznych spirali i dotarciu do punktu (x', y') . W tym nowym punkcie startowym możemy już wywołać rekurencyjnie procedurę rysowania, obciążoną oczywiście pewnymi warunkami gwarantującymi jej poprawne zakończenie.

Elementarny przypadek rozwiązania prezentuje rysunek 2.7.

Rysunek 2.7.
*Spirala narysowana
 rekurencyjnie*
 — szkic rozwiązania
 + wynik



Jedna z kilku możliwych wersji programu, który realizuje to, co zostało wyżej opisane, jest przedstawiona poniżej. Program używa następujących, umownych instrukcji graficznych⁸:

- ♦ `lineto(x,y)` — kreśli odcinek prostej od pozycji bieżącej do punktu (x, y)
- ♦ `moveto(x,y)` — przesuwa kursor graficzny do punktu (x, y)
- ♦ `getmaxx()` — zwraca maksymalną współrzędną poziomą (zależy od rozdzielczości trybu graficznego)
- ♦ `getmaxy()` — zwraca maksymalną współrzędną pionową (j. w.)
- ♦ `getx()` — zwraca aktualną współrzędną poziomą
- ♦ `gety()` — zwraca aktualną współrzędną pionową



spirala.cpp

```
const double alpha=10;
void spirala(double lg,double x,double y)
{
    if (lg>0)
    {
        lineto(x+lg, y);
        lineto(x+lg, y+lg);
        lineto(x+alpha, y+lg);
        lineto(x+alpha, y+alpha);
        spirala(lg-2*alpha, x+alpha, y+alpha);
    }
}

int main()
{
    // tu zainicjuj tryb graficzny
    moveto(90,50);
    spirala(getmaxx()/2,getx(),gety());
    getch(); // poczekaj na naciśnięcie klawisza
            // tu zamknij tryb graficzny
}
```

⁸ Są one zgodne z bibliotekami kompilatora Borland dla systemu DOS (pliki znajdziesz w folderze INNE w archiwum ZIP na ftp).



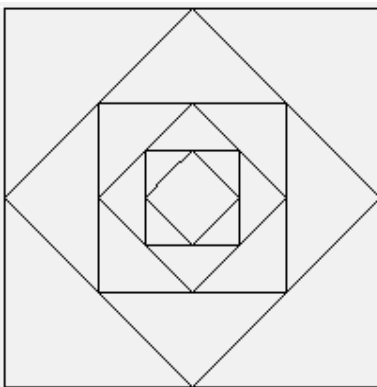
Program graficzny w wersji Microsoft Windows został przygotowany w kompilatorze Microsoft Visual C++ Express Edition. Pełny projekt przygotowany dla środowiska graficznego Windows znajduje się w katalogu VISUAL/spirala. Ponieważ kod rozwiązania dla Windows jest niezbyt czytelny (zawiera wiele komend specyficznych dla programowania w tym systemie), zdecydowałem się pominąć pełny listing w treści książki, pozostawiając w niej oryginalny, zwięzły kod użyty w poprzednich jej wydaniach. Kod dla Windows znajdziesz w plikach: *spirala_win.cpp* i *spirala_win.h*.

Przykład 2.: Kwadraty „parzyste”

Zadanie jest podobne do poprzedniego: jak jednym pociągnięciem kreski narysować figurę przedstawioną na rysunku 2.8?

Rysunek 2.8.

Kwadraty narysowane rekurencyjnie ($n = 3$)



Przypadkiem elementarnym będzie tutaj narysowanie jednej pary kwadratów (wewnętrzny obrócony w stosunku do zewnętrznego).

To zadanie jest nawet prostsze niż poprzednie, sztuka polega jedynie na wyborze właściwego miejsca wywołania rekurencyjnego⁹:



kwadraty.cpp

```
void kwadraty(int n, double lg, double x, double y)
{
    // n — parzysta liczba kwadratów
    // x, y — współrzędne punktu startowego
    if (n > 0)
    {
        lineto(x+lg, y);
        lineto(x+lg, y+lg);
        lineto(x, y+lg);
        lineto(x, y+lg/2);
        lineto(x+lg/2, y+lg);
        lineto(x+lg, y+lg/2);
        lineto(x+lg/2, y);
        lineto(x+lg/4, y+lg/4);
        kwadraty(n-1, lg/2, x+lg/4, y+lg/4);
        lineto(x, y+lg/2);
    }
}
```

⁹ Ten kod został zachowany z przyczyn historycznych, jest on zgodny z kompilatorem Borland i systemem DOS, znajdziesz go w folderze INNE w archiwum ZIP.


```
        lineto(x,y):
        }
    }

    int main()
    {
        // inicjuj tryb graficzny
        moveto(90,50):
        kwadraty(5, getmaxx()/2, getx(), gety());
        getch():
        // zamknij tryb graficzny
    }
```



Uwaga

Program graficzny w wersji Microsoft Windows został przygotowany w kompilatorze Microsoft Visual C++ Express Edition, projekt znajduje się w katalogu VISUAL/kwadraty. Podobnie jak w poprzednim przykładzie, także i tu zdecydowałem się pominąć pełny listing w treści książki, pozostawiając w niej oryginalny, zwięzły kod użyty w poprzednich jej wydaniach. Kod dla Windows znajdziesz w plikach: *kwadraty_win.cpp* i *kwadraty_win.h*.

Uwagi praktyczne na temat technik rekurencyjnych

Szczegółowy wgląd w techniki rekurencyjne uświadomił nam, że niosą one ze sobą zarówno plusy, jak i minusy. Zasadniczą zaletą jest czytelność i naturalność zapisu algorytmów w formie rekursywnej — szczególnie gdy zarówno problem, jak i struktury danych z nim związane są wyrażone w postaci rekurencyjnej. Procedury rekurencyjne są zazwyczaj klarowne i krótkie, dzięki czemu dość łatwo jest wykryć w nich ewentualne błędy. Jak jednak ocenić algorytm rekurencyjny? Otóż nawet bez używania skomplikowanego aparatu matematycznego dość bezpieczną techniką oceny jakości programu rekurencyjnego jest ocena, czy warunek zakończenia jest poprawny (przypadek elementarny) i czy dekompozycja rekurencyjna prowadzi do *zmniejszenia* rozmiaru problemu. Wszelkie dziwaczne konstrukcje (patrz funkcja MacCarthy’ego) stanowią zazwyczaj bardziej ciekawostki akademickie niż przykłady poprawnych procedur.

Dużą wadą wielu algorytmów rekurencyjnych jest pamięciożerność: wielokrotne wywołania rekurencyjne mogą łatwo zablokować całą dostępną pamięć! Problemem jest tu jednak nie fakt zajętości pamięci, ale typowa niemożność łatwego jej oszacowania przez konkretny algorytm rekurencyjny. Można do tego wykorzystać metody służące do analizy efektywności algorytmów (patrz rozdział 3.), jednakże jest to dość nużące obliczeniowo, a czasami nawet po prostu niemożliwe.

W podrozdziale „Typy programów rekurencyjnych” poznaliśmy metodę na ominięcie kłopotów z pamięcią poprzez stosowanie rekurencji „z parametrem dodatkowym”. Nie wszystkie jednak problemy dadzą się rozwiązać w ten sposób, ponadto programy używające tej metody tracą odrobinę na czytelności. No cóż, nie ma róży bez kolców...

Kiedy nie należy używać rekurencji? Ostateczna decyzja należy zawsze do programisty, tym niemniej istnieją sytuacje, gdy ów dylemat jest dość łatwy do rozstrzygnięcia. Nie powinniśmy używać rozwiązań rekurencyjnych, gdy:

- ♦ W miejsce algorytmu rekurencyjnego można podać czytelny lub szybki odpowiednik iteracyjny.
- ♦ Algorytm rekurencyjny jest *niestabilny* (np. dla pewnych wartości danych wejściowych może się zapęlić lub dawać dziwne wyniki — często wynika to ze specyfiki kompilatora lub cech platformy sprzętowej).

Ostatnią uwagę podaję już raczej, by dopełnić formalności. Otóż w literaturze można czasem napotkać rozważania na temat niekorzystnych cech tzw. *rekurencji skrośnej*: podprogram A wywołuje podprogram B , który wywołuje z kolei podprogram A . Nie podałem celowo przykładu takiego „dziwoląga”, gdyż nadmiar złych przykładów może być szkodliwy. Praktyczny wniosek, który możemy wysnuć, analizując osobliwe programy rekurencyjne, pełne nieprawdopodobnych konstrukcji, jest jeden: UNIKAJMY ICH, jeśli tylko nie jesteśmy całkowicie pewni poprawności programu, a intuicja nam podpowiada, że w danej procedurze jest coś, co może spowodować kłopoty.

Korzystając z katalogów algorytmów, formalizując programowanie, etc., można bardzo łatwo zapomnieć, że wiele pięknych i eleganckich metod powstało samo z siebie — jako przeblisk geniuszu, intuicji, sztuki... A może i my moglibyśmy dołożyć nasze „co nieco” do tej kolekcji? Proponuję ocenić własne siły poprzez rozwiązywanie zadań, które odpowiedzą w sposób najbardziej obiektywny, czy rozumiemy rekurencję jako metodę programowania.

Zadania

Wybór reprezentatywnego dla rekurencji zestawu zadań wcale nie był łatwy dla autora tej książki — dziedzina ta jest bardzo rozległa i w zasadzie wszystko w niej jest w jakiś sposób interesujące. Ostatecznie, co zwykłem podkreślać, zadecydowały względy praktyczne i prostota.

Zadanie 1.

Załóżmy, że chcemy odwrócić w sposób rekurencyjny tablicę liczb całkowitych. Proszę zaproponować algorytm z użyciem rekurencji „naturalnej”, który wykona to zadanie.

Zadanie 2.

Powróćmy do problemu poszukiwania pewnej zadanej liczby x w tablicy, tym razem jednak posortowanej od wartości minimalnych do maksymalnych. Metoda poszukiwania, bardzo znana i efektywna (tzw. *przeszukiwanie binarne*), polega na następującej obserwacji:

- ◆ Podzielmy tablicę o rozmiarze n na połowę:
 - ◆ $t[0]$, $t[1]$... $t[n/2-1]$, $t[n/2]$, $t[n/2+1]$, ..., $t[n-1]$.
 - ◆ Jeśli $x = t[n/2]$, to element x został znaleziony¹⁰.
 - ◆ Jeśli $x < t[n/2]$, to element x być może znajduje się w lewej połowie tablicy; analizuj ją.
 - ◆ Jeśli $x > t[n/2]$, to element x być może znajduje się w prawej połowie tablicy; analizuj ją.

Wyrażenie *być może* daje nam furtkę bezpieczeństwa w przypadku niepowodzenia poszukiwania. Zadanie polega na napisaniu dwóch wersji funkcji realizującej powyższy algorytm, jednej używającej rekurencji naturalnej i drugiej — dla odmiany — nierekurencyjnej.

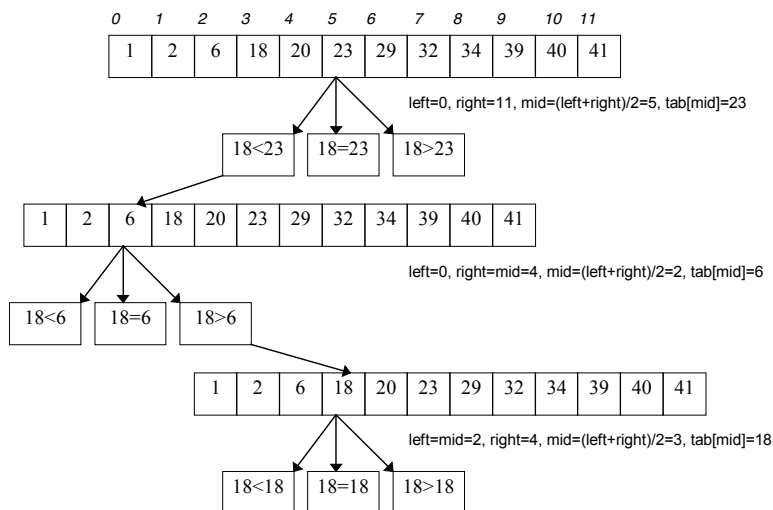
Rysunek 2.9 prezentuje działanie algorytmu dla następujących danych:

- ◆ 12-elementowa tablica zawiera liczby: 1, 2, 6, 18, 20, 23, 29, 32, 34, 39, 40, 41.
- ◆ Szukamy liczby 18.

W celu dokładniejszego przeanalizowania algorytmu posłużymy się kilkoma zmiennymi pomocniczymi:

- ◆ `left` — indeks tablicy ograniczający obserwowany obszar tablicy od lewej strony;
- ◆ `right` — indeks tablicy ograniczający obserwowany obszar tablicy od prawej strony;
- ◆ `mid` — indeks elementu środkowego obserwowanego aktualnie obszaru tablicy.

¹⁰ W C++ dzielenie całkowite obcina wynik do liczby całkowitej (odpowiednik *div* w Pascalu).

Rysunek 2.9.Przeszukiwanie binarne
na przykładzie

Na rysunku 2.9 przedstawione jest działanie algorytmu oraz wartości zmiennych `left`, `right` i `mid` podczas każdego ważniejszego etapu. Poszukiwanie zakończyło się pomyślnie już po trzech etapach¹¹. Warto zauważyć, że to samo zadanie, rozwiązywane za pomocą przeglądania od lewej do prawej elementów tablicy, zostałoby ukończone dopiero po 4 etapach. Być może otrzymany zysk nie oszałamia, proszę sobie jednak wyobrazić, co by było, gdyby tablica miała rozmiar kilkanaście razy większy niż ten użyty w przykładzie?! Proszę napisać funkcję, która realizuje poszukiwanie binarne w sposób rekurencyjny.

Zadanie 3.

Napisz funkcję, która — otrzymując liczbę całkowitą dodatnią — wypisze jej *reprezentację dwójkową*. Należy wykorzystać znany algorytm dzielenia przez podstawę systemu.

Przykładowo zamieńmy liczbę 13 na jej postać binarną:

$$\begin{array}{rclclcl}
 13 & : & 2 & = & 6 & + & 1 \\
 6 & : & 2 & = & 3 & + & 0 \\
 3 & : & 2 & = & 1 & + & 1 \\
 1 & : & 2 & = & 0 & + & 1 \\
 & & & & \uparrow & & \\
 & & & & \text{Koniec algorytmu!} & &
 \end{array}$$

Problem polega na tym, że otrzymaliśmy prawidłowy wynik, ale... od tyłu! Algorytm dał nam 1011, natomiast prawidłową postacią jest 1101. Dopiero w tym miejscu zaczyna się właściwe zadanie:

Pytanie 1

Jak wykorzystać rekurencję do odwrócenia kolejności wypisywanych cyfr?

Pytanie 2

Czy istnieje nieskomplikowane i eleganckie rozwiązanie tego zadania, wykorzystujące rekurencję z „parametrem dodatkowym”?

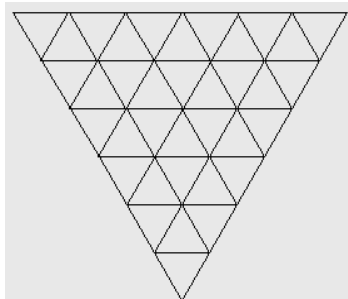
¹¹ Za „etap” będziemy tu uważali moment testowania, czy dana jest tą liczbą, której poszukujemy.

Zadanie 4.

Spróbuj napisać funkcję, która rekurencyjnie wyrysuje „dywanik” przedstawiony na rysunku 2.10:

Rysunek 2.10.

Trójkąty narysowane rekurencyjnie



Zadanie 5.

W ramach relaksu zaprogramuj rekurencyjnie algorytm Euklidesa — algorytm poszukiwania największego wspólnego dzielnika dwóch liczb (NWD lub GCD po angielsku — *greatest common divisor*). Największy wspólny dzielnik, NWD, dla dwóch liczb naturalnych a , b : taka największa liczba naturalna c , że a oraz b dzieli się bez reszty przez c .

Euklides zauważył, że jeśli od większej liczby odejmiemy mniejszą, to ta mniejsza liczba i otrzymana różnica będą miały taki sam największy wspólny dzielnik, jak pierwotne liczby. Gdy przy kolejnym odejmowaniu otrzymamy parę takich samych liczb, to znaleźliśmy NWD.

Zadanie 6.

Napisz programu obliczający silnię bez użycia rekurencji.

Rozwiązania i wskazówki do zadań

Zadanie 1.

Idea rozwiązania jest następująca:

- ◆ Zamieńmy miejscami elementy skrajne tablicy (*przypadek elementarny*).
- ◆ Odwróćmy pozostałą część tablicy (*wywołanie rekurencyjne*).

Odpowiadający temu rozumowaniu program przedstawia się następująco:



rev_tab.cpp

```
//zamiana zmiennych:
void swap(int& a, int& b)
{
    int temp=a;
    a=b;
    b=temp;
}

void odwroc(int *tab, int left, int right)
{

```

```

if(left<right)
{
    swap(tab[left],tab[right]); //zamieniamy elementy skrajne
    odwroc(tab,left+1,right-1); //odwracamy resztę
}
}
int main()
{
    int tab1[8]={1,2,3,4,5,6,7,8};
    for(int i=0;i<8;i++)
        cout << tab1[i] << " "; cout << endl;
    odwroc(tab1,0,7); //przykładowe wywołanie i weryfikacja:
    for(int i=0;i<8;i++)
        cout << tab1[i] << " "; cout << endl;
}

```

Zadanie 2.

Poniżej przedstawiona jest wyłącznie wersja rekurencyjna programu. Jestem przekonany, że Czytelnik odkryje bez trudu analogiczne rozwiązanie iteracyjne¹²:



binary_s.cpp

```

int szukaj_rec(int* tab, int x, int left, int right)
{
    if(left>right)
        return -1; // element niezalezony
    else
    {
        int mid=(left+right)/2;
        if(tab[mid]==x)
            return mid; // element zostal znaleziony!
        else
        {
            if(x<tab[mid])
                return szukaj_rec(tab,x,left,mid-1);
            Else
                return szukaj_rec(tab,x,mid+1,right);
        }
    }
}

```

Zadanie 3.

Program nie należy do zbyt skomplikowanych, choć wcale nie jest trywialny. Zastanówmy się, jak zmusić algorytm do przedstawienia wyniku w postaci normalnej, tzn. od lewej do prawej. W tym celu przeanalizujemy raz jeszcze działanie algorytmu bazującego na dzieleniu przez podstawę systemu liczbowego (tutaj 2). Liczba x jest dzielona przez dwa, co daje nam liczbę $[x \div 2]$ plus reszta. Owa reszta to oczywiście $[x \bmod 2]$ i jest to jednocześnie *ostatnia* cyfra reprezentacji binarnej, którą chcemy otrzymać.

Czy jest jakiś sposób, aby odwrócić kolejność wyprowadzania cyfr dwójkowych, korzystając ciągle z tego prostego algorytmu? Otóż tak, pod warunkiem że spojrzymy nań nieco inaczej. Popatrzmy, jak symbolicznie można rozisać tworzenie reprezentacji dwójkowej pewnej liczby x , używając już właściwych dla C++ operatorów:

$$[x]_2 = [x \% 2] * \left[\frac{x}{2} \right]_2$$

¹² Lub zajrzy do rozdziału 7.

Zapis ten sugeruje już, jak można rekurencyjnie przedstawić ten algorytm:

$$[x]_2 = \text{przypadek elementarny} * \text{wywołanie rekurencyjne}$$

Jeśli w powyższym algorytmie każemy komputerowi najpierw wypisać liczbę $[x/2]$ dwójkowo, a dopiero potem $[x \% 2]$ (które to wyrażenie przybiera dwie wartości: 0 lub 1), to wynik pojawi się na ekranie w postaci normalnej, a nie odwrócony, jak poprzednio.

Warto zapamiętać tę sztuczkę, może być ona pomocna w wielu innych programach.



post_2.cpp

```
void post_dw(unsigned long int n)
{
    if(n!=0)
    {
        post_dw(n/2); // n modulo 2
        cout << n % 2; // reszta z dzielenia przez 2
    }
}
```

Co zaś się tyczy pytania drugiego, to z mojej strony mogę dać na nie odpowiedź: być może. Rozwiązałem ten problem z użyciem rekurencji „z parametrem dodatkowym”, ale nie udało mi się znaleźć rozwiązania na tyle eleganckiego, aby było warte prezentacji jako odpowiedź. Być może któryś z Czytelników znajdzie więcej czasu i dokona tego wyczynu? Gorąco zachęcam do prób — być może do niczego nie doprowadzą, ale na pewno nauczą więcej niż lektura gotowych rozwiązań.

Zadanie 4.

Oto jedno z możliwych rozwiązań:



trojkaty.cpp

```
void trojkaty(double n, double lg, double x, double y)
{
    // n = liczba podziałów
    if (n>0)
    {
        double a=lg/n;
        double h=a*sqrt(3)/2.0;
        lineto(x-a/2.0,y-h);
        trojkaty(n-1,lg-a,x-a/2.0,y-h);
        lineto(x+a/2.0,y-h);
        for(double i=1;i<n;i++)
        {
            lineto(x+(i-1)*a/2.0,y-(i+1)*h);
            lineto(x+(i+1)*a/2.0,y-(i+1)*h);
        }
        lineto( x,y);
    }
}

int main()
{
    // inicjuj tryb graficzny
    moveto(getmaxx()/2,getmaxy());
    trojkaty(6,getmaxx()/2,getx(),gety());
    getch();
    // zamknij tryb graficzny
}
```

Zadanie 5.

Oto jedno z możliwych rozwiązań z przykładem użycia (w pliku na *fip* znajdziesz także wariant bardziej klasyczny):



nwd.cpp

```
int nwd(int a, int b)
{
    if (b==0)
        return a;
    else
        nwd2(b, a % b); // modulo
}
int main()
{
    cout << " nwd(12,3) =" << nwd(12,3) << endl;
    cout << " nwd(24,30) =" << nwd(24,30) << endl;
    cout << " nwd(5,7) =" << nwd(5,7) << endl;
    cout << " nwd(54,69) =" << nwd(54,69) << endl;
}
```

Zadanie 6.

Funkcja silnia jest tak naprawdę bardzo prosta w realizacji, gdy zauważymy, że jest to seria sekwencyjnych operacji mnożenia (czytaj od prawej do lewej strony):

$$\begin{array}{ccccccc}
 & & & & & 0! = 1 & \\
 & & & & & \downarrow & \\
 & & & & 1! = 1 * 0! & & \\
 & & & \downarrow & & & \\
 & & 2! = 2 * 1! & & & & \\
 & \downarrow & & & & & \\
 3! = 3 * 2! & & & & & & \\
 \downarrow & & & & & & \\
 \dots & & & & & &
 \end{array}$$

Jak łatwo zauważyć, aby obliczyć kolejną wartość silni, wystarczy dysponować skumulowanym wynikiem poprzednio wykonanych operacji mnożenia, co można zrealizować przy pomocy jednej pętli w C++. Rozwiązanie znajdziesz w programach dołączonych do książki, ale postaraj się do niego dojść samodzielnie, bez zaglądania do gotowego kodu.

Rozdział 3.

Analiza złożoności algorytmów

Podstawowe kryteria pozwalające na wybór właściwego algorytmu zależą głównie od kontekstu, w jakim zamierzamy go używać. Jeśli chodzi nam o sporadyczne korzystanie z programu do celów „domowych” czy też po prostu prezentacji wykładowej, współczynnikiem najczęściej decydującym bywa *prostota algorytmu*.

Nieco inna sytuacja powstaje w momencie zamierzonej komercjalizacji programu, ewentualnie udostępnienia go szerszej grupie osób.

Klient, który otrzymuje płytę z programem w postaci binarnej (gotowym do instalacji i użytkowania), jest w nikłym stopniu (jeśli w ogóle) zainteresowany estetyką wewnętrzną programu, klawiaturnością i pięknem użytych algorytmów, etc. Użytkownik ten — zwany czasem *końcowym* — będzie się koncentrował na tym, co jest dla niego bezpośrednio dostępne: rozbudowanych systemach menu, pomocy kontekstowej, jakości prezentacji wyników w postaci graficznej itp. Taki punkt widzenia jest często spotykany i programista, który zapomni go uwzględnić, ryzykuje wyeliminowaniem się z rynku programów komercyjnych.

Konflikt interesów, z którym mamy tu do czynienia, jest zresztą typowy dla wszelkich relacji typu producent — klient: pierwszy jest głęboko zainteresowany, aby stworzyć produkt jak najtaniej i sprzedać jak najdrożej, aby stworzyć program jak *najtaniej* i sprzedać jak *najdrożej*, natomiast drugi chciałby za niewielką sumę nabyć produkt najwyższej jakości. Upraszczając dla potrzeb naszej dyskusji wyżej zaanonsowaną problematykę, możemy wyróżnić podstawowe kryteria oceny programu. Są to:

- ◆ sposób komunikacji z użytkownikiem,
- ◆ szybkość wykonywania podstawowych funkcji programu,
- ◆ awaryjność.

W rozdziale tym zajmiemy się wyłącznie aspektem sprawnościowym wykonywania programów, problem komunikacji — jako zbyt obszerny — zostawiając może na inną okazję, a o zagadnieniach awaryjności oprogramowania już w ogóle nie będę wspominał!

Tematyką tego rozdziału jest tzw. złożoność obliczeniowa algorytmów, czyli próba odpowiedzi na pytanie: *który z dwóch programów wykonujących to samo zadanie (ale odmiennymi metodami) jest efektywniejszy?* Wbrew pozorom w wielu przypadkach odpowiedź wcale nie jest taka prosta i wymaga użycia dość złożonego aparatu matematycznego. Nie będzie jednak wymagane od Czytelnika posiadanie jakichś szczególnych kwalifikacji matematycznych — prezentowane metody będą w dużym stopniu uproszczone i nastawione raczej na zastosowania praktyczne niż teoretyczne studia.

Istotna uwaga należy się osobom, które byłyby głębiej zainteresowane stroną matematyczną prezentowanych zagadnień, dowodami użytych metod, etc. Ponieważ głównym kryterium doboru zaprezentowanych narzędzi matematycznych była ich prostota, a nie kompletność i zgodność z wszelkimi formalizmami, w celu pogłębienia wiedzy odsyłam do podręczników analizy matematycznej. W końcu nie każdy programista musi być matematykiem.

Tych Czytelników, którym brakuje nieco formalizmu matematycznego, można odesłać do dokładniejszej lektury, np. [BB87], [Gri84], [Kro89] czy też klasycznych tytułów: [Knu73], [Knu69], [Knu75].

Pomocne będą także zwykle podręczniki matematyczne, ale należy zdawać sobie sprawę z tego, iż częstokroć zawierają one nadmiar informacji i wyłuskanie tego, co jest nam niezbędne, jest znacznie trudniejsze niż w przypadku tytułów z założenia przeznaczonych dla programistów.

Definicje i przykłady

Zanurzając się w problematykę analizy sprawnościowej programów, możemy wyróżnić minimum dwa ważne czynniki wpływające na dobre samopoczucie użytkownika programu:

- ♦ czas wykonania (*znowu się zawiesił, czy też coś liczy?!*);
- ♦ zajętość pamięci (mam już dość komunikatów typu: *Insufficient memory* — *save your work!*).

Z uwagi na znaczne potanieńnię pamięci RAM w ostatnich latach to drugie kryterium straciło już praktycznie na znaczeniu². Co innego jest z pierwszym! Wcale nie jest aż tak dobrze z szybkością współczesnych komputerów³, aby przestać się tym zupełnie przejmować. Bo coś z tego, że komputer X jest 12 razy szybszy od Y , jeśli dla algorytmu A i problemu P oznacza to przyspieszenie czasu zakończenia obliczeń z... 12 lat do „zaledwie” jednego roku?! Abstrahuję tu od tego, że nikt by algorytmu A do tego zadania nie użył. Dla tego samego problemu znaleziono inny algorytm, który zrobił to samo w przeciągu kilku godzin.

Jednym ze szczególnie istotnych problemów w dziedzinie analizy algorytmów jest dobór właściwej *miary złożoności obliczeniowej*. Musi być to miara na tyle reprezentatywna, aby użytkownicy np. małego komputera osobistego i potężnej stacji roboczej — obaj używający tego samego algorytmu — mogli się ze sobą porozumieć co do jego sprawności obliczeniowej. Jeśli ktoś stwierdzi, że *jego program jest szybki, bo wykonał się w 1 minutę*, to nie dostaniemy w ten sposób żadnej reprezentatywnej informacji. Musi on jeszcze odpowiedzieć na dodatkowe pytania, na przykład:

- ♦ Jakiego komputera użył?
- ♦ Jaka była liczba przetwarzanych informacji?
- ♦ Jaka jest częstotliwość pracy zegara taktującego procesor?
- ♦ Czy program był jedynym wykonującym się wówczas w pamięci? Jeśli nie, to jaki miał priorytet?

¹ *Brak pamięci* — *zachowaj swoje dane*: niegdyś częsty w wielu profesjonalnych programach pisanych dla systemu Windows. Obecnie z uwagi na duże dyski twarde symulujące pamięć dynamiczną występuje on rzadziej, ale czy nie wynika to bardziej z postępu technologii niż ze wzrostu jakości oprogramowania?

² Stwierdzenie to jest fałszywe w odniesieniu do niektórych dziedzin techniki; niektóre algorytmy używane w syntezie obrazu pochłaniają tyle pamięci, że w praktyce są ciągle nieużywane w komputerach osobistych. Ponadto należy sobie zdać sprawę, że obsługa skomplikowanych struktur danych jest na ogół dość czasochłonna — jedno kryterium oddziałuje zatem na drugie!

³ Oczywiście mam na myśli komputery osobiste.

- ♦ Jakiego kompilatora użyto podczas pisania tego programu?
- ♦ Jeśli to był kompilator XYZ, to czy zostały włączone opcje optymalizacji kodu?

Od razu jednak widać, że daleko w ten sposób nie zajdziemy. Potrzebna jest nam *miara uniwersalna*, niemająca nic wspólnego ze szczegółami natury, nazwijmy to, sprzętowej.



Uwaga

W świecie „fizycznym”, gdzie mamy do czynienia z pamięciami dyskowymi, sieciami i opóźnieniami transmisji, często operuje się pojęciem *wydajności oprogramowania*. Odbiorca oprogramowania definiuje wobec dostawcy swoje wymagania wydajnościowe (np. czasy przetwarzania raportów miesięcznych), a dostawca próbuje je zrealizować, biorąc pod uwagę wszelkie uwarunkowania technologiczne. Wydajność jest jednak dalece mylącym pojęciem, bowiem tak naprawdę niewiele mówi o zaaplikowanych algorytmach. Znacznie bardziej interesujące byłoby porównanie (nawiązując do podanego wyżej przykładu) czasu wykonania raportu miesięcznego w przypadku podwojenia liczby danych wejściowych (np. rekordów z bazy danych).

Parametrem najczęściej decydującym o czasie wykonania określonego algorytmu jest rozmiar danych n , z którymi ma on do czynienia⁴. Pojęcie *rozmiaru danych* jest wieloznaczne: dla funkcji sortującej tablicę będzie to po prostu rozmiar tablicy, natomiast dla programu obliczającego wartość funkcji *silnia* — wielkość danej wejściowej.

Podobnie funkcja wyszukująca dane w liście (patrz rozdział 5.) będzie bardzo „uczulona” na jej długość. Wszystkie te przypadki określa się właśnie jako rozmiar danych wejściowych. Ponieważ odczytanie właściwego znaczenia tego terminu jest intuicyjnie bardzo proste, dalej będziemy używać właśnie tego nieprecyzyjnego określenia w miejsce rozwlekłych wyjaśnień cytowanych powyżej.



Definicja

Uogólniając, można powiedzieć, że n jest to najbardziej znaczący parametr algorytmu, wpływający na czas jego wykonania.

Powróćmy jeszcze do przykładu przytoczonego na samym początku tego rozdziału. Nieprzygotowany Czytelnik, widząc stwierdzenia: „czas wykonania programu to 12 lat”, ma prawo się nieco obruszyć — czy to jest w ogóle możliwe?! W istocie w miarę rozwoju techniki mamy do czynienia z coraz szybszymi komputerami i być może kiedyś 12 lat mogło być nawet prawdą, ale obecnie?

Niestety trzeba podkreślić, że podany czas wcale nie jest tak przerażająco długi. Proszę spojrzeć na tabelę 3.1.

Tabela 3.1. Czasy wykonania programów dla algorytmów różnej klasy

	10	20	30	40	50	60
n	0,000 01 s	0,000 02s	0,000 03 s	0,000 04 s	0,000 05 s	0,000 06
n^2	0,000 1 s	0,000 4s	0,000 09 s	0,001 6 s	0,002 5 s	0,003 6s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,216 s
2^n	0,001 s	1,0 s	17,9 min	12,7 dni	35,7 lat	366 w
3^n	0,59 s	58 min	6,5 lat	3855 w	200 •10 ⁶ w.	1,3•10 ¹³ w.
$n!$	3,6 s	768 w.	8,4•10 ¹⁶ w.	2,6•10 ³² w.	9,6•10 ⁴⁸ w.	2,6 10 ⁶⁶ w.

⁴ W toku dalszego wykładu okaże się, że nie jest to bynajmniej jedyny współczynnik decydujący o czasie wykonania programu.

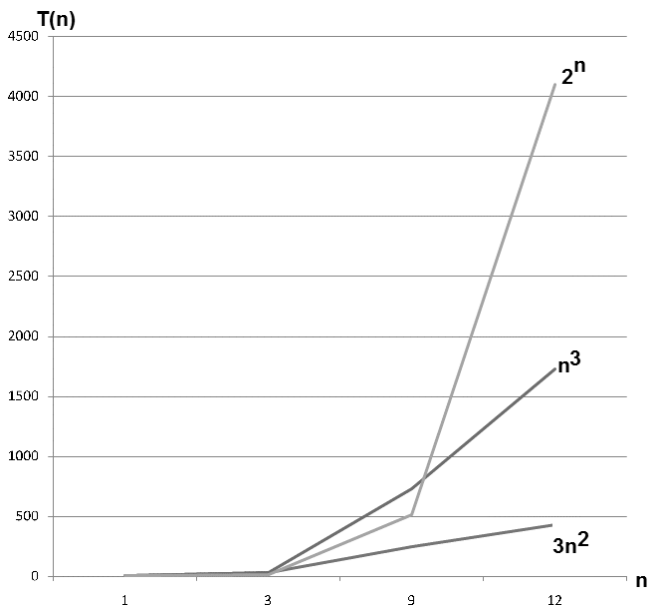
Tabela zawiera krótkie zestawienie czasów wykonania algorytmów⁵ przy następujących założeniach:

- ◆ Niech czas wykonania algorytmu A jest proporcjonalny do pewnej wybranej funkcji matematycznej, np. dla danej wejściowej o rozmiarze x i funkcji $n!$ czas wykonania programu jest proporcjonalny do $x!$.
- ◆ Niech czas wykonania operacji elementarnej wynosi jedną mikrosekundę (np. dla $n = 10$ i funkcji silnia wynik $10! = 3628800 / 1000000 = 3,6s$).

Przy powyższych założeniach można otrzymać zacytowane w tabelce wyniki — dość szokujące, zwłaszcza jeśli spojrzymy na ostatnie jej pozycje.

Popatrzmy jeszcze, jak szybko wzrasta czas wykonania programów dla pozornie bliskich sobie klas funkcji (rysunek 3.1), przy nawet małym wzroście n .

Rysunek 3.1.
Graficzna ilustracja
czasów wykonania
algorytmów różnej klasy



Wartości n zostały celowo dobrane jako bardzo małe, gdyż... tylko dla nich wyniki mieściły się na wykresie!

Wykres ten powinien nas doprowadzić do istotnej konkluzji: tylko dla małych klas złożoności (bliskich liniowej) przyspieszenie mocy obliczeniowej komputera faktycznie wpływa na możliwość rozwiązywania problemów. Proste symulacje pokazują, że dla „kosztownych” czasowo funkcji krótszy czas realizacji uzyskany z przyspieszenia jest i tak niewystarczający, aby pozwolić obsłużyć więcej problemów wejściowych.

Aby to dokładniej zrozumieć, spójrzmy na rysunek 3.2.

Widać na nim dwie tabelki czasów wykonania programów klasy $100n$ i 2^n , w zakresie n od 10 do 10 000 000 w wersji podstawowej i po 1000-krotnym zwiększeniu mocy obliczeniowej. Interesuje nas graniczny czas realizacji około milion (np. sekund) i sprawdzenie, jaki zakres wartości n może być obsłużony przez oba programy w tym zadanym czasie.

⁵ Oznaczenia: s — sekunda, w. — wiek.

Rysunek 3.2.

*Żłudna wiara
w moc komputera*

Przed przyspieszeniem			Po przyspieszeniu 1000x	
n	100n	2 ⁿ	100n	2 ⁿ
10	1 000	1 024	1	1
20	2 000	1 048 576	2	1 049
30	3 000	1 073 741 824	3	1 073 742
40	4 000	1 099 511 627 776	4	10 995 116 278
50	5 000	...	5	...
100	10 000	...	10	...
1 000	100 000	...	100	...
10 000	1 000 000	...	1 000	...
100 000	10 000 000	...	10 000	...
1 000 000	100 000 000	...	100 000	...
10 000 000	1 000 000 000	...	1 000 000	...

Okazuje się, że dla programu klasy $100n$ liczba możliwych do rozwiązania w zadanym czasie problemów wzrasta drastycznie (o trzy rzędy wielkości), gdy tymczasem dla drugiego algorytmu zyskujemy raptem możliwość obsłużenia mniej więcej dodatkowych 50% (praktyczny zakres $10 - 20$ rozszerza się zaledwie do $10 - 30$)!

Teraz każdy sceptyk powinien przyznać należyte miejsce dziedzinie wiedzy pozwalającej uniknąć nużącego, kilkusetwiecznego oczekiwania na efekt zadziałania programu...

Aby lepiej zrozumieć mechanizmy obliczeniowe używane przy analizie złożoności algorytmów, wspólnie zgłębimy kilka charakterystycznych przykładów obliczeniowych. Nowe pojęcia związane z obliczaniem złożoności obliczeniowej algorytmów zostaną wprowadzone na reprezentatywnych przykładach, co wydaje się lepszym rozwiązaniem niż zacytowanie suchych definicji.

Jeszcze raz funkcja silnia

Do zdumiewających zalet funkcji *silnia* należy niewątpliwie mnogość zagadnień, które można za jej pomocą zilustrować. Z rozdziału poprzedniego pamiętamy jeszcze zapewne rekurencyjną definicję:

$$0! = 1,$$

$$n! = n * (n-1)!, \quad n \geq 1, n \in N$$

Odpowiadająca tej formule funkcja w C++ ma następującą postać:

```
int silnia(int n)
{
    if (n==0)
        return 1;
    else
        return n*silnia(n-1);
}
```

Przyjmijmy dla uproszczenia założenie, bardzo zresztą charakterystyczne w tego typu zadaniach, że najbardziej czasochłonną operacją jest tutaj instrukcja porównania `if`. Przy takim założeniu czas, w jakim wykona się program, możemy zapisać również w postaci rekurencyjnej:

$$T(0) = t_c,$$

$$T(n) = t_c + T(n-1) \text{ dla } n \geq 1.$$

Powyższe wzory należy odczytać w sposób następujący: dla danej wejściowej równej zero czas wykonania funkcji, oznaczany jako $T(0)$, równa się czasowi wykonania jednej instrukcji porównania, oznaczonej symbolicznie przez t_c . Analogiczny czas dla danych wejściowych ≥ 1 jest równy, zgodnie z formułą rekurencyjną, $T(n) = t_c + T(n-1)$.

Niestety tego typu zapis jest nam do niczego nieprzydatny — trudno np. powiedzieć od razu, ile czasu zajmie obliczenie $\text{silnia}(100)$. Widać już, że do problemu należy podejść nieco inaczej. Zastanówmy się, jak z tego układu wyliczyć $T(n)$, tak aby otrzymać jakąś funkcję nierekurencyjną

pokazującą, jak czas wykonania programu zależy od danej wejściowej n . W tym celu spróbujmy rozpisać równania:

$$\begin{aligned} T(n) &= t_c + T(n-1), \\ T(n-1) &= t_c + T(n-2), \\ T(n-2) &= t_c + T(n-3), \\ &\vdots \\ T(1) &= t_c + T(0), \\ T(0) &= t_c. \end{aligned}$$

Jeśli dodamy je teraz stronami, to powinniśmy otrzymać:

$$T(n) + T(n-1) + \dots + T(0) = (n+1)t_c + T(n-1) + \dots + T(0),$$

co powinno dać, po zredukowaniu składników identycznych po obu stronach równości, następującą zależność:

$$T(n) = (n+1)t_c.$$

Jest to funkcja, która w satysfakcjonującej, nieskomplikowanej formie pokazuje, w jaki sposób rozmiar danej wejściowej wpływa na liczbę instrukcji porównań wykonanych przez program — czyli de facto na czas wykonania algorytmu. Znając bowiem parametr t_c i wartość n , możemy powiedzieć dokładnie, w ciągu ilu sekund (minut, godzin, lat...) wykona się algorytm na określonym komputerze.

Tego typu rezultat dokładnych obliczeń zwykło się nazywać *złożonością praktyczną* algorytmu. Funkcja ta jest zazwyczaj oznaczana tak jak wyżej — przez T .

W praktyce rzadko interesuje nas aż tak dokładny wynik. Dla odpowiednio dużych n niewiele bowiem się zmienia, jeśli zamiast $T(n) = (n+1)t_c$ otrzymamy $T(n) = (n+3)t_c!$

Do czego zmierzam? Otóż w dalszych rozważaniach będziemy głównie szukać odpowiedzi na pytanie:

Jaki *typ funkcji matematycznej*, występującej w zależności określającej złożoność praktyczną programu, odgrywa w niej najważniejszą rolę, wpływając najsilniej na czas wykonywania programu?



Definicja

Tę poszukiwaną funkcję będziemy zwać *złożonością teoretyczną* lub klasą algorytmu i z nią najczęściej można się spotkać przy opisach katalogowych określonych algorytmów. Funkcja ta jest najczęściej oznaczana przez O . Zastanówmy się, w jaki sposób możemy ją otrzymać.

Istnieją dwa klasyczne podejścia, prowadzące z reguły do tego samego rezultatu: albo będziemy opierać się na pewnych twierdzeniach matematycznych i je aplikować w określonych sytuacjach, albo też dojdziemy do prawidłowego wyniku metodą intuicyjną.

Wydaje mi się, że to drugie podejście jest zarówno szybkie, jak i znacznie przystępniejsze, dlatego skoncentrujemy się najpierw na nim. Popatrzmy w tym celu na tabelę 3.2 zawierającą kilka przykładów wyłuskiwania złożoności teoretycznej z równań określających złożoność praktyczną.

Wyniki zawarte w tej tabelce możemy wyjaśnić w następujący sposób: w równaniu $3n+1$ pozwolimy sobie pominąć stałą 1 i wynik nie ulegnie znaczącej zmianie. W równaniu n^2-n+1 o wiele ważniejsza jest funkcja kwadratowa niż liniowa zależność od n ; podobnie w ostatnim równaniu dominuje funkcja 2^n .

Tabela 3.2. *Złożoność teoretyczna algorytmów — przykłady*

$T(n)$	O
6	$O(1)$
$3n+1$	$O(n)$
n^2-n+1	$O(n^2)$
2^n+n^2+4	$O(2^n)$

W algorytmice spotyka się często kilka charakterystycznych funkcji O , oto kilka z nich:

- ♦ Klasa $O(1)$ umownie oznacza, że liczba operacji wykonywanych przez algorytm jest niezależna od rozmiarów problemu⁶. Czy taka sytuacja w przypadku algorytmów nietrywialnych w ogóle może mieć miejsce? Okazuje się, że w pewnym przybliżeniu tak, weźmy na przykład funkcje wyszukiwania oparte na metodzie transformacji kluczowej (hashing) omówionej w rozdziale 7. Teoretycznie wielkość zbioru do przeszukania nie ma znaczenia, jeśli pewna dobra funkcja H pozwoli nam dotrzeć do poszukiwanego rekordu w mniej więcej tym samym, skończonym czasie. Mam świadomość, że teraz brzmi to enigmatycznie, ale po lekturze wspomnianego rozdziału na pewno zgodzisz się z tym stwierdzeniem.
- ♦ Klasa $O(n)$ oznacza, że algorytm wykonuje się w czasie proporcjonalnym do rozmiaru problemu. Przykładem takiego algorytmu może być przetwarzanie sekwencyjne ciągu znaków, obsługa kolejki itp. Jest to prosta zależność liniowa, gdzie każdej z n danych wejściowych algorytm musi poświęcić pewien czas na wykonanie swoich obliczeń.
- ♦ Złożoność typu $O(\log n)$ jest lepsza od liniowej⁷, co arytmetycznie oznacza, że jeśli klasa problemu rośnie geometrycznie (np. o rząd wielkości, ze 100 na 1 000), to wzrost złożoności będzie arytmetyczny (tutaj dwukrotny). Jeśli n rośnie niezbyt szybko, to algorytm zwalnia, ale nie drastycznie. Ze złożonością logarytmiczną spotkamy się np. w algorytmie przeszukiwania posortowanej tablicy, gdzie w każdym kroku algorytmu będziemy pomijali część danych (patrz rozdział 7.).
- ♦ Klasa $O(n^2)$ jest często spotykana w rozważaniach arytmetycznych lub kombinatorycznych, gdzie mamy do czynienia z regułą „każdy z każdym”. Przykładem może być dodawanie macierzy o rozmiarach $n \times n$. Przez analogię dla klasy $O(n^3)$ można jako przykład podać mnożenie macierzy o rozmiarach $n \times n$. Nie powinno być dla nikogo niespodzianką, że algorytmy „kwadratowe” i wyższe nadają się do wykorzystania dla raczej małych wartości n .
- ♦ Klasa wykładnicza $O(2^n)$ jest często przytaczana jako swego rodzaju straszak, choć w praktyce algorytmy tej klasy mogą być też używane, oczywiście jeśli zwracają wyniki w sensownym dla użytkownika czasie.

Poglądowe przykłady na funkcję O można by jeszcze mnożyć, ale pojęcie to jest na tyle kluczowe, że przytoczę formalną definicję matematyczną.

W tym celu odświeżmy następujące oznaczenia znane z podręczników analizy matematycznej:

- ♦ \mathbb{N}, \mathbb{R} są zbiorami liczb odpowiednio naturalnych i rzeczywistych (wraz z zerem).
- ♦ \mathbb{N}^+ jest zbiorem liczb naturalnych dodatnich.
- ♦ Za pomocą \mathbb{R}^+ będziemy oznaczać zbiór liczb rzeczywistych dodatnich łącznie z zerem.
- ♦ Znak graficzny \mapsto oznacza przyporządkowanie.

⁶ Co wcale nie oznacza, że będzie mała; ta „1” jest często mylnie utożsamiana z pojedynczą instrukcją!

⁷ Przypomnę, że logarytm liczby $x > 0$ o podstawie $b \neq 1$, oznaczony jako $u = \log_b x$, jest to taka liczba u spełniająca zależność $b^u = x$, np. $3 = \log_2 8$.

- ◆ Znak graficzny \forall należy czytać jako „dla każdego”.
- ◆ Znak graficzny \exists należy czytać jako „istnieje”.
- ◆ Znak graficzny \in należy czytać jako „należy do” lub „należący do”.
- ◆ Małe litery pisane *kursywą* na ogół oznaczają nazwy funkcji (np. g).
- ◆ Dwukropek zapisany po pewnym symbolu S należy odczytywać: S *taki, że...*

Bazując na powyższych oznaczeniach, klasę O dowolnej funkcji $T: \mathbb{N} \mapsto \mathbb{R}^*$ możemy zdefiniować jako:

$$O(T(n)) = \{g : T : \mathbb{N} \mapsto \mathbb{R}^* \mid (\exists M \in \mathbb{R}^+) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) [|g(n)| \leq |M \cdot T(n)|]\}$$

Jak wynika z powyższej definicji, klasa O (wedle definicji jest to zbiór funkcji) ma charakter wielkości asymptotycznej, pozwalającej wyrazić w postaci arytmetycznej wielkości z góry nieznane w postaci analitycznej. Samo istnienie tej notacji pozwala na znaczne uproszczenie wielu dociekań matematycznych, w których dokładna znajomość rozważanych wielkości nie jest konieczna.

Dysponując tak formalną definicją, można łatwo udowodnić pewne „oczywiste” wyniki, np.: $|5n^3 + 3n^2 + 2| \in O(n^3)$, (można dobrać doświadczalnie takie M i n_0 , dla których zawsze będzie spełnione $\{|5n^3 + 3n^2 + 2| \leq M \cdot |n^3|\}$, dla każdego $n \geq n_0$). W sposób zbliżony można przeprowadzić dowody wielu podobnych zadań.

Funkcja O jest wielkością, której można używać w równaniach matematycznych. Notacja dużego O jest czasami zwana notacją Landaua od nazwiska niemieckiego matematyka (1877 – 1938) żydowskiego pochodzenia, autora wielu prac z teorii liczb, który miał wątpliwe szczęście urodzić się w Berlinie i żyć w Niemczech w okresie kulminacji polityki nazizmu i prześladowań etnicznych.

Oto kilka własności, które mogą posłużyć do znacznego uproszczenia wyrażeń je zawierających:

$$\begin{aligned} c \cdot O(f(n)) &= O(f(n)) \\ O(f(n)) + O(f(n)) &= O(f(n)) \\ O(O(f(n))) &= O(f(n)) \\ O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)) \\ O(f(n) \cdot g(n)) &= f(n) \cdot O(g(n)) \end{aligned}$$

Do ciekawszych należy pierwsza z powyższych własności, która znosi wpływ wszelkich współczynników o wartościach stałych. Własność ta pomoże nam dalej zrozumieć, dlaczego w literaturze mówi się, że „algorytm A jest klasy $O(\log N)$ ”, a nie „ $O(\log_2 N)$ ”.

Ktoś o bardzo radykalnym podejściu do wszelkich sztucznych założeń, mających ułatwić wyliczenie pewnych normalnie skomplikowanych zagadnień, mógłby zakwestionować przyjmowanie podstawy 2 za punkt odniesienia, zapytując się przykładowo: „a dlaczego nie 2,5 lub 3”? Pozornie takie postawienie sprawy wydaje się słuszne, ale na szczęście tylko pozornie!

Najpierw zauważmy, że używanie dwójki jako podstawy obliczeń jest pewną manierą związaną z rozpowszechnieniem systemu dwójkowego. Z tego powodu zakładamy, że np. rozmiar tablicy jest wielokrotnością liczby 2 i następnie na podstawie takich założeń częstokroć wyliczana jest złożoność praktyczna i z niej dedukowana jego klasa, czyli funkcja O .

Przypomnijmy sobie jednak elementarny wzór podający zależność pomiędzy logarytmami o różnych podstawach:

$$\log_a N = \frac{\log_b N}{\log_b a} = \log_b N \cdot \frac{1}{\log_b a}.$$

Widać wyraźnie, że logarytmy o odmiennych podstawach (a i b) różnią się pomiędzy sobą tylko pewnym współczynnikiem stałym, który zostanie „pochłonięty” przez O na podstawie własności:

$$c \cdot O(f(n)) = O(f(n)).$$

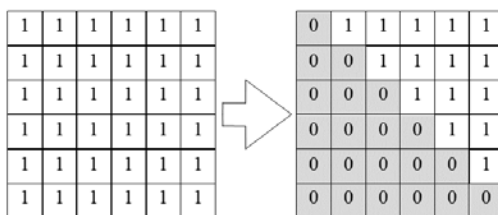
Popatrzmy jeszcze na inny aspekt stosowania O -notacji. Załóżmy, że pewien algorytm A został wykonany w dwóch wersjach: $W1$ i $W2$, charakteryzujących się złożonością praktyczną odpowiednio: $100 \log_2 N$ i $10N$. Na podstawie uprzednio poznanych własności możemy szybko określić, że $W1 \in O(\log N)$, $W2 \in O(N)$, czyli $W1$ jest lepszy od $W2$. Niestety ktoś szczególnie złośliwy mógłby się uprzeć, że jednak algorytm $W2$ jest lepszy, bowiem dla np. $N = 2$ mamy $100 \log_2 2 > 10 \cdot 2$... Wobec takiego stwierdzenia nie należy wpadać w panikę, tylko wziąć do ręki odpowiednio duże N , dla którego algorytm $W1$ okaże się jednak lepszy od $W2$! Nie należy bowiem zapominać, że O -notacja ma charakter asymptotyczny i jest prawdziwa dla „odpowiednio dużych wartości N ”.

Zerowanie fragmentu tablicy

Rozwiążemy teraz następujący problem: jak wyzerować fragment tablicy (tzn. macierzy) poniżej przekątnej (wraz z nią)? Tę ideę przedstawia rysunek 3.3.

Rysunek 3.3.

Koszt zerowania tablicy



Funkcja wykonująca to zadanie jest bardzo prosta:

```
int tab[n][n];
void zerowanie()
{
    int i, j;
    i=0; // ta
    while (i<n) // tc
    {
        j=0; // ta
        while (j<=i) // tc
        {
            tab[i][j]=0; // ta
            j=j+1; // ta
        }
        i=i+1; // ta
    }
}
```

Oznaczenia:

- ♦ t_a — czas wykonania instrukcji *przypisania*;
- ♦ t_c — czas wykonania instrukcji *porównania*.

Do dalszych rozważań niezbędne będzie zrozumienie funkcjonowania pętli typu `while`:

```
i=1;
while (i<=n)
{
```

```

instrukcje:
i=i+1;
}

```

Jej działanie polega na wykonaniu n razy instrukcji zawartych pomiędzy nawiasami klamrowymi, warunek natomiast jest sprawdzany $n+1$ razy⁸.

Korzystając z powyższej uwagi oraz informacji zawartych w liniach komentarza, możemy napisać:

$$T(n) = t_c + t_a + \sum_{i=1}^N \left(2t_a + 2t_c + \sum_{j=1}^i (t_c + 2t_a) \right).$$

Po usunięciu sumy z wewnętrznego nawiasu otrzymamy:

$$T(n) = t_c + t_a + \sum_{i=1}^N (2t_a + 2t_c + i(t_c + 2t_a)). \quad (*)$$

Przypomnijmy jeszcze użyteczny wzór na sumę szeregu liczb naturalnych od 1 do N :

$$1 + 2 + 3 + \dots + N = \frac{N(N+1)}{2}.$$

Po jego zastosowaniu w równaniu (*) otrzymamy:

$$T(n) = t_c + t_a + 2N(t_a + t_c) + \frac{N(N+1)}{2}(t_c + 2t_a).$$

Ostateczne uproszczenie wyrażenia powinno nam dać:

$$T(n) = t_a(1 + 3N + N^2) + t_c\left(1 + 2,5t_c + \frac{N^2}{2}\right);$$

co sugeruje od razu, że analizowany program jest klasy $O(n^2)$.

Ufff!

Nie było to przyjemne, prawda? A problem wcale nie należał do specjalnie złożonych. Nie zrażajmy się jednak trudnym początkiem, wkrótce okaże się, że można było zrobić to znacznie prościej! Do tego potrzebna nam będzie odrobina wiedzy teoretycznej, dotyczącej rozwiązywania równań rekurencyjnych. Poznamy ją szczegółowo po przerobieniu kolejnego przykładu zawierającego pewną pułapkę, której istnienie trzeba niestety co najmniej raz sobie uświadomić.

Wpadamy w pułapkę

Zadania z dwóch poprzednich przykładów charakteryzowała istotna cecha: czas wykonania programu nie zależał od wartości, jakie przybierała dana, lecz tylko od jej rozmiaru. Niestety nie zawsze tak jest! Takiemu właśnie przypadkowi poświęcone jest kolejne zadanie obliczeniowe. Jest to fragment większego programu, którego rola nie jest dla nas istotna w tym miejscu. Założmy, że otrzymujemy ten wyrwany z kontekstu fragment kodu i musimy się zająć jego analizą:

```

const int N=10;
int t[N];
funkcja_ad_hoc()
{
    int k,i;
    int suma=0;           // t_a

```

⁸ Warto zauważyć, że istniejące w C++ pętle łatwo dają się sprowadzić do odmiany pętli zacytowanej powyżej.

```

while (i<N)                // tc
{
    while (j<=t[i])         // tc
    {
        suma=suma+2;        // ta
        j=j+1;              // ta
    }
    i=i+1;                  // ta
}

```

Uprośćmy nieco problem, zakładając, że:

- ♦ Najbardziej czasochłonne są instrukcje porównania, wszelkie inne zaś ignorujemy jako niemające większego wpływu na czas wykonania programu.
- ♦ Zamiast pisać explicite t_c , wprowadzimy pojęcie czasu jednostkowego wykonania instrukcji, oznaczając go przez 1.

Niestety jedno zasadnicze utrudnienie pozostanie aktualne: nie znamy zawartości tablicy, a zatem nie wiemy, ile razy wykona się wewnętrzna pętla `while`! Popatrzmy, jak możemy sobie poradzić w tym przypadku:

$$T(n) = t_c + \sum_{i=1}^N \left(t_c + \sum_{j=1}^{t[i]} t_c \right), \quad (*)$$

$$T(n) = t_c + Nt_c + \sum_{i=1}^N t[i]t_c, \quad (**)$$

$$T(n) = t_c + Nt_c + Nt[i]t_c,$$

$$T(n) = t_c (1 + N + Nt[i]),$$

$$T(n) \approx \max(N, Nt[i]).$$

Początek jest klasyczny: zewnętrzna suma od 1 do N z równania (*) zostaje zamieniona na N -krotny iloczyn swojego argumentu. Podobny trik zostaje wykonany w równaniu (**), po czym możemy już spokojnie zająć się grupowaniem i upraszczaniem. Czas wykonania programu jest proporcjonalny do większej z liczb: N i $Nt[i]$ i tylko tyle możemy na razie stwierdzić. Niestety, kończąc w tym miejscu rozważania, wpadlibyśmy w pułapkę. Naszym problemem jest bowiem nieznanie zawartości tablicy, a ta jest potrzebna do otrzymania ostatecznego wyniku! Nie możemy przecież zastosować funkcji matematycznej do wartości nieokreślonej.

Nasze obliczenia doprowadziły zatem do momentu, w którym zauważamy brak pełnej informacji o rozważanym problemie. Gdybyśmy przykładowo wiedzieli, że o tym fragmencie programu, w którym pracuje nasza funkcja, można z dużym prawdopodobieństwem powiedzieć, iż tablica wypełniona jest głównie zerami, to nie byłoby w ogóle problemu! Nie mamy jednak żadnej pewności, czy rzeczywiście zajdzie taka sytuacja. Jedynym rozwiązaniem wydaje się zwrócenie do jakiegoś matematyka, aby ten — po przyjęciu dużej liczby założeń — przeprowadził analizę statystyczną zadania i doprowadził do ostatecznego wyniku w satysfakcjonującej nas postaci.

Różne typy złożoności obliczeniowej

Rozważmy ponownie problem: należy sprawdzić, czy pewna liczba x znajduje się w tablicy o rozmiarze n . Został już on rozwiązany w rozdziale 2., spróbujmy teraz napisać iteracyjną wersję tej samej procedury. Nie jest to czynność szczególnie skomplikowana i sprowadza się do napisania następującego programu:



szukaj.cpp

```

const int n=10;
int tab[n]={1,2,3,2,-7,44,5,1,0,-3};
int szukaj(int tab[n], int x)
{
    int pos=0;
    while ((pos<n) && (tab[pos]!=x)) pos++;
    if (pos<n)
        return pos; // element znaleziony
    else
        return -1; // porażka poszukiwań
}

int main()
{
    cout << szukaj(tab,7) <<endl; //wynik = -1
    cout << szukaj(tab,5) <<endl; //wynik = 6
}

```

Idea tego algorytmu polega na sprawdzeniu, czy w badanym fragmencie tablicy lewy skrajny element jest poszukiwaną wartością x . Wywołując procedurę w następujący sposób: `szukaj(tab, x)`, powodujemy przebadanie całej tablicy o rozmiarze n . Co można powiedzieć o złożoności obliczeniowej tego algorytmu, przyjmując jako kryterium liczbę porównań wykonanych w pętli `while`? Na tak sformułowane pytanie można się niestety tylko obruszyć i mruknąć: „To zależy, gdzie znajduje się x ”! Istotnie mamy do czynienia z co najmniej dwoma skrajnymi przypadkami:

- ◆ Znajdujemy się w komórce `tab[0]`, czyli $T(n) = 1$ i trafiamy na tzw. *najlepszy przypadek*.
- ◆ W poszukiwaniu x przeglądamy całą tablicę, czyli $T(n) = n$ i trafiamy na tzw. *najgorszy przypadek*.

Jeśli na jedno precyzyjne pytanie: „Jaka jest złożoność obliczeniowa algorytmu liniowego przeszukiwania tablicy n -elementowej?” otrzymujemy dwie odpowiedzi, obarczone klauzulami „jeśli”, „w przypadku gdy...”, to jedno jest pewne: odpowiedzi na pytanie ciągle nie mamy!

Błąd tkwił oczywiście w pytaniu, które powinno uwzględniać konfigurację danych, a ma ona w przypadku przeszukiwania tablicy kluczowe znaczenie. Proponowane odpowiedzi mogą być zatem następujące: rozważany algorytm ma w najlepszym przypadku złożoność praktyczną równą $T(n) = 1$, a w najgorszym przypadku — $T(n) = n$. Ponieważ jednak życie toczy się raczej równomiernie i nie balansuje pomiędzy skrajnościami (co jest dość prowokacyjnym stwierdzeniem, ale przyjmijmy chwilowo, że jest to prawda), warto byłoby poznać również odpowiedź na pytanie: jaka jest *średnia* wartość $T(n)$ tego algorytmu? Należy ono do gatunku nieprecyzyjnych, jest zatem stworzone dla statystyka. Nie pozostaje nam nic innego, jak przeprowadzić analizę statystyczną omawianego algorytmu.

Oznaczmy przez p prawdopodobieństwo, że x znajduje się w tablicy `tab`, i przypuśćmy, że jeśli istotnie x znajduje się w tablicy, to wszystkie miejsca są jednakowo prawdopodobne.

Oznaczmy również przez $D_{n,i}$ (gdzie $0 \leq i < n$) zbiór danych, dla których x znajduje się na i -tym miejscu tablicy i przez $D_{n,n}$ zbiór danych, gdzie x jest nieobecne. Wedle przyjętych wyżej oznaczeń możemy napisać, że:

$$P(D_{n,i}) = \frac{p}{n} \quad \text{ i } \quad P(D_{n,n}) = 1 - p.$$

Koszt algorytmu oznaczmy klasycznie przez T , tak więc:

$$T(D_{n,i}) = i \quad \text{ oraz } \quad T(D_{n,n}) = n.$$

Otrzymujemy zatem wyrażenie:

$$T_{\text{średnie}} = \sum_{i=0}^N P(D_{n,i}) T(D_{n,i}) = (1-p)n + \sum_{i=0}^{n-1} i \frac{p}{n} = (1-p)n + (n+1) \frac{p}{2}.$$

Przykładowo: wiedząc, że x na pewno znajduje się w tablicy ($p = 1$), możemy od razu napisać:

$$T_{\text{średnie}} = (1-1)n + \frac{(n+1)}{2} = \frac{(n+1)}{2}.$$

Zdefiniowaliśmy zatem trzy podstawowe typy złożoności obliczeniowej (dla przypadków: *najgorszego*, *najkorzystniejszego* i *średniego*), warto teraz zastanowić się nad użytecznością praktyczną tych pojęć. Z matematycznego punktu widzenia te trzy określenia definiują w pełni zachowanie się algorytmu, ale czy aby na pewno robią to dobrze?

W katalogowych opisach algorytmów najczęściej mamy do czynienia z rozważaniami na temat przypadku *najgorszego* — tak aby wyznaczyć sobie pewną górną granicę, której algorytm na pewno nie przekroczy (jest to informacja najbardziej użyteczna dla programisty).

Przypadek *najkorzystniejszy* ma podobny charakter, dotyczy jednak progu dolnego czasu wykonywania programu.

Widzimy, że pojęcia złożoności obliczeniowej programu w przypadkach *najlepszym* i *najgorszym* mają sens nie tylko matematyczny, lecz dają programiście pewne granice, w których może on go umieścić. Czy podobnie możemy rozpatrywać przypadek *średni*?

Jak łatwo zauważyć, wyliczenie przypadku średniego (inaczej to określając: *typowego*) nie jest łatwe i wymaga założenia szeregu hipotez dotyczących możliwych konfiguracji danych. Między innymi musimy umówić się co do definicji zbioru danych, z którym program ma do czynienia — niestety zazwyczaj nie jest to ani możliwe, ani nie ma żadnego sensu! Programista dostający informację o średniej złożoności obliczeniowej programu powinien być zatem świadomy tych ograniczeń i nie brać tego parametru za informację wzorcową.

Nowe zadanie: uprościć obliczenia!

Nie sposób pominąć faktu, że wszystkie nasze dotychczasowe zadania były dość skomplikowane rachunkowo, a tego leniwi ludzie (czytaj: programiści) nie lubią. Jak zatem postępować, aby wykonać tylko te obliczenia, które są naprawdę niezbędne do otrzymania wyniku? Otóż warto zapamiętać następujące sztuczki, które znacznie ułatwią nam to zadanie, pozwalając niejednokrotnie natychmiastowo określić poszukiwany wynik:

- ♦ W analizie programu zwracamy uwagę tylko na najbardziej czasochłonne operacje (np. poprzednio były to instrukcje porównań).
- ♦ Wybieramy jeden wiersz programu znajdujący się w najgłębiej położonej instrukcji iteracyjnej (pętla w pętlach, a te jeszcze w innych pętlach...), a następnie obliczamy, ile razy się on wykona. Z tego wyniku dedukujemy złożoność teoretyczną.

Pierwszy sposób był już wcześniej stosowany. Aby wyjaśnić nieco szerzej drugą metodę, proponuję przestudiować poniższy fragment programu:

```
while (i<N)
{
    while (j<=N)
    {
        suma=suma+2;
        j=j+1;
    }
}
```

Wybieramy instrukcję $\text{suma}=\text{suma}+2$ i obliczamy w prosty sposób, iż wykona się ona $\frac{N(N+1)}{2}$ razy. Wnioskujemy, że ten fragment programu ma złożoność teoretyczną równą $O(n^2)$.

Analiza programów rekurencyjnych

Większość programów rekurencyjnych nie da się, niestety, rozważyć przy użyciu poznanej wcześniej metody. Istotnie zastosowana tam metoda rozwiązywania równania rekurencyjnego, polegająca na rozpisaniu jego składników i dodaniu stronami układu równań, nie zawsze się sprawdza. U nas doprowadziła ona do sukcesu, tzn. do uproszczenia obliczeń — niestety, zazwyczaj równania potraktowane w ten sposób jeszcze bardziej się komplikują.

W tym paragrafie przedstawiona zostanie metoda mająca charakter o wiele ogólniejszy. Ma ona swoje uzasadnienie matematyczne, którego z powodu jego skomplikowania nie będę przedstawiał. Osoby szczególnie zainteresowane stroną matematyczną powinny dotrzeć bez kłopotu do odpowiedniej literatury (patrz uwagi zamieszczone we wstępie rozdziału).

Terminologia i definicje

Lektura kilku następnych paragrafów wymaga od nas poznania terminologii, którą będziemy się dość często posługiwać. Pomimo ich „groźnego” wyglądu zrozumienie poniższych definicji nie powinno Czytelnikowi sprawić szczególnych kłopotów.

Szereg rekurencyjny liniowy SRL jest to szereg o następującej postaci:

$$X_{n+r, n \geq 0} = \sum_{i=1}^r a_i X_{n+r-i} + u(n, m).$$

Oznaczenia:

$u(n, m)$ — nierekurencyjna reszta równania, będąca wielomianem stopnia m i zmiennej n .

Przykładowo:

- ◆ Jeśli $u(n, m) = 3n+1$, to mamy wielomian stopnia *pierwszego*.
- ◆ Jeśli $u(n, m) = 2$, to jest to wielomian stopnia *zerowego*.

Uwagi:

- ◆ Współczynniki a_i są dowolnymi liczbami rzeczywistymi.
- ◆ r jest liczbą całkowitą.

Skomplikowaną postacią tego wzoru nie należy się przejmować, jest to po prostu sformalizowany zapis ogólnego równania rekurencyjnego, podany raczej gwoili formalności niż w jakimś praktycznym celu.



Definicja

Równanie charakterystyczne RC jest to wielomian sztucznie stworzony na podstawie równania rekurencyjnego, powstały wg wzoru:

$$R(x) = X^r - \sum_{i=1}^r a_i x^{r-i}.$$

Równanie to można rozwiązać, otrzymując rozkład postaci:

$$R(x) = \prod_{i=1}^p (x - \lambda_i)^{m_i}.$$

Przykład: $SRL = x_n - 3x_{n-1} + 2x_{n-2} = 0$ daje $R(x) = x^2 - 3x + 2 = (x-1)(x-2)$.



Definicja

Otrzymane powyżej współczynniki λ_i posłużą do skonstruowania tzw. **rozwiązania ogólnego RO** liniowego równania rekurencyjnego:

$$RO = \sum_{i=1}^p P_i \lambda_i.$$

Dodatkowo będziemy potrzebować tzw. **rozwiązania szczególnego RS** liniowego równania rekurencyjnego.

Postać **RS** zależy od formy, jaką przybiera reszta $u(n, m)$. Poszczególne przypadki rozpisane są poniżej.

- ♦ Jeśli $u(n, m) = 0$, to $RS = 0$.
- ♦ Jeśli $u(n, m)$ jest wielomianem stopnia m i zmiennej n oraz 1 (jeden) *nie jest* rozwiązaniem RC, wówczas $RS = Q(n, m)$, gdzie $Q(n, m)$ jest pewnym wielomianem stopnia m i zmiennej n o współczynnikach nieznanach (do odnalezienia).
- ♦ Jeśli $u(n, m)$ jest wielomianem stopnia m i zmiennej n oraz 1 jest rozwiązaniem RC, wtedy $RS = n^p Q(n, m)$, gdzie p jest stopniem pierwiastka.

Przykładowo: jeśli jedynka jest pierwiastkiem pojedynczym RC, to $p = 1$, jeśli pierwiastkiem podwójnym, to $p = 2$ itd.

- ♦ Jeśli $u(n, m) = \alpha^n$ i α *nie jest* rozwiązaniem RC, wtedy:

$$RS = c\alpha^n.$$

- ♦ Jeśli $u(n, m) = \alpha^n$ i α *jest* pierwiastkiem stopnia p RC, wtedy:

$$RS = c\alpha^n n^p.$$

- ♦ Jeśli $u(n, m) = \alpha^n W(n, m)$ i α *nie jest* rozwiązaniem RC (tradycyjnie już $W(n, m)$ jest pewnym wielomianem stopnia m i zmiennej n), będziemy wówczas mieli:

$$RS = \alpha^n S(n, m),$$

gdzie $S(n, m)$ jest pewnym wielomianem stopnia m i zmiennej n .



Uwaga

Występujące po prawej stronie wzorów wielomiany i stałe mają charakter zmiennych, które należy odnaleźć!

Rozwiązaniem równania rekurencyjnego jest suma obu równań: *ogólnego i szczególnego*.

Cały ten bagaż wzorów był naprawdę niezbędny! Dla zilustrowania metody rozwiążemy proste zadanie.

Ilustracja metody na przykładzie

Spójrzmy jeszcze raz na przykład ze strony 53 dotyczący funkcji silnia. Otrzymaliśmy wtedy następujące równanie:

$$\begin{aligned} T(0) &= 1, \\ T(n) &= 1 + T(n-1). \end{aligned}$$

Spróbujmy je rozwiązać nowo poznaną metodą.

♦ **ETAP 1.** Poszukiwanie równania charakterystycznego:

Z postaci ogólnej $SRL = T(n) - T(n-1)$ wynika, że $RC = x - 1$.

♦ **ETAP 2.** Pierwiastek równania charakterystycznego:

Jest to oczywiście $r = 1$.

♦ **ETAP 3.** Równanie ogólne:

$RO = Ar^n$, gdzie A jest jakąś stałą do odnalezienia. Ponieważ $r = 1$, to $RO = A$ (1 podniesione do dowolnej potęgi da nam oczywiście 1). Stałą A wyliczymy dalej.

♦ **ETAP 4.** Poszukiwanie równania szczególnego:

Wiemy, że $u(n, m) = 1$ (jeden jest wielomianem stopnia zero!). Ponadto 1 jest pierwiastkiem pierwszego stopnia równania charakterystycznego. Tak więc:

$$S = n^p c = n \cdot c.$$

Pozostaje nam jeszcze do odnalezienia stała c . Wiemy, że RS musi spełniać pierwotne równanie rekurencyjne, zatem po podstawieniu go jako $T(n)$ otrzymamy:

$$\begin{aligned} n \cdot c &= 1 + (n-1)c, \\ n \cdot c &= 1 + n \cdot c - c, \\ c &= 1. \end{aligned}$$

♦ **ETAP 5.** Poszukiwanie ostatecznego rozwiązania:

Wiemy, że ostatecznym rozwiązaniem równania jest suma RO i RS :

$T(n) = RO + RS = A + n \cdot c = A + n$. Stałą A możemy z łatwością wyliczyć poprzez podstawienie przypadku elementarnego:

$$\begin{aligned} T(0) &= 1, \\ 1 &= A + 0, \\ A &= 1. \end{aligned}$$

Po tych karkołomnych wyliczeniach otrzymujemy: $T(n) = n + 1$.

Jest ono identyczne z poprzednim rozwiązaniem⁹.

Metoda równań charakterystycznych jest, jak widać, bardzo elastyczna. Pozwala ona na szybkie określenie złożoności algorytmicznej nawet dość rozbudowanych programów. Są oczywiście zadania wymagające interwencji matematyka, ale zdarzają się one rzadko i dotyczą zazwyczaj programów rekurencyjnych o nikłym znaczeniu praktycznym.

Rozkład logarytmiczny

Z poprzedniego rozdziału pamiętamy zapewne zadanie poświęcone przeszukiwaniu binarnemu. Jedną z możliwych wersji funkcji¹⁰ wykonującej to zadanie jest:

```
int binary_search(int *tab, int x, int left, int right)
{
    if (left==right)
        if (t[left]==x)
            return left;
        else
            return -1;
    else
        return -1;
}
```

⁹ Jeśli dwie metody prowadzą do takiego samego, prawidłowego wyniku, to istnieje duże prawdopodobieństwo, iż obie są dobre.

¹⁰ Nieco innej niż poprzednio zaproponowana.


```

int mid=(left+right)/2;
if (tab[mid]==x)
    return mid;      // element znaleziony!
else
    if (x<tab[mid])
        return binary_search(tab,x,left,mid);
    else
        return binary_search(tab,x,mid,right);
}
}

```

Jaka jest złożoność obliczeniowa tej funkcji? Analiza liczby instrukcji porównań prowadzi nas do następujących równości:

$$T(1) = 1 + 1 = 2,$$

$$T(n) = 1 + 1 + T\left(\frac{n}{2}\right) = 2 + T\left(\frac{n}{2}\right).$$

Widać już, że powyższy układ ma się nijak do podanej poprzednio metody. W określeniu równania charakterystycznego przeszkadza nam owo dzielenie n przez 2. Otóż można z tej pułapki wybrnąć, np. przez podstawienie $n = 2p$, ale ciąg dalszy obliczeń będzie dość złożony. Na szczęście matematycy zrobili w tym miejscu programistom miły prezent: bez żadnych skomplikowanych obliczeń można określić złożoność tego typu zadań, korzystając z kilku gotowych reguł. Prezent ten jest tym bardziej cenny, że zadania o rozkładzie podobnym do powyższego występują bardzo często w praktyce programowania. Przed ostatecznym jego rozwiązaniem musimy zatem poznać jeszcze kilka wzorów matematycznych, ale obiecuję, że na tym już będzie koniec, jeśli chodzi o matematykę „wyższą”.

Założmy, że ogólna postać otrzymanego układu równań rekurencyjnych przedstawia się następująco:

$$T(1) = 1,$$

$$T(n) = aT\left(\frac{n}{b}\right) + d(n).$$

(Przy założeniu, że $n \geq 2$ oraz a i b są pewnymi stałymi).

W zależności od wartości a , b i $d(n)$ otrzymamy różne rozwiązania zgrupowane w tabeli 3.3.

Tabela 3.3. Analiza przeszukiwania binarnego

Klasa algorytmu		
$a > d(b)$	$T(n) \in O(n^{\log_b a})$	
$a < d(b)$	$T(n) \in O(n^{\log_b d(b)})$	gdy $d(n) = n^\alpha$ to $T(n) \in O(n^\alpha) = O(d(n))$
$a = d(b)$	$T(n) \in O(n^{\log_b d(b)} \log_b n)$	gdy $d(n) = n^\alpha$ to $T(n) \in O(n^\alpha \log_b n)$

Wzory te są wynikiem dość skomplikowanych wyliczeń bazujących na następujących założeniach:

- ♦ n jest potęgą b , co pozwala wykonać podstawienie $n = b^k$ sprowadzające równanie nieliniowe do równania $T(b^k) = aT(b^{k-1}) + d(b^k)$. Podstawiając ponadto $t_k = T(b^k)$, otrzymujemy równanie liniowe $t_k = at_{k-1} + d(b^k)$ z warunkiem początkowym $t_0 = 1$. Dyskusja wyników tego równania prowadzi do wniosków końcowych, przedstawionych w tabeli 3.3.
- ♦ Funkcja $d(n)$ musi spełniać następującą własność: $d(xy) = d(x)d(y)$ (np. $d(n) = n^2$ spełnia tę własność, a $d(n) = n-1$ już nie).

Pomimo tych ograniczeń okazuje się, iż bardzo duża klasa równań może być dzięki powyższemu wzorom z łatwością rozwiązana. Spróbujmy dla przykładu skończyć zadanie dotyczące przeszukiwania binarnego. Jak pamiętamy, otrzymaliśmy wówczas następujące równania:

$$T(1) = 2,$$

$$T(n) = 2 + T\left(\frac{n}{2}\right).$$

Patrząc na zestaw podanych powyżej wzorów, widzimy, że nie jest on zgodny z wzorcem podanym wcześniej. Nic nie stoi jednak na przeszkodzie, aby za pomocą prostego podstawienia doprowadzić do postaci, która będzie nas satysfakcjonowała:

$$\begin{aligned} U(n) = T(n) - 1 &\Leftrightarrow U(1) = T(1) - 1 = 1, \\ T(n) - 1 = 1 + T\left(\frac{n}{2}\right) &\Leftrightarrow U(n) = U\left(\frac{n}{2}\right) + 1. \end{aligned}$$

Identyfikujemy wartości stałych: $a = 1$, $b = 2$ i $d(n) = 1$, co pozwala nam zauważyć, iż zachodzi przypadek trzeci: $a = d(b)$. Poszukiwany wynik ma zatem postać:

$$U(n) \in O(n^{\log_2 1} \log_2 n) = O(n^0 \log_2 n) = O(\log_2 n).$$

Zamiana dziedziny równania rekurencyjnego

Pewna grupa równań charakteryzuje się zdecydowanie nieprzyjemnym wyglądem i nijak nie odpowiada podanym uprzednio wzorom i metodom. Czasem jednak zwykła zmiana dziedziny powoduje, iż rozwiązanie pojawia się niemal natychmiastowo. Przeanalizujmy następujący przykład:

$$\begin{aligned} a_n &= 3_{n-1}^2 \text{ dla } n \geq 1, \\ a_0 &= 1. \end{aligned}$$

Równanie nie jest zgodne z żadnym poznanym wcześniej schematem. Podstawmy jednak $b_n = \log_2 a_n$ i zlogarytmujmy obie strony równania:

$$\log_2 a_n = \log_2 (3_{n-1}^2),$$

otrzymując w efekcie:

$$\left. \begin{aligned} b_n &= 2b_{n-1} + 3\log_2 3 \\ b_0 &= 0 \end{aligned} \right\} \text{równanie liniowe.}$$

Zadanie w tej postaci nadaje się już do rozwiązania! Po dokonaniu niewielkich obliczeń możemy otrzymać: $b_n = (2n-1)\log_2 3$, co ostatecznie daje $a_n = 2^{(2n-1)\log_2 3} = 3^{2n-1}$.

Funkcja Ackermanna, czyli coś dla smakoszy

Gdyby małe dzieci znały się odrobinę na informatyce, to rodzice na pewno straszyliby je nie kominiarzem, ale funkcją Ackermanna. Jest to wspaniały przykład ukazujący, jak pozornie niegroźna z wyglądu funkcja rekurencyjna może być kosztowna w użyciu. Spójrzmy na listing:



Listing

A.cpp

```
int A(int n, int p)
{
    if (n==0)
        return 1;
```

```

if ((p==0)&&(n>=1))
    if (n==1)
        return 2;
    else
        return n+2;
if ((p>=1)&&(n>=1))
    return A(A(n-1,p),p-1);
}
int main()
{
    cout << "A(3,4)="<<A(3,4) <<endl;
}

```

Pytanie dotyczące tego programu brzmi: jaki jest powód komunikatu *Stack overflow!* (przepełnienie stosu) podczas próby jego wykonania? Komunikat ten jednoznacznie sugeruje, iż podczas wykonywania programu nastąpiła znaczna liczba wywołań funkcji Ackermanna. Jak znaczna, okaże się już za chwilę.

Pobieżna analiza funkcji A prowadzi do następującego spostrzeżenia:

$$\forall n \geq 1, A(n, 1) = A(A(n-1, 1), 0) = A(n-1, 1) + 2,$$

co daje natychmiast

$$\forall n \geq 1, A(n, 1) = 2n.$$

Analogicznie dla 2 otrzymamy:

$$\forall n \geq 1, A(n, 2) = A(A(n-1, 2), 1) = 2A(n-1, 2),$$

co z kolei pozwala nam napisać, że:

$$\forall n \geq 1, A(n, 2) = 2^n.$$

Z samej definicji funkcji Ackermanna możemy wywnioskować, że:

$$\forall n \geq 1 A(n, 3) = A(A(n-1, 3), 2) = 2^{A(n-1, 3)} \text{ oraz } A(0, 3) = 1.$$

Na bazie tych równań możliwe jest rekurencyjne udowodnienie, że:

$$\forall n \geq 1, A(n, 3) = 2^{\left. 2^{\cdot^{\cdot^{\cdot^2}}}}_n\right\}.$$

Nieco gorsza sytuacja występuje w przypadku $A(n, 4)$, gdzie trudno jest podać wzór ogólny. Proponuję spojrzeć na kilka przykładów liczbowych:

$$\begin{aligned}
 A(1, 4) &= 2. \\
 A(2, 4) &= 2^2 = 4. \\
 A(3, 4) &= 2^{2^{2^2}} = 65536. \\
 A(4, 4) &= 2^{\left. 2^{\cdot^{\cdot^{\cdot^2}}}}_{65536}\right\}.
 \end{aligned}$$

Wyrażenie w formie liczbowej $A(4, 4)$ jest — co może będzie zbyt dyplomatycznym stwierdzeniem — niezbyt oczywiste, nieprawdaż? W przypadku funkcji Ackermanna trudno jest nawet nazwać jej klasę — stwierdzenie, że zachowuje się ona wykładniczo, może zabrzmieć jak kpina!

Złożoność obliczeniowa to nie religia!

Wbrew pozorom złożoność obliczeniowa nie zawsze stanowi istotne kryterium decyzyjne, gdy piszemy programy lub dobieramy algorytmy „z biblioteki”.

Oto kilka sytuacji, gdy możemy nieco odprężyć się w tej kwestii:

- ◆ Zdarza się, że pisany jest algorytm, który ma za zadanie coś obliczyć kilka razy i jego optymalizacja mija się z celem. Taniej będzie spróbować go uruchomić lub ekstrapolować czas wykonania i ocenić, czy jest on do zaakceptowania. Oczywiście taka sytuacja ma miejsce dla funkcji (procedur), których czas wykonania i tak mieści się w wymaganiach wydajnościowych aplikacji.
- ◆ Prostota: czasem optymalne algorytmy są zupełnie niezrozumiałe na pierwszy rzut oka i stopień ich skomplikowania łatwo prowadzi do błędów logicznych, nawet spowodowanych trywialną pomyłką w stylu określenia warunków brzegowych lub logicznych w pętlach.
- ◆ Precyzja, tak potrzebna np. w algorytmach numerycznych, może być ważniejsza niż klasa algorytmu lub jego fragmentu.
- ◆ Notacja dużego O tak naprawdę odgrywa rolę dla takich danych wejściowych, dla których czas wykonania może zbliżać się do bardzo dużych liczb (nie wspominając już o nieskończoności, która jest BARDZO dużą liczbą...!). W praktyce może to oznaczać, że algorytm „kwadratowy” będzie w pewnych konfiguracjach lepszy od „logarymicznego”!

Jako zasadę generalną przyjmijmy zatem zdrowy rozsądek, który uchroni nas przed popadaniem w zbyt akademickie decyzje. Programy wykonują się bowiem nie w teoretycznych środowiskach albo modelach, lecz w prawdziwych komputerach. Co z tego, że przyspieszymy czas realizacji wyliczeń finansowych 100-krotnie, np. ze 100 ms do 1 ms, gdy na koniec czas zapisu wyniku do bazy danych i tak wyniesie 2 – 3 s?

Techniki optymalizacji programów

Załóżmy, że stworzyliśmy program i wydaje nam się, że działa on zbyt wolno¹¹. Jak możemy się zorientować, gdzie tkwi problem wydajnościowy? Podstawowym problemem przy optymalizacji jest zlokalizowanie tego miejsca, które warto optymalizować. Jeśli nasz program jest w pełni funkcjonalny i dysponujemy danymi, które są reprezentatywne (np. baza danych wypełniona tysiącami rekordów, a nie kilkoma fałszywymi wpisami), to możemy przystąpić do pomiaru czasów realizacji modułów lub procedur. To, gdzie mierzymy, zależy już od architektury i trzeba postępować ze świadomością, jakie są przepływy danych i wąskie gardła.

Jednym z prostszych sposobów jest wykorzystanie metod śledzenia czasów realizacji procedur poprzez wbudowanie do nich dodatkowych liczników. Ważne jest jednak, aby te liczniki nie obciążały samych algorytmów, zatem powinny one być zrealizowane z wykorzystaniem mechanizmów systemowych (czas odczytywany z systemu, a nie liczony przez program).

Gdy już posiadamy wyniki naszych pomiarów, to postępujemy wg zasady Pareto, czyli koncentrujemy naszą uwagę na najbardziej obciążonych lub najczęściej wywoływanych fragmentach kodu (np. pętlach, procedurach).

¹¹ W tym punkcie nie interesuje mnie kwestia *rozmiaru* kodu, gdyż w obecnych czasach to kryterium trochę straciło na znaczeniu.

Możliwe są teraz dwa podejścia:

- ♦ Analizujemy algorytmy według znanych nam reguł (ten rozdział!) i tutaj szukamy przysłowiowej „dziury w całym”.
- ♦ Optymalizujemy kod, wyszukując w nim fragmenty, w których można dokonać uproszczeń, np. przez zastosowanie jednej ze sztuczek:
 - ♦ Wykorzystanie pamięci podręcznej (ang. *cache*) : liczymy coś jeden raz i zapisujemy do pamięci wykorzystywanej przez cały czas życia programu, oferując, np. poprzez zmienne statyczne, widzialność wyników w całym programie lub modułach.
 - ♦ Eliminujemy operacje czasochłonne, np. bazodanowe, wykonując je wówczas, gdy jest to naprawdę potrzebne.
 - ♦ Zastępujemy kosztowne czasowo instrukcje mnożenia lub potęgowania poprzez szybkie operacje przesuwania bitów w lewo lub w prawo (np. $n \ll 3$ oznacza to samo co $n \cdot 2^3 = n \cdot 8$). Oczywiście dobre kompilatory tak naprawdę robią to samodzielnie, jeśli się „domyślą”, gdzie to zrobić i... będą miały włączone odpowiednie opcje optymalizacji!
 - ♦ Przerzucamy część logiki aplikacji, która przetwarza dane na serwer bazodanowy (procedury wbudowane).
 - ♦ Stosujemy wielowątkowość.
 - ♦ Wykorzystujemy znane nam cechy architektury komputera, np. zmuszamy kompilator do użycia konkretnych, zoptymalizowanych instrukcji konkretnego procesora.
 - ♦ Wykorzystujemy cechy języka programowania (np. skuteczne, choć utrudniające analizę, instrukcje `break` lub `goto` w C++).

Technik optymalizacji kodu jest naprawdę wiele i piszę o nich w tym miejscu tylko po to, aby zasygnalizować obszar, którym można się zainteresować podczas poważnego programowania!

Zadania

Zadanie 1.

Proszę rozważyć problem prawdziwości lub fałszu poniższych równań:

- ♦ $T(n^3) \in O(n^3)$;
- ♦ $T(n^2) \in O(n^3)$;
- ♦ $T(2^{n+1}) \in O(2^n)$;
- ♦ $T((n+1)!) \in O(n!)$;
- ♦ $T(n) \in O(n) \Rightarrow \{T(n)\}^2 \in O(n^2)$;
- ♦ Twój własny przykład?

Zadanie 2.

Jednym z analizowanych już wcześniej przykładów był tzw. ciąg Fibonacciego. Funkcja obliczająca elementy tego ciągu jest nieskomplikowana:

```
unsigned long int fib(int x)
{
    if (x < 2)
        return x;
```

```

else
    return fib(x-1)+fib(x-2);
}

```

Proszę określić, jakiej klasy jest to funkcja.

Zadanie 3.

Proszę przeanalizować taki ze swoich programów, w którym jest dużo wszelkiego rodzaju zagnieżdżonych pętli i tego rodzaju skomplikowanych konstrukcji. Czy nie dałoby się go zoptymalizować w jakiś sposób?

Przykładowo często się zdarza, że w pętlach są inicjowane pewne zmienne i to za każdym przebiegiem pętli, choć w praktyce wystarczyłoby je zainicjować tylko raz. W takim przypadku instrukcję przypisania przenosi się przed pętlę. Skracza to czas wykonywania się pętli. Podobnie, odpowiednio układając kolejność pewnych obliczeń, można wykorzystywać częściowe wyniki, będące rezultatem pewnego bloku instrukcji, w dalszych blokach — oczywiście pod warunkiem że nie zostały zamazane przez pozostałe fragmenty programu. Zadanie polega na obliczeniu złożoności praktycznej naszego programu *przed i po* optymalizacji i przekonaniu się na własne oczy o osiągniętym (ewentualnie) przyspieszeniu.

Zadanie 4.

Proszę rozwiązać następujące równanie rekurencyjne:

$$u_n = u_{n-1} - u_n \cdot u_{n-1} \text{ (dla } n \geq 1),$$

$$u_0 = 1.$$

Rozwiązania i wskazówki do zadań

Zadanie 2.

Równanie rekurencyjne ma postać:

$$T(0) = 0,$$

$$T(1) = 1,$$

$$T(n) = T(n-1) + T(n-2).$$

Mimo dość skomplikowanej postaci w zadaniu tym nie kryje się żadna pułapka i rozwiązuje „się” ono całkiem przyjemnie. Spójrzmy na szkic rozwiązania:

ETAP 1. Poszukiwanie równania charakterystycznego:

Z postaci ogólnej SRL: $T(n) = T(n-1) + T(n-2)$ wynika, że $RC = x^2 - x - 1$.

ETAP 2. Pierwiastki równania charakterystycznego:

Po prostych wyliczeniach otrzymujemy dwa pierwiastki tego równania kwadratowego:

$$RC = x^2 - x - 1 = (x - r_1)(x - r_2), \text{ gdzie: } r_1 = \frac{1 + \sqrt{5}}{2} \text{ i } r_2 = \frac{1 - \sqrt{5}}{2}$$

ETAP 3. Równanie ogólne:

Z teorii wyłożonej wcześniej wynika, że równanie ogólne ma postać $RO = A r_1^n + B r_2^n$
— zostawmy je chwilowo w tej formie.

ETAP 4. Poszukiwanie równania szczególnego:

Wiemy, że $u(n, m) = 0$, a zatem $RS = 0$.

ETAP 5. Poszukiwanie ostatecznego rozwiązania:

Poszukiwanym rozwiązaniem jest suma RO i RS :

$$T(n) = RO + RS = Ar_1^n + Br_2^n.$$

Pozostają nam do odnalezienia tajemnicze stałe A i B . Do tego celu posłużymy się warunkami początkowymi (tzn. przypadkami elementarnymi, aby pozostać w zgodzie z terminologią z rozdziału 2.) układu równań rekurencyjnych ($T(0) = 0$ i $T(1) = 1$). Po wykonaniu podstawienia otrzymamy:

$$0 = A + B,$$

$$1 = Ar_1 + Br_2.$$

Jest to prosty układ dwóch równań z dwoma niewiadomymi (A i B). Jego wyliczenie powinno nam dać poszukiwany wynik. Skończenie tego zadania pozostawiam Czytelnikowi.

Zadanie 4.

Oto szkic rozwiązania:

Założmy, że $u_n \neq 0$, co pozwoli nam podzielić równania przez $u_n u_{n-1}$:

$$\frac{1}{u_{n-1}} = \frac{1}{u_n} - 1.$$

Podstawmy wówczas $v_n = \frac{1}{u_n} - 1$, co da nam bardzo proste równanie, z którym już mieliśmy prawo wcześniej się spotkać:

$$v_n = v_{n-1} + 1,$$

$$v_0 = 1.$$

Jego rozwiązaniem jest oczywiście $v_n = n + 1$. Po powrocie do pierwotnej dziedziny otrzymamy dość zaskakujący wynik: $u_n = \frac{1}{n+1}$.

Rozdział 4.

Algorytmy sortowania

Tematem tego rozdziału będzie opis kilku bardziej znanych metod sortowania danych. O użyteczności tych zagadnień nie trzeba chyba przekonywać; każdy programista prędzej czy później musi mieć do czynienia z tym zagadnieniem. Algorytmy sortowania są znakomitym sposobem na poznanie algorytmiki i zagadnień związanych ze złożonością obliczeniową oprogramowania.

Pokazane w tym rozdziale opisy algorytmów sortowania będą dotyczyły głównie tzw. *sortowania wewnętrznego*, używającego tylko pamięci operacyjnej komputera. W tym wydaniu książki zdecydowałem się również omówić problematykę tzw. *sortowania zewnętrznego*. Sortowanie zewnętrzne dotyczy sytuacji, z którą większość Czytelników być może nigdy się nie zetknie w praktyce programowania: ilość danych do posortowania jest tak ogromna, że niemożliwe jest ich umieszczenie w pamięci operacyjnej i użycie jednej z metod sortowania wewnętrznego. Mimo to zagadnienia omawiane w kontekście sortowania zewnętrznego są na tyle ciekawe od strony praktycznej, że już samo pobieżne zapoznanie się z nimi może wzbogacić naszą wiedzę o programowaniu o zupełnie nowe obszary.

W praktyce jednak pamięć komputerowa (tzw. RAM) i dyski twarde systematycznie tanieją i do większości zagadnień sortowanie wewnętrzne jest wystarczające¹. Od kilku lat coraz głośniej zaczyna być o bazach danych całkowicie rezydujących w pamięci (dla zwiększenia sprawności), rzecz niewyobrażalna kilka lat temu w praktyce.

Potrzeba sortowania danych jest związana pośrednio z typowo ludzką chęcią gromadzenia (chcemy mieć dużo) i porządkowania (a jak już dużo posiadamy, to zaczyna się problem z kontrolowaniem, co i gdzie mamy). A sortowanie rozwiązuje w dużym stopniu ten ostatni problem.

Istotnym problemem w dziedzinie sortowania danych jest ogromna różnorodność algorytmów wykonujących to zadanie. Początkujący programista często nie jest w stanie samodzielnie dokonać wyboru algorytmu sortowania najodpowiedniejszego do konkretnego zadania. Jedno z możliwych podejść do tematu polegałoby zatem na krótkim opisaniu każdego algorytmu, wskazaniu jego wad i zalet oraz podaniu swoistego rankingu jakości. Wydaje mi się jednak, że tego typu prezentacja nie spełniłaby dobrze swojego zadania informacyjnego, a jedynie sporo zamieszała w głowie Czytelnika. Skąd to przekonanie? Z pobieżnych obserwacji wynika, że programiści raczej używają dobrze sprawdzonych klasycznych rozwiązań, takich jak np. sortowanie przez wstawianie, sortowanie bąbelkowe, sortowanie szybkie niż równie dobrych (jeśli nie lepszych) rozwiązań, które służą głównie jako tematy artykułów czy też przyczynki do badań porównawczych z dziedziny efektywności algorytmów.

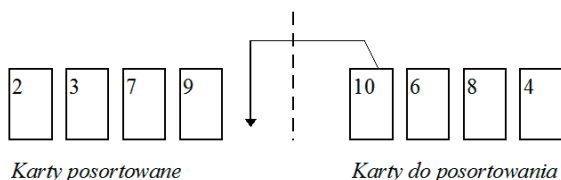
¹ Mój pierwszy prywatny komputer osobisty typu IBM PC XT miał 1 MB RAM-u i dysk twardy 20 MB!

Aby nie powiększać entropii wszechświata, skoncentrujemy się na szczegółowym opisie tylko kilku dobrze znanych, wręcz wzorcowych metod. Będą to algorytmy charakteryzujące się różnym stopniem trudności (rozpatrywanej w kontekście wysiłku poświęconego na pełne zrozumienie idei) i mające odmienne parametry czasowe. Wybór tych właśnie, a nie innych metod jest dość arbitralny i pozostaje mi tylko żywić nadzieję, że zaspokoi on potrzeby jak największej grupy Czytelników.

Sortowanie przez wstawianie, algorytm klasy $O(N^2)$

Metoda sortowania przez wstawianie jest używana bezwiednie przez większość graczy podczas układania otrzymanych w rozdaniu kart. Rysunek 4.1 przedstawia sytuację widzianą z punktu widzenia gracza będącego w trakcie tasowania kart, które otrzymał on w dość podłym rozdaniu:

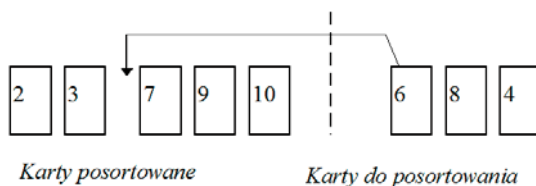
Rysunek 4.1.
Sortowanie przez
wstawianie (1)



Idea tego algorytmu opiera się na następującym niezmienniku: w danym momencie trzymamy w ręku karty posortowane² oraz karty pozostałe do posortowania. W celu kontynuowania procesu sortowania bierzemy pierwszą z brzegu kartę ze sterty nieposortowanej i wstawiamy ją na właściwe miejsce w pakiecie już wcześniej posortowanym.

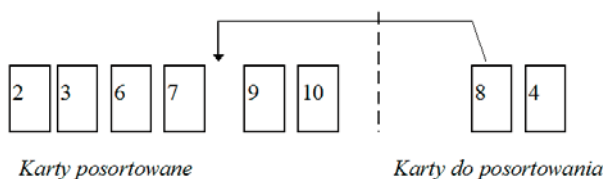
Popatrzmy na dwa kolejne etapy sortowania. Rysunek 4.2 obrazuje sytuację już po wstawieniu karty 10 na właściwe miejsce, kolejną kartą do wstawienia będzie 6.

Rysunek 4.2.
Sortowanie przez
wstawianie (2)



Tuż po poprawnym ułożeniu szóstki otrzymujemy układ z rysunku 4.3.

Rysunek 4.3.
Sortowanie przez
wstawianie (3)



Widać już, że algorytm jest nużąco jednostajny i... raczej dość wolny.

² Na samym początku algorytmu możemy mieć puste ręce, ale dla zasady twierdzimy wówczas, że trzymamy w nich zerową liczbę kart.

Ciekawa odmiana tego algorytmu realizuje *wstawianie* poprzez przesuwanie zawartości tablicy w prawo o jedno miejsce w celu wytworzenia odpowiedniej luki, w której następnie umieszcza ów element. Skąd mamy wiedzieć, czy kontynuować przesuwanie zawartości tablicy podczas poszukiwania luki? Podjęcie decyzji umożliwi nam sprawdzanie warunku sortowania (sortowanie w kierunku wartości malejących, rosnących czy też wg innych kryteriów).

Popatrzmy na tekst programu³:



Listing

insert.cpp

```
void InsertSort(int *tab)
{
    for(int i=1; i<n; i++)
    {
        int j=i; //fragment [0..., i-1] jest już posortowany
        int temp=tab[j];
        while ((j>0) && (tab[j-1]>temp))
        {
            tab[j]=tab[j-1];
            j--;
        }
        tab[j]=temp;
    }
}
```

Algorytm sortowania przez wstawianie charakteryzuje się dość wysokim kosztem: jest on bowiem klasy $O(N^2)$, co eliminuje go w praktyce z sortowania dużych tablic. Niemniej jeśli nie zależy nam na szybkości sortowania, a potrzebujemy algorytmu na tyle krótkiego, by się w nim na pewno nie pomylić — to wówczas jest on idealny w swojej niepodważalnej prostocie.



Uwaga

Dla prostoty przykładów będziemy analizować jedynie sortowanie tablic liczb całkowitych. W rzeczywistości sortowaniu podlegają najczęściej tablice lub listy rekordów; kryterium sortowania odnosi się wówczas do jednego z pól rekordu. (Patrz również rozdział 5. i zagadnienie sortowania list).

Sortowanie bąbelkowe, algorytm klasy $O(N^2)$

Podobnie jak sortowanie przez wstawianie, algorytm sortowania bąbelkowego charakteryzuje się olbrzymią prostotą zapisu. Intrygująca jego nazwa wzięła się z analogii pęcherzyków powietrza ulatujących w górę tuby wypełnionej wodą — o ile postawioną pionowo tablicę potraktować jako pojemnik z wodą, a liczby jako pęcherzyki powietrza. Najszybciej ulatują do góry „bąbelki” najlżejsze — liczby o najmniejszej wartości (przyjmując oczywiście sortowanie w kierunku wartości niemalejących). Oto pełny tekst programu:



Listing

bubble.cpp

```
void bubble(int *tab)
{
    for (int i=1; i<n; i++)
        for (int j=n-1; j>=i; j--)
```

³ W tym i dalszych przykładach zakładam, że w kodzie jest umieszczona instrukcja `const int n=jakaś wartość` określająca rozmiar tablicy przeznaczonej do posortowania — pomijam to w wydrukowanych listingach.

```

    if (tab[j]<tab[j-1])
    { //zamiana
      int tmp=tab[j-1];
      tab[j-1]=tab[j];
      tab[j]=tmp;
    }
  }

```

Przeanalizujemy dokładnie sortowanie bąbelkowe pewnej 7-elementowej tablicy. Na rysunku 4.4 element zaciemniany jest tym, który w pojedynczym przebiegu głównej pętli programu „uleciał” do góry jako najlżejszy. Tablica jest przemiatana sukcesywnie od dołu do góry (pętla zmiennej *i*). Analizowane są zawsze dwa sąsiadujące ze sobą elementy (pętla zmiennej *j*): jeśli nie są one uporządkowane (u góry jest element „cięższy”), to następuje ich zamiana. W trakcie pierwszego przebiegu na pierwszą pozycję tablicy (indeks 0) ulatuje element „najlżejszy”, w trakcie drugiego przebiegu drugi najlżejszy wędruje na drugą pozycję tablicy (indeks 1) i tak dalej, aż do ostatecznego posortowania tablicy. Strefa pracy algorytmu zmniejsza się zatem o 1 w kolejnym przejściu dużej pętli — analizowanie za każdym razem całej tablicy byłoby oczywistym marnotrawstwem!

Rysunek 4.4.
Sortowanie „bąbelkowe”

etapy sortowania

	0	1	2	3	4	5	6
0	40	2	2	2	2	2	2
1	2	40	4	4	4	4	4
2	39	4	40	6	6	6	6
3	6	39	6	40	18	18	18
4	18	6	39	18	40	20	20
5	4	18	18	39	20	40	39
6	20	20	20	20	39	39	40

indeks w tablicy

Nawet dość pobieżna analiza prowadzi do kilku negatywnych uwag na temat samego algorytmu:

- ◆ Dość często zdarzają się „puste przebiegi” (nie jest dokonywana żadna wymiana, bowiem elementy są już posortowane).
- ◆ *Algorytm* jest bardzo wrażliwy na konfigurację danych. Oto przykład dwóch niewiele różniących się tablic, z których pierwsza wymaga jednej zamiany sąsiadujących ze sobą elementów, a druga będzie wymagać ich aż sześciu:
 - ◆ **wersja 1:** 4 2 6 18 20 39 40,
 - ◆ **wersja 2:** 4 6 18 20 39 40 2.

Istnieje kilka możliwości poprawy jakości tego algorytmu — nie prowadzą one co prawda do zmiany jego klasy (w dalszym ciągu mamy do czynienia z $O(N^2)$), ale mimo to dość znacznie go przyspieszają. Ulepszenia te polegają odpowiednio na:

- ◆ zapamiętywaniu indeksu ostatniej zamiany (walka z „pustymi przebiegami”);
- ◆ przełączaniu kierunków przeglądania tablicy (walka z niekorzystnymi konfiguracjami danych).

Tak poprawiony algorytm sortowania bąbelkowego nazwiemy sobie po polsku *sortowaniem przez wytrząsanie* (ang. *shaker-sort*). Jego pełny tekst jest zamieszczony poniżej, lecz tym razem już bez tak dokładnej analizy, jak poprzednio:

**shaker.cpp**

```

void ShakerSort(int *tab)
{
    int left=1, right=n-1, k=n-1;
    do
    {
        for(int j=right; j>=left; j--)
            if(tab[j-1]>tab[j])
            {
                swap(tab[j-1],tab[j]);
                k=j;
            }
        left=k+1;
        for(int j=left; j<=right; j++)
            if(tab[j-1]>tab[j])
            {
                swap(tab[j-1],tab[j]);
                k=j;
            }
        right=k-1;
    }
    while (left<=right);
}

```

Quicksort, algorytm klasy $O(N \log N)$

Jest to słynny algorytm⁴, zwany również po polsku sortowaniem szybkim. Należy on do tych rozwiązań, w których poprzez odpowiednią dekompozycję osiągnięty został znaczny zysk szybkości sortowania. Procedura sortowania dzieli się na dwie zasadnicze części: część służącą do właściwego sortowania, która nie robi w zasadzie nic oprócz wywoływania samej siebie, oraz procedury rozdzielania elementów tablicy względem wartości pewnej komórki tablicy służącej za oś (ang. *pivot*) podziału. Proces sortowania jest dokonywany przez tę właśnie procedurę, natomiast rekurencja zapewnia poskładanie wyników częściowych i w konsekwencji posortowanie całej tablicy.

Jak dokładnie działa procedura podziału? Otóż w pierwszym momencie odczytuje się wartość elementu osiowego P , którym zazwyczaj jest po prostu pierwszy element analizowanego fragmentu tablicy. Tenże fragment tablicy jest następnie dzielony, tak aby elementy mniejsze od ' P ' znalazły się po lewej stronie, a większe — po prawej⁵. Ten nowy układ jest symbolicznie przedstawiony na rysunku 4.5.

Rysunek 4.5.

Podział tablicy
w metodzie Quicksort

$< P$	P	$\geq P$
-------	-----	----------

Kolejnym etapem jest zaaplikowanie procedury Quicksort na lewym i prawym fragmencie tablicy, czego efektem będzie jej posortowanie. To wszystko!

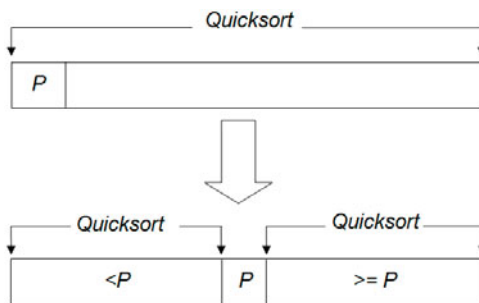
Na rysunku 4.6 są przedstawione symbolicznie dwa główne etapy sortowania metodą Quicksort (P oznacza tradycyjnie komórkę tablicy służącą za oś).

Jest chyba dość oczywiste, że wywołania rekurencyjne zatrzymają się w momencie, gdy rozmiar fragmentu tablicy wynosi 1 — nie ma już bowiem czego sortować.

⁴ Patrz C.A.R. Hoare — „Quicksort” w Computer Journal, 5, 1(1962).

⁵ Elementy tablicy są fizycznie przemieszczane, jeśli zachodzi potrzeba.

Rysunek 4.6.
Zasada działania
procedury *Quicksort*



Przedstawiona powyżej metoda sortowania charakteryzuje się olbrzymią prostotą, wyrażoną najdoskonalej przez zwięzły zapis samej procedury (prawdziwy kod znajduje się nieco dalej):

```
void Quicksort(int *tab, int left, int right)
{
    if (left < right)
    {
        int m;
        // Podziel tablicę względem elementu osiowego 'P'
        // ('P' może być np. pierwszym elementem tablicy, czyli tab[left])
        // Pod koniec podziału element 'P' zostanie przeniesiony do komórki o numerze 'm'
        Quicksort(tab, left, m-1);
        Quicksort(tab, m+1, right);
    }
}
```

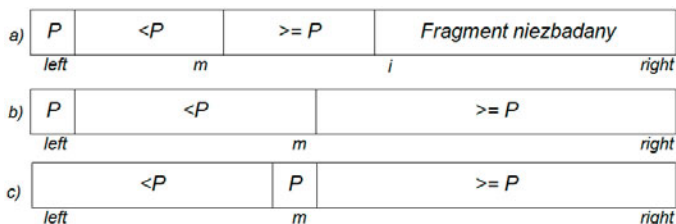
Jak najprościej zrealizować fragment procedury sprytnie ukryty za komentarzem? Jego działanie jest przecież najistotniejszą częścią algorytmu, a my jak dotąd traktowaliśmy go dość ogólnikowo. Takie postępowanie wynikało z dość prozaicznej przyczyny — implementacji algorytmu *Quicksort* jest mnóstwo i różnią się one właśnie realizacją procedury podziału tablicy względem wartości osi.

Oszczędzając Czytelnikowi dylematów dotyczących wyboru właściwej wersji, zaprezentujemy poniżej — zgodnie z prawdziwymi zasadami współczesnej demokracji — tę najwłaściwszą.

Kryteriami wyboru były: piękno, szybkość i prostota — tych cech można niewątpliwie doszukać się w rozwiązaniu przedstawionym w [Ben92].

Pomysł opiera się na zachowaniu dość prostego niezmiennika w aktualnie rozdzielanym fragmencie tablicy (patrz rysunek 4.7).

Rysunek 4.7.
Budowa niezmiennika
dla algorytmu *Quicksort*



Oznaczenia:

left — lewy skrajny indeks aktualnego fragmentu tablicy;

right — prawy skrajny indeks aktualnego fragmentu tablicy;

P — wartość osiowa (zazwyczaj będzie to `tab[left]`);

i — indeks „przechadzający się” po indeksach tablicy od left do right;

m — poszukiwany indeks komórki tablicy, w której umieścimy element osiowy.

Przemieszczanie się po tablicy służy do poukładania jej elementów w taki sposób, aby po lewej stronie m znajdowały się wartości mniejsze od elementu osiowego, po prawej zaś — większe lub równe (rysunek 4.7a). W tym celu podczas przemieszczania indeksu i sprawdzamy prawdziwość warunku $\text{tab}[i] > P$. Jeśli jest on fałszywy, to poprzez inkrementację i i wymianę wartości $\text{tab}[m]$ i $\text{tab}[i]$ przywracamy porządek. Gdy zakończymy ostatecznie przeglądanie tablicy (rysunek 4.7b) w pogoni za komórkami, które nie chciały się podporządkować niezmiennikowi, zamiana $\text{tab}[\text{left}]$ i $\text{tab}[m]$ doprowadzi do oczekiwanej sytuacji (rysunek 4.7c).

Nic już teraz nie stoi na przeszkodzie, aby zaproponować ostateczną wersję procedury *Quicksort*. Omówione wcześniej etapy działania algorytmu zostały połączone w jedną procedurę:



quick.cpp

```
void qsort(int *tab, int left, int right)
{
    if (left < right)
    {
        int m=left;
        for (int i=left+1; i<=right; i++)
            if (tab[i]<tab[left])
                swap(tab[++m], tab[i]);
        swap(tab[left], tab[m]);
        qsort(tab, left, m-1);
        qsort(tab, m+1, right);
    }
}
```

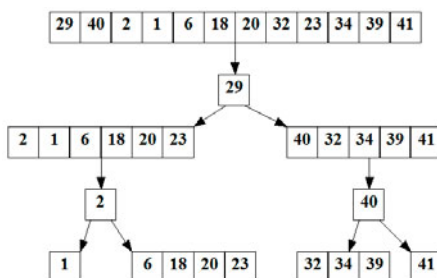
W celu dobrego zrozumienia działania algorytmu spróbujmy posortować za jego pomocą ręcznie małą tablicę, np. zawierającą następujące liczby:

29, 40, 2, 1, 6, 18, 20, 32, 23, 34, 39, 41.

Rysunek 4.8 przedstawia efekt działania tych egzemplarzy procedury *Quicksort*, które faktycznie coś robią.

Rysunek 4.8.

Sortowanie metodą
Quicksort na przykładzie



Widać wyraźnie, że przechodząc od skrajnie lewej gałęzi drzewa do skrajnie prawej i odwiedzając w pierwszej kolejności lewe jego odnogi, przechadzamy się w istocie po posortowanej tablicy! W naszym programie taki spacer realizują wywołania rekurencyjne procedury *qsort*. Algorytm *Quicksort* stanowi dobry przykład techniki programowania zwanej „dziel i zwyciężaj”, która zostanie dokładnie omówiona w rozdziale 9.

Tutaj zapowiem jedynie, że chodzi o taką dekompozycję problemu, aby osiągnąć zysk czasowy wykonywania programu (a przy okazji uproszczenie rozwiązywanego zadania). Algorytm *Quicksort* wzorowo spełnia oba te wymagania!

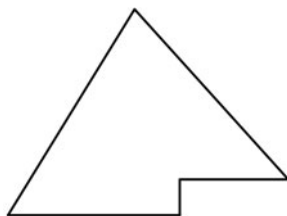
Heap Sort — sortowanie przez kopcowanie

Algorytm sortowania przez kopcowanie (zaraz wyjaśnię ten przedziwny termin!) jest klasy $O(N \log N)$. Używa on ciekawej struktury danych zwanej kopcem, której charakterystykę zaraz omówię. Zaletą tego algorytmu jest właśnie ta struktura danych, która działa jednocześnie jak kolejka priorytetowa (patrz rozdział 5.).

Przedstawmy zatem nową strukturę danych, trochę jako element kolejnego rozdziału poświęconego właśnie strukturom danych. Gdyby Czytelnik miał kłopoty terminologiczne, zachęcam co najmniej do pobieżnego przejrzania kolejnego rozdziału, gdzie struktura ta jest omówiona dokładniej, zaprezentowana jest także kolejna wersja samego algorytmu w notacji obiektowej. W tym rozdziale zaprezentuję bardziej zwięzłą wersję tego samego algorytmu, gdyż chciałbym skoncentrować się bardziej na algorytmie sortowania niż na strukturze danych.

Kopiec czy też, jak niektórzy wolą, sterta jest drzewem binarnym zawierającym liczby lub dowolne inne elementy dającego się porządkować zbioru. Cechą kopca jest kształt przedstawiony na rysunku 4.9 — jest to piramidka, w której podstawa z prawej strony jest „nadgryziona”.

Rysunek 4.9.
Kopiec i jego
własności (1)



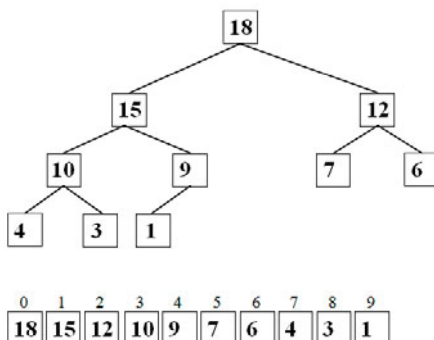
Oprócz kształtu kluczową cechą kopca jest uporządkowanie — każda wartość w węźle poniżej danego węzła jest od niej mniejsza. W notacji tablicowej warunek ten można zapisać jako: $T[ojciec(i)] \geq T[i]$.

Zawartość kopca łatwo reprezentujemy w zwykłej tablicy T wg następujących reguł:

- ◆ Wierzchołek kopca wstaw do $T[0]$.
- ◆ Dla dowolnego (jeśli istnieje) węzła w $T[i]$ jego lewy syn jest w $T[2i+1]$, prawy w $T[2i+2]$.

Przykład takiej struktury znajdziemy na rysunku 4.10.

Rysunek 4.10.
Kopiec i jego
własności (2)



Algorytm sortowania przez kopcowanie zapisuje się następująco:

1. Ułóż dane w kopcie — niech znajdą się one w tablicy o rozmiarze `rozmiar`.
2. Usuń wierzchołek z kopca (jest to największy element) poprzez zamianę go z ostatnim liściem drzewa (rozmiar--).

3. Przywróć własność kopca dla pozostałej (oprócz usuniętego elementu) części kopca.
4. Idź do pkt. 2.

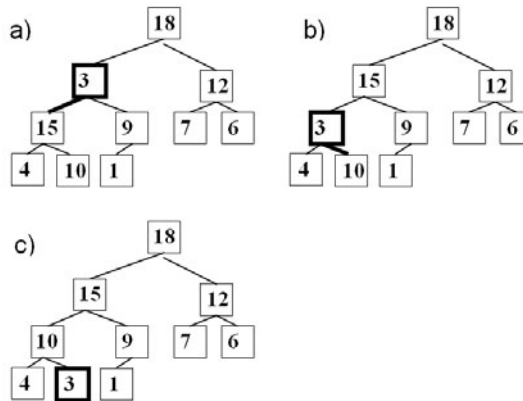
Procedura z pkt. 3. jest zaimplementowana w następujący sposób:

1. Jeśli wierzchołek jest większy od obojga dzieci, wyjdź.
2. Zamień wierzchołek z większym dzieckiem.
3. Przywróć własność kopca w tej części kopca, w której nastąpiła zamiana.

Rysunek 4.11 pokazuje efekt działania procedury z pkt. 3. zaaplikowanej na pogrubionym węźle, który zaburza warunek $T[ojciec(i)] \geq T[i]$. W kolejnych krokach a), b) i c) poprzez zamianę wierzchołka z największym dzieckiem doprowadzamy do uporządkowania struktury.

Rysunek 4.11.

Przywracanie własności kopca na przykładzie



Prosta realizacja algorytmu sortowania przez kopcowanie przedstawiona jest poniżej.



heap.cpp

```

void przywroc(int T[], int k, int n)
{
    int i, j;

    i = T[k-1];
    while (k <= n/2)
    {
        j = 2*k;
        if ((j < n) && (T[j-1] < T[j])) j++;
        if (i >= T[j-1])
            break;
        else
        {
            T[k-1] = T[j-1];
            k = j;
        }
    }
    T[k-1] = i;
}

// sortowanie tablicy T
void heapsort(int T[], int n)
{
    int k, swap;
    for(k = n/2; k > 0; k--) przywroc(T, k, n);
}
  
```

```

do
{
    swap=T[0];
    T[0]=T[n-1];
    T[n-1]=swap;
    n--;
    przywroc(T, 1, n);
} while (n > 1);
}

int main()
{
    int i, T[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
    for (i=0; i<14; i++)
        cout << " " << T[i];

    cout << endl;

    heapsort(T,14);

    for (i=0; i<14; i++)
        cout << " " << T[i];
}

```

Scalanie zbiorów posortowanych

Prosty algorytm omówiony w tym punkcie zajmuje się sortowaniem dość szczególnego przypadku: tablice wejściowe $T_1[N]$ i $T_2[M]$ są uporządkowane i naszym zadaniem jest utworzenie tablicy $T_3[N+M]$, która także będzie uporządkowana.

Sam algorytm wydaje się prosty: należy pobierać dane z obu tablic, aż do ich „wyczerpania”, przenosząc do tablicy wynikowej zawsze element najmniejszy (jeśli interesuje nas taki porządek danych). Algorytm „nie traci czasu” na odszukiwanie elementów do porównania, gdyż zakładamy, że dane wejściowe są faktycznie posortowane. Jest to o tyle ważne, iż w procedurze, którą pokażę dalej, nie jest w żadnym miejscu dokonywana weryfikacja tego założenia!

Popatrzmy na sam program:



Listing

scalaj.cpp

```

void scalaj(int T1[], int T2[], int T3[])
{ // T1, T2 — tablice wejściowe o rozmiarze M i M, T3 — tablica wynikowa
    for (int i=0, j=0, k=0; k < N+M; k++)
    {
        if (i==N) // osiągnięto koniec zbioru T1, kopiujemy resztę
        {
            T3[k]=T2[j++]; continue;
        }
        if (j==M) // osiągnięto koniec zbioru T2, kopiujemy resztę
        {
            T3[k]=T1[i++]; continue;
        }
        if (T1[i]<T2[j])
            T3[k]=T1[i++];
        else
            T3[k]=T2[j++];
    }
}

```

Zauważmy, że dostęp do danych jest sekwencyjny, co przypomina nieco... pliki, zazwyczaj odczytywane w taki właśnie sposób!

W ramach drobnego ćwiczenia proponuję przerobić kod procedury operującej na tablicach na jego odpowiednik plikowy. Nie jest to aż takie trywialne, gdyż pełna analogia tablicowa nie może być zastosowana (zakładamy nieznane nam rozmiary plików wejściowych). Jeśli masz kłopoty z używaniem funkcji operujących na plikach, spójrz do dodatku A, gdzie podany jest program odczytujący dane z pliku.

Sortowanie przez scalanie

Kolejną metodą sortowania, którą omówimy, jest — podobnie jak *Quicksort* — metoda rekurencyjna z gatunku „dziel i zwyciężaj” (omówiona szerzej w rozdziale 9.). Algorytm jest prosty:

- ♦ Dzielimy n -elementowy ciąg na dwa podciągi po $n/2$ elementów każdy.
- ♦ Sortujemy metodą przez scalanie każdy z podciągów osobno.
- ♦ Łączymy posortowane podciągi w jeden posortowany ciąg.

Oczywiście cała zabawa polega w tej metodzie na zgrabnym napisaniu procedury scalającej. Poniżej zamieściłem jedną z możliwych implementacji takiej procedury:



Listing

merge.cpp

```
const int N = 10;
void Scalaj(int T[], int p, int mid, int k)
// p — początek, k — koniec, mid — środek
// procedura łączy 2 posortowane tablice T[p...mid] i T[mid+1...k]
{
    int T2[N]; // tablica pomocnicza
    int p1 = p, k1 = mid; // podtablica 1
    int p2 = mid+1, k2 = k; // podtablica 2
    // aż do wyczerpania tablic dokonaj scalenia przy pomocy tablicy pomocniczej
    int i = p1;
    while((p1 <= k1) && (p2 <= k2))
    {
        if(T[p1] < T[p2])
        {
            T2[i] = T[p1];
            p1++;
        }
        else
        {
            T2[i] = T[p2];
            p2++;
        }
        i++;
    }

    while(p1 <= k1)
    {
        T2[i] = T[p1];
        p1++;
        i++;
    }
    while(p2 <= k2)
    {
        T2[i] = T[p2];
        p2++;
        i++;
    }
}
```

```
//skopiuj z tablicy tymczasowej do oryginalnej
for(i = p; i <= k; i++)
    T[i] = T2[i];
}
```

Sama procedura sortująca jest oczywiście powtórzeniem podanego wyżej schematu rekurencyjnego:

```
void MergeSort(int T[], int p, int k)
{
    if(p < k)
    {
        int mid = (p + k) / 2; //środek
        MergeSort(T, p, mid); //sortuj lewą połowę
        MergeSort(T, mid+1, k); //sortuj prawą połowę
        Scalaj(T, p, mid, k); //scalaj
    }
}

int main()
{
    int T[N] = {4, 6, 4, 12, -3, 6, -6, 1, 8, '2'};
    cout << "Przed sortowaniem:\n";
    for(int x=0; x<N; x++) cout << T[x] << " "; cout << endl;
    MergeSort(T, 0, N-1);
    cout << "Po sortowaniu:\n";
    for(int x=0; x<N; x++) cout << T[x] << " "; cout << endl;
}
```

Wyniki:

```
Przed sortowaniem:
4 6 4 12 -3 6 -6 1 8 50
Po sortowaniu:
-6 -3 1 4 4 6 6 8 12 50
```

Sortowanie zewnętrzne

Sortowanie zewnętrzne występuje, gdy rozmiar danych znacznie przekracza pojemność pamięci. Czy taka sytuacja może wystąpić w praktyce? Owszem, ale warto od razu zwrócić uwagę, że systemy informatyczne, w których można napotkać takie przypadki, są dość niszowe: np. wielkie bankowe systemy billingowe lub bankowe systemy finansowe. Czy jednak bank lub firma telekomunikacyjna użyje wiedzy opisanej w tym podrozdziale? Odpowiedź jest przewrotna: NIE.

Oto powody:

- ◆ Wielkie systemy informatyczne używają od dawna architektury n -warstwowej, której elementem jest serwer (lub farma serwerów) bazy danych. W bazie danych ciężar sortowania może zostać przerzucony na motor bazy danych, np. obok tabel danych zakłada się tzw. indeksy, które mogą zostać użyte do przeglądania danych w wersji posortowanej.
- ◆ Niektóre systemy (np. bankowe, oparte na komputerach mainframe) stosują COBOL, język programowania wykorzystywany do tworzenia dużych aplikacji biznesowych, opracowany w latach 60-tych, który pozwala na sprawne operowanie na plikach, także sortowanie.

Mimo to teoretyczny model sortowania zewnętrznego jest dość ciekawy i warto rzucić na niego okiem, choćby tylko w celu zapoznania się z problemami, które muszą być rozwiązane przez programistę.

Jak zauważymy, samo sortowanie zewnętrzne może być traktowane jako pewna *technika* programowania, gdyż do właściwej operacji sortowania, tej wykonywanej w pamięci operacyjnej, są wykorzystywane klasyczne, szybkie algorytmy sortowania wewnętrznego, a cały ciężar algorytmów spoczywa na umiejętnym manipulowaniu plikami roboczymi.

Nasze założenia modelu sortowania zewnętrznego zakładają następujące warunki i ograniczenia:

- ♦ Występują bardzo duże dysproporcje rozmiaru pamięci operacyjnej komputera (jest ona *ograniczona*) i zbioru danych do posortowania (teoretycznie *nieograniczony* lub ogólnie bardzo duży).
- ♦ Utrudniony jest dostęp do elementów przeznaczonych do sortowania, gdyż znajdują się one np. w plikach, które trzeba odczytywać przy pomocy mechanizmów systemu operacyjnego, zazwyczaj sekwencyjnie lub co najwyżej blokami.
- ♦ Czasy przetwarzania algorytmu sortującego w pamięci są pomijalne w porównaniu z czasami odczytu i zapisu danych z i do plików zewnętrznych.
- ♦ Fizyczne ograniczenia struktury dysków sprawiają, że korzystne jest odczytywanie danych z pliku wejściowego w większych blokach (stronach) pamięci. W związku z tym zawartość pliku dyskowego można traktować jako swego rodzaju listę złożoną z serii poszczególnych bloków.
- ♦ Do wykonywania sortowania fragmentów pliku mieszczących się w pamięci operacyjnej używa się serii buforów wejściowych, czyli zarezerwowanego fragmentu pamięci operacyjnej, w której system operacyjny umieszcza odczytany z dysku blok danych lub z których pobiera blok danych do zapisania na dysku.

Posiadając taką moc przerobową, możliwości odczytywania i zapisywania plików oraz dysponując pewnym szybkim klasycznym algorytmem do sortowania *wewnętrznego*, można zaproponować np. metodę sortowania polegającą na *rozdzieleniu i scalaniu połączonym z sortowaniem*:

- ♦ Dzielimy duży plik wejściowy na mniejsze, cząstkowe pliki, takie, które bez problemu mieszczą się w pamięci głównej.
- ♦ Sortujemy pliki cząstkowe znaną nam szybką metodą, np. Quicksort, i zapisujemy je już w wersji posortowanej. W praktyce ten punkt polega na sekwencyjnym zapisie do pamięci porcji pliku wejściowego i gdy zostanie przekroczony pewien umowny rozmiar, sortujemy w pamięci operacyjnej odczytaną zawartość i zapisujemy wynik jako kolejny plik cząstkowy. Postępujemy tak aż do pełnego podzielenia pliku wejściowego na mniejsze, już posortowane.
- ♦ W kolejnych krokach (przebiegach) scalamy te mniejsze i posortowane pliki w większe, aż do momentu posortowania całego pliku wejściowego.

Jak może działać operacja scalania plików posortowanych? Nie jest to tak trudne, jak samo sortowanie, gdyż dane wejściowe są już uporządkowane. Prosty algorytm scalający dane posortowane rosnąco mógłby wyglądać tak:

- ♦ Otwórz *pliki wejściowe*.
- ♦ Odczytuj sekwencyjnie dane z obu plików i przenoś wartość mniejszą do *pliku wyjściowego*, aż do momentu gdy któryś z plików się zakończy.
- ♦ Dopisz resztę na koniec pliku wyjściowego.

Oczywiście umowna prostota wcale nie oznacza krótkiego kodu w C++, poprawna implementacja, badająca warunki końca plików i wykonująca szereg porównań, może zająć nawet kilkadziesiąt linii kodu. Pewną alternatywą, wcale nie taką niemądrą, jest sklejenie plików razem i posortowanie przy pomocy komend systemu operacyjnego, np. tak jak to robi listing poniżej.



systemsort.cpp

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream plik_W1 ("input1.txt"); //plik wejściowy 1
    ifstream plik_W2 ("input2.txt"); //plik wejściowy 2
    ofstream plik_WYJ ("output.txt"); //plik wynikowy
    string s; //złączamy pliki ze sobą, używając C++
    while (getline(plik_W1,s))
        plik_WYJ << s << endl;
    while (getline(plik_W2,s))
        plik_WYJ << s << endl;
    plik_WYJ.close(); //zwalniamy plik wynikowy
    // Oczywiście złączanie można ewentualnie wykonać przez polecenie
    // systemowe: system("copy input1.txt+input2.txt output.txt");
    system("sort output.txt /O output.txt"); //sortujemy plik wynikowy
    // przy pomocy komendy systemu operacyjnego
}
```

Wady sortowania systemowego są oczywiste:

- ◆ Tracimy kontrolę nad efektywnością (proces sortowania może mieć tak niski priorytet, że wykona się bardzo wolno).
- ◆ Program w C++ przestaje być przenośny (np. aby powyższy kod zadziałał pod Linuxem, należy dostosować składnię komendy sort, zamienić copy na cp).
- ◆ Sortowanie systemowe domyślnie nie rozróżnia liczb i znaków, np. gdyby nasz plik zawierał liczby, to mogłoby się okazać, że „111” jest mniejsze od „22”.

Jak zatem posortować pliki z ciągami liczb, ale zapisanych jako znaki (w edytorze tekstowym)? W systemie Unix (Linux) możemy zmusić komendę sort do interpretowania wartości numerycznych (przełącznik -n), co jednak zrobić w DOS-ie (Windows)?

Moja sugestia jest następująca: proszę skopiować zawartość plików dyskowych do np. jednej tablicy liczb całkowitych, posortować ją jakąś znaną nam metodą i posortowaną tablicę znowu zapisać do pliku.

Jedyny kłopot mogą nam sprawić konwersje z formatu tekstowego na format liczby całkowitej. W C++ można do tego wykorzystać prostą instrukcję:

```
int jakaś_zmienna_int = atoi(jakaś_zmienna_string.c_str());
```

Zamiast w programie porównywać odczytywane znaki (typ string), konwertujemy je do zmiennej typu całkowitego i jej używamy w instrukcjach porównania.

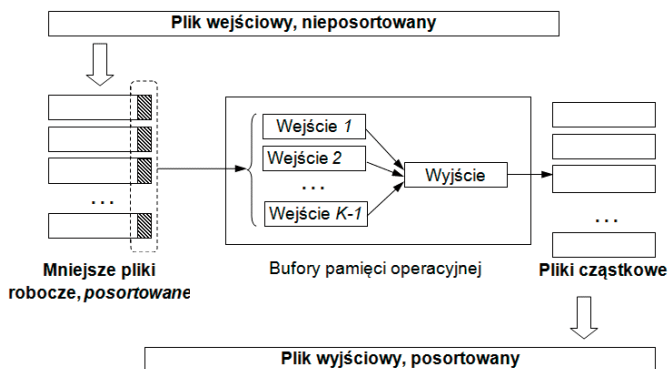
Jak łatwo zauważyć, sortowanie zewnętrzne bardziej dotyczy konkretnych problemów systemowych (pliki, konwersje danych) niż zagadnień czysto algorytmicznych. Wróćmy jednak na chwilę do modelu teoretycznego.

Pewien uogólniony schemat systemu sortowania zewnętrznego jest pokazany na rysunku 4.12. Zakładamy, że plik wejściowy jest N-stronicowy, a do jego sortowania możemy używać, oprócz mocy procesora, umownych obszarów pamięci:

- ◆ $K-1$ buforów *wejściowych*, służących do przetwarzania danych z plików wejściowych.
- ◆ 1 bufor *wyjściowy*, służący do zapisu do plików zewnętrznych (roboczych lub wynikowych).

Rysunek 4.12.

Ogólny model
sortowania
zewnętrznego



Suma tych K buforów nie może oczywiście przekraczać pojemności dostępnej pamięci operacyjnej.

Proces sortowania będzie oczywiście wieloprzebiegowy, bowiem w pojedynczym przebiegu możemy przetworzyć wyłącznie fragment dużego pliku wejściowego. W pierwszym, „zerowym” przebiegu trzeba podzielić plik wejściowy na mniejsze pliki robocze i posortować je. Rozmiar tego „mniejszego” pliku zależy od dostępnej pamięci operacyjnej — im większy, tym lepiej.

Następnie, przy użyciu naszych umownych buforów w pamięci operacyjnej, możemy odczytywać początkowe (a zatem posortowane po „zerowym” przebiegu) fragmenty plików posortowanych (patrz lewa część rysunku), sortować je przez złączanie w pamięci komputera (przy użyciu buforów pokazanych w środkowej części rysunku) i zapisywać do plików cząstkowych wynikowych pokazanych z prawej strony. (W jakimś prawdziwym systemie operacyjnym te umowne pliki z prawej strony mogą być dynamicznie tworzone lub też mogą być używane ponownie poprzednie pliki do zapisywania nowych rezultatów cząstkowych).

Proces sortowania przez scalanie z użyciem buforów i zapisu do pliku (plików) wynikowego (wynikowych) kontynuujemy aż do momentu, gdy pliki z lewej strony i same bufory staną się puste. Tak jak już wcześniej wspomniałem, koszt sortowania algorytmu jest związany z liczbą wykonywanych przebiegów, gdyż operacje plikowe stanowią największy narzut czasowy. Ponieważ wąskim gardłem jest sam podział pliku na N -bloków i liczba zastosowanych K -buforów w pamięci głównej, można obliczyć, że koszt sortowania zewnętrznego wyniesie:

$$2N \cdot \left\lceil 1 + \log_{K-1} \left\lceil \frac{N}{K} \right\rceil \right\rceil$$

Prawy człon tego iloczynu stanowi tak naprawdę liczbę przebiegów, wartości są zaokrąglane do najbliższej wartości całkowitej w górę (symbol $\lceil \rceil$).

Uwagi praktyczne

Kryteria wyboru algorytmu sortowania mogą być zebrane w kilka łatwych do zapamiętania zasad:

- ♦ Do sortowania małej liczby elementów nie używaj superszybkich algorytmów, takich jak np. *Quicksort*, gdyż zysk będzie znikomy.
- ♦ Część znanych z literatury i prasy fachowej algorytmów sortowania nie jest nigdy — lub jest bardzo rzadko — stosowana praktycznie. Powodem ich stworzenia były prace akademickie, które nigdy nie zostały wykorzystane w praktyce programistycznej. Zresztą, trzymając się dobrze znanych metod, mamy większą pewność, iż nie popełnimy jakiegos nadprogramowego błędu!

Podczas programowania warto również uważnie czytać, czy w bibliotekach standardowych używanego kompilatora nie ma już zaimplementowanej funkcji sortującej. Przykładowo: w wielu kompilatorach⁶ istnieje gotowa funkcja o nazwie... `qsort` o następującym nagłówku:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *))
```

Tablica do posortowania może być dowolna (typ `void`), ale musimy dokładnie podać jej rozmiar: liczbę elementów `nmemb` o rozmiarze `size`. Funkcja `qsort` wymaga ponadto jako parametru *wskaźnika do funkcji porównawczej*. Przy omawianiu list jednokierunkowych dokładnie omówiono pojęcie wskaźników do funkcji; tutaj w celu zilustrowania podam tylko gotowy kod do sortowania tablicy liczb całkowitych (przeróbka na sortowanie tablic innego typu niż `int` wymaga jedynie modyfikacji funkcji porównawczej `comp` i sposobu wywołania funkcji `qsort`):



qsort2.cpp

```
int comp(const void *x, const void *y)
{
    int xx=(int*)x; //jawna konwersja z typu
    int yy=(int*)y; //void* do int*
    if ( xx < yy)
        return -1;
    if ( xx == yy)
        return 0;
    else
        return 1;
}
const int n=12;
int tab[n]={40,29,2,1,6,18,20,32,34,39,23,41};
int main()
{
    for (int i=0; i<n; i++)
        cout << tab[i] <<" ";
    cout << endl;
    qsort(tab, n, sizeof(int), comp);
    for (int i=0; i<n; i++)
        cout << tab[i] <<" ";
    cout << endl;
}
```

Funkcja porównawcza `comp` zmienia się w zależności od typu danych sortowanej tablicy. Przykładowo: dla tablicy wskaźników ciągów znaków użylibyśmy jej następująco:

```
int comp(const void* a, const void* b)
{
    return(strcmp((char*)a,(char*)b));
}

int main()
{
    char s[5][4]={ "aaa", "ccc", "ddd", "zzz", "fff" };
    qsort((void*)s, 5, sizeof(s[0]), comp);
    for (int i=0; i<5; i++)
        cout << s[i] << endl;
}
```

Wadą stosowania gotowej funkcji bibliotecznej jest brak dostępu do kodu źródłowego: dostajemy koda w worku i musimy się do niego przyzwyczaić.

Pisanie własnej procedury sortującej ma tę zaletę, że możemy ją zoptymalizować pod kątem naszej własnej aplikacji. Już wbudowanie funkcji `comp` wprost do procedury sortującej powinno nieco poprawić jej parametry czasowe, nie zmieniając jednak klasy algorytmu!

⁶ Przykładowo: Borland C++ Builder, GNU C++, Visual C++.

Rozdział 5.

Typy i struktury danych

Nikogo nie trzeba chyba przekonywać o wadze tematu, który zostanie poruszony w tym rozdziale. Od wyboru właściwej w danym momencie struktury danych może zależeć wszystko: szybkość działania programu, zakres obsługiwanych przypadków podstawowych, łatwość i możliwość modyfikacji, czytelność zapisu algorytmów, przenośność i... w konsekwencji dobre samopoczucie programisty.

Typy podstawowe i złożone

Każdy, kto poznał jakikolwiek język programowania, został niejako zmuszony do opanowania zasad posługiwania się tzw. typami podstawowymi (wbudowanymi).

Przykładowo w C++ dysponujemy m.in. następującymi typami:

- ♦ `int` i `long` (liczby całkowite, np. deklaracja `int zmienna_typu_int=5;` pozwala na przechowywanie w zmiennej `zmienna_typu_int` wartości całkowitych, tutaj przykładowo 5),
- ♦ `float` i `double` (liczby zmiennopozycyjne),
- ♦ `char` (znaki, np. `char znak='a';`),
- ♦ `bool` (przydatny do deklarowania zmiennych logicznych, przyjmujących wartości `true` i `false`),
- ♦ typy wskaźnikowe¹ (z „gwiazdką”, służące do przechowywania adresów, np. deklaracja `int *zm;` oznacza, że `zm` będzie wskazywała na zmienną typu całkowitego. „Wskaźnik” przechowuje *adres*, aby „wyluskać” z niego wskazywaną wartość, trzeba go poprzedzić symbolem gwiazdki (przykładowo `zm` zawiera adres, a `*zm` — wartość, oczywiście pod warunkiem, że zmienna `m` została wcześniej zainicjowana, np. poprzez instrukcję `zm=&zmienna_typu_int`). Operator `&` wyluskuje ze zmiennej jej adres i stanowi, jak widać, przeciwieństwo operatora `*`).

Typy podstawowe stanowią niezbędne cegiełki, z których budujemy bardziej złożone struktury danych i warto je dobrze opanować, aby np. rozumieć, że `*` może oznaczać nie tylko mnożenie. Ponadto każdy typ podstawowy jednoznacznie determinuje zbiór dozwolonych wartości (zakres) oraz operacji (np. dodawanie, mnożenie itp.).

¹ Dodatek A omawia także tzw. referencje, które są nieco wygodniejsze w użyciu od wskaźników, choć nie do końca mogą je zastąpić w rzeczywistych programach.

W C++ niektóre typy danych mają charakter obiektowy (np. `string`, `vector`), co pozwala na łatwe manipulowanie ich zawartością (np. dodawanie obiektów) oraz izoluje nas od kwestii technicznych, takich jak np. przydzielanie i zwalnianie pamięci.



Ostrzeżenie

Lista typów podstawowych oraz ich precyzja jest w C++ określona przez standard tego języka, ale ich konkretna *implementacja* (np. liczba bitów zajmowanych przez zmienną określonego typu) — już od konkretnej technologii. Jeśli zatem nasze programy używają niskopoziomowych cech sprzętu, takich jak rozmiar komórki pamięci, to tracą przenośność! Minimalne i maksymalne wartości poszczególnych typów danych znajdziesz w plikach `limits.h` i `float.h`.

Zakres dozwolonych wartości jest szczególnie istotny dla algorytmów matematycznych. Przykładowo w Visual C++ znajdziemy w pliku `limits.h` następujące informacje:

```
#define INT_MIN  (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX   2147483647      /* maximum (signed) int value */
```

Jeśli zatem użyjesz zmiennej `int` do wyliczania przypadków pewnego algorytmu, który teoretycznie może się nie zmieścić w powyższym zakresie, to musisz albo użyć zmiennej innego typu, albo zastosować sztuczki podobne do tych podanych np. w rozdziale 13. w punkcie „Kodowanie danych i arytmetyka dużych liczb”.

Nawet początkujący programista C++ używa bez problemu nieco bardziej złożonych struktur danych, do jakich należą tablice i rekordy. Tablice i rekordy są na tyle proste, że nie będą one stanowiły przedmiotu naszych głębszych rozważań. (Jeśli jednak masz kłopoty ze zrozumieniem składni C++, to zerknij do dodatku A po konkretne przykłady). Warto jednak przytoczyć ogólne cechy tych „nieco bardziej złożonych” struktur danych, aby mieć świadomość ich zalet i wad:

- ◆ Tablice (np. `int t[N]`) pozwalają na przechowywanie obok siebie (dosłownie) zbioru `N` zmiennych tego samego typu, adresowanych przez podanie indeksu od 0 do `N-1` (`t[0]`, `t[1]`, ... `t[N-1]`). Dostęp do danych jest zatem naturalny i swobodny: znając indeks konkretnej wartości, można do niej sięgnąć poprzez adresowanie bezpośrednie. Wadą klasycznych tablic jest oczywiście konieczność rezerwacji pewnego stałego obszaru pamięci. Tablice w C++ mogą być wielowymiarowe i zawierać nie tylko zmienne typu podstawowego, ale i rekordy danych.
- ◆ Rekordy (w C++ definiowane przez słowo kluczowe `struct`) tworzą nowy typ podstawowy, który pozwala na grupowanie w swego rodzaju „paczce” kilku zmiennych innego typu. Dzięki temu łatwo jest zgrupować np. informacje adresowe, zarobki, numer PESEL (są to tzw. pola) itp. w jednym miejscu. Dostęp do pól jest bardzo naturalny (notacja z kropką — bardziej rozbudowane przykłady w C++ znajdują się w dodatku A), pola mogą też zawierać typy złożone (np. tablice) oraz wskaźniki.



Uwaga

W C++ występuje też słowo kluczowe `union`, zaszłość z języka C pozwalająca na oszczędne wykorzystanie pamięci komputera w pewnych specyficznych sytuacjach poprzez definiowanie tzw. unii, ale ceną jest niska czytelność tworzonego kodu. Na pierwszy rzut oka definicja unii jest bowiem podobna do definicji rekordu, ale jest to tylko pozór: pola w unii „nachodzą” na siebie w pamięci i programista musi ściśle kontrolować w programie, co tak naprawdę z niej odczytuje. Jeśli zależy Ci na czytelności... nie używaj unii, gdyż jest to konstrukcja języka bardzo myląca i w zasadzie zbędna w większości zastosowań.

Ciągi znaków i napisy w C++

Czytelnik ma prawo być nieco zdziwiony pojawieniem się tego punktu, gdyż ma on charakter ściśle związany z C++ i dość rzadko bywa wzmiankowany w podręcznikach algorytmiki (chlubnym wyjątkiem jest [Sed92]). Tymczasem wyodrębnienie dyskusji o napisach i ciągach znaków ma

sens, gdyż umiejętność manipulowania napisami w programach ułatwia rozwiązanie wielu zagadnień. Bez sprawnego przetwarzania napisów nie da się napisać sensownej aplikacji biznesowej, więc postanowiłem w tym wydaniu książki pokazać kilka przykładów, które powinny wyjaśnić tajemnicze konstrukcje spotykane w kodzie C++.

Pierwsze, co warto zrozumieć, to fakt, iż ciąg znaków w C++ jest tablicą... o zmiennej długości. Ta pozornie niewielka różnica jest dość istotna z punktu widzenia systemowego: zwykła tablica jest od początku statycznie określona w programie, posiada znaną nam długość i wiemy, jak czytać poszczególne elementy (np. znaki, liczby, rekordy). Ciąg znaków zachowuje się teoretycznie dość podobnie, ale ponieważ jego długość może się zmieniać w trakcie działania programu, to kompilator potrzebuje dodatkowej informacji, znacznika końca, którym jest specjalny kod zero (zapisywany jako `'\0'`).

W C++ spotyka się często deklaracje zbliżone do:

```
char *s1;
char s2[100];
char *s3="Kod błędu 100";
```

W pierwszym przypadku deklarowany jest wskaźnik, który ewentualnie można przypisać do jakiejś zmiennej tekstowej (np. do argumentów wywołania funkcji `main`, innej zmiennej tekstowej), ale sam w sobie *nie rezerwuje on pamięci* do przechowywania znaków! Dopiero druga forma pozwala na bezpieczne składowanie znaków i swobodny dostęp do poszczególnych znaków poprzez indeksowanie, bez łamania reguł bezpieczeństwa w zakresie odczytywania pamięci. Trzecia forma jest równoważna zadeklarowaniu wskaźnika `s3`, przydzieleniu w pamięci miejsca na 14 znaków (tekst + znacznik końca, czyli 0) i zainicjowaniu zawartości napisem „Kod błędu 100”.

Popatrzmy na prosty program przykładowy, który odczytuje ciąg znaków, zapisuje go do tablicy i odczytuje jej zawartość, znak po znaku, włącznie ze znacznikiem końca, automatycznie dołożonym przez program.



znaki.cpp

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
    char s[100]; // na razie tu nic nie ma!
    cout << "Podaj słowo:";
    cin >> s;
    for (int i=0; i<=strlen(s); i++) // strlen = długość ciągu
        cout << "Znak [" << i << "]=" << s[i] << ",kod: "
            << (int) s[i] << endl;
}
```

Oto przykładowe wykonanie programu:

```
Podaj słowo:pies
Znak [0]=p,kod: 112
Znak [1]=i,kod: 105
Znak [2]=e,kod: 101
Znak [3]=s,kod: 115
Znak [4]= ,kod: 0
```

Gdybyśmy w tym kodzie zastąpili `char s[100]` przez `char *s`, to program zgłosiłby błąd dostępu do pamięci!

Konieczność badania długości ciągów znakowych, kontrolowania ewentualnego przepełniania „buforów” i rezerwowania pamięci była zawsze koszmarem wielu programistów C++, prowadząc do wielu pomyłek wykrywalnych dopiero podczas wykonywania programu. Z tego powodu

w odświeżonej wersji standardu C++ pojawił się wygodny, obiektowy typ danych o nazwie `string`, umożliwiający łatwe operowanie łańcuchami znaków, bez nużącego używania tablic i wskaźników i całej tej męczącej kontroli pamięci.

Popatrzmy na prostą ilustrację użycia obiektów klasy `string`.



string.cpp

```
int main()
{
    string s1, s2="ma kota";           // deklaracja + inicjalizacja
    s1 = "ala ";                       // przypisanie wartości
    string s3 = s1 + s2 + "\n";        // sklejanie łańcuchów
    cout << "s3=" << s3;
    s3.erase();                        // zerujemy ciąg znaków
    cout << "s3=" << s3;
}
```

Program po uruchomieniu wyświetli:

```
s3=ala ma kota
-----
s3=
s2: znak [0]=m.kod: 109
s2: znak [1]=a.kod: 97
s2: znak [2]=.kod: 32
s2: znak [3]=k.kod: 107
s2: znak [4]=o.kod: 111
s2: znak [5]=t.kod: 116
s2: znak [6]=a.kod: 97
```

Ten prosty kod pokazuje, jak łatwe i naturalne jest tworzenie i modyfikowanie ciągów znaków. Klasa `string` posiada oczywiście wiele ciekawych metod poza np. `erase`, a zdefiniowane (tzn. przeciążone) operatory standardowe pozwalają na przypisywanie i naturalne sklejanie ciągów znaków. Oprócz tego oczywiście pozostaje pełny dostęp do zawartości ciągów, tak jakby to była klasyczna tablica znakowa.



W tej książce w zależności od potrzeb będę stosował klasyczne (tablicowe) podejście i podejście obiektowe, z wykorzystaniem klasy `string`. Aby zostać sprawnym programistą C++ musisz niestety poznać oba te podejścia, gdyż do tej pory napisano bardzo dużo dobrych programów (często przeniesionych z C), które używają głównie podejścia klasycznego.

Abstrakcyjne struktury danych

Prawdziwa przygoda ze strukturami danych rozpoczyna się, dopiero gdy dostajemy do ręki tzw. listy, drzewa binarne, grafy itd. Wraz z nimi rozszerzają się znacznie możliwości rozwiązania programowego wielu ciekawych zagadnień; zwiększa się wachlarz potencjalnych zastosowań informatyki. Ponieważ z tymi strukturami związane są pewne reguły użycia, często określa się je jako *abstrakcyjne*, co w C++ przekłada się na... programowanie obiektowe, które doskonale pozwala implementować takie nieistniejące w naturze (czytaj: kompilatorach) twory!

Struktury danych są olbrzymim ułatwieniem dla programistów, gdyż pozwalają na uporządkowanie informacji zapamiętywanych w komputerach w formie łatwej do zrozumienia dla człowieka. Jak wykażemy w dalszej części książki, są one nie tylko formą organizacji danych, ale i ciekawym narzędziem do rozwiązywania skomplikowanych problemów algorytmicznych, np.:

- ♦ *Listy* ułatwiają tworzenie elastycznych baz danych.
- ♦ *Drzewa binarne* mogą posłużyć do analizy symbolicznej wyrażeń arytmetycznych.
- ♦ *Grafy* ułatwiają rozwiązywanie wielu zagadnień z dziedziny tzw. sztucznej inteligencji.

Listy i drzewa binarne omówimy w tym rozdziale, materiał dotyczący grafów został, ze względu na jego znaczenie i objętość, wyodrębniony w rozdziale 10.

W kolejnych podrozdziałach zostaną przedstawione najważniejsze struktury danych i sposoby posługiwania się nimi. Jednocześnie przykłady ilustrujące ich użycie zostały tak wybrane, aby zasugerować niejako ewentualną dziedzinę zastosowań.

Ten rozdział jest bogato ilustrowany kodem w C++, którego poziom skomplikowania może okazać się dużym wyzwaniem dla początkujących adeptów tego języka (np. szablonny klas, wskaźniki do funkcji, funkcje zaprzyjaźnione, przeciążanie operatorów...). Ponieważ nowoczesne implementacje języka C++ zawierają bogatą kolekcję gotowych klas realizujących nawet złożone struktury danych, to nie jest wymagane, aby każdy potrafił projektować od zera wszystko, co jest dostępne w postaci gotowej. Warto jednak co najmniej zrozumieć, w jaki sposób projektuje się nieco bardziej złożone klasy, aby nabrać wprawy w używaniu pewnych technik programowania specyficznego dla C++. Zapraszam zatem do lektury zarówno osoby zainteresowane samymi strukturami danych, jak i programistów pragnących wyłącznie podejrzeć gotowe rozwiązania niejako „od kuchni”.

Listy jednokierunkowe

Lista jednokierunkowa jest oszczędną pamięciowo strukturą danych, pozwalającą grupować dowolną — ograniczoną tylko ilością dostępnej pamięci — liczbę elementów: liczb, znaków, rekordów itd. Jest to duża zaleta w porównaniu z tablicami, których rozmiar może być, co prawda, określany dynamicznie, ale przydział dużego, „liniowego” obszaru pamięci podczas wykonywania programu nie zawsze musi się zakończyć sukcesem. Nietrudno sobie bowiem wyobrazić, że o wiele bardziej prawdopodobne jest bezproblemowe przydzielenie przez system operacyjny 50 000 razy pamięci na pojedyncze dane (powiedzmy 1 kB), niż zarezerwowanie „za jednym zamachem” miejsca na tablicę o rozmiarze 50 000 kB. Ponadto tablica, przynajmniej na początku, i tak będzie prawie w ogóle nieużywana!

Użycie listy wymaga zarezerwowania pewnych dodatkowych informacji wskaźnikowych, które oczywiście także zajmują miejsce w pamięci, nie przechowując w sobie bezpośrednio danych „biznesowych”. Ten narzut jednak jest niewielki i w dużych programach może być pomijalny, zwłaszcza dla wielkich rekordów informacyjnych.

Do budowy listy jednokierunkowej używane są dwa typy komórek pamięci (rysunek 5.1)²:

- ♦ Rekord natury informacyjnej (oznaczenie INFO), zawierający dwa wskaźniki: *do początku* listy i *do końca* listy. Wskaźnik jest adresem komórki w pamięci komputera i przechowywany jest w zmiennej typu wskaźnikowego.
- ♦ Rekord o charakterze roboczym (oznaczenie ELEMENT). Zawiera pole *wartości* (tutaj wpisujemy zawartość informacyjną, np. liczbę, ciąg znaków, zbiór atrybutów itd.) i wskaźnik na następny element listy.

Rysunek 5.1.
Typy rekordów
używanych podczas
programowania list



² Na rysunku pogrubiona kropka oznacza wartość typu „wskaźnik”.

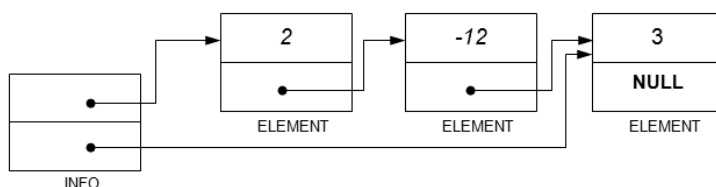
Zauważyłem, że w większości opisów struktur listowych nie wzmiankuje się zazwyczaj rekordu informacyjnego (nie jest on bezpośrednio elementem struktury danych) — jest to oczywisty błąd. Kosztem kilku bajtów pamięci³ uzyskujemy bowiem ciągły dostęp do bardzo istotnych operacji i ułatwiamy ogromnie operację podstawową: dołączenie nowego elementu na koniec listy (jeśli nie wstawiamy na koniec listy, to zawsze możemy przyłączyć nowy element na początek listy, ale tracimy wówczas informację o kolejności przybywania danych!).

Pola: głowa, ogon i następny są wskaźnikami⁴, natomiast wartość może być czymkolwiek — liczbą, znakiem, rekordem, etc. W przykładach znajdujących się w tej książce dla uproszczenia operuje się głównie wartościami typu całkowitego, co nie umniejsza bynajmniej ogólności wywodu. Ewentualne przeróbki tak uproszczonych algorytmów należą już raczej do kosmetyki niż do zmian o charakterze zasadniczym.

Idea jest zatem następująca: jeżeli lista jest pusta, to struktura informacyjna zawiera dwa wskaźniki NULL. Warto pamiętać, że w ogólnym przypadku NULL nie jest bynajmniej równy zeru — jest to pewien adres, na który na pewno żadna zmienna nie wskazuje (taka jest ogólna idea wskaźnika NULL, niestety wielu programistów o tym nie pamięta). Pierwszy element listy jest złożony z jego własnej wartości (informacji do przechowania) oraz ze wskaźnika na drugi element listy. Drugi zawiera własne pole informacyjne i, oczywiście, wskaźnik na trzeci element listy itd. Miejsce zakończenia listy zaznaczamy także poprzez wartość specjalną, wskaźnik NULL. Spójrzmy na rysunek 5.2 przedstawiający listę złożoną z trzech elementów: 2, -12, 3.

Rysunek 5.2.

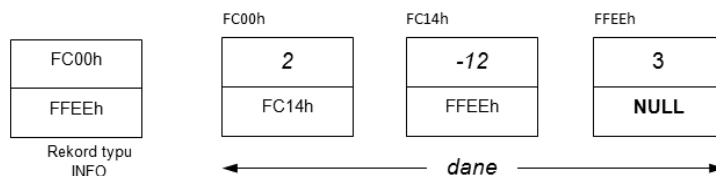
Przykład listy
jednokierunkowej (1)



Rysunek 5.3 jest dokładnym odbiciem swojego poprzednika — z tą tylko różnicą, że — w miejsce strzałek symbolizujących „wskazywanie” — są użyte konkretne wartości liczbowe adresów komórek pamięci. Liczba szesnastkowa, umieszczona nad rekordem, jest adresem w pamięci operacyjnej komputera, pod którym zostało mu przydzielone miejsce przez standardową procedurę `new`⁵.

Rysunek 5.3.

Przykład listy
jednokierunkowej (2)



Wróćmy jeszcze do analizy rekordów składających się na listę. Pole `głowa` struktury informacyjnej wskazuje na komórkę zawierającą 2 — pierwszy element listy, czyli — wyrażając się jaśniej — zawiera adres, pod którym w pamięci komputera jest zapamiętany rekord.

³ Wielkość zmiennej wskaźnikowej zależy od potrzeb adresowych wymaganych dla używanego modelu pamięci i determinuje ją zasadniczo system operacyjny.

⁴ Fakt wskazywania na coś jest symbolizowany dalej przez strzałki.

⁵ Ze względów historycznych warto może przypomnieć, że w klasycznym języku C trzeba było w celu przydzielania pamięci używać funkcji bibliotecznych `calloc` i `malloc`. W C++ instrukcja `new` wykonuje dokładnie to samo, lecz o wiele czytelniej, i stanowi już element języka.

Pole ogon struktury informacyjnej wskazuje na komórkę zawierającą 3 (ostatni element listy). Pola te służą do przeglądania elementów listy i do dołączania nowych. Oto jak może wyglądać procedura przeglądająca elementy listy, np. w poszukiwaniu wartości *x* (komórka informacyjna nazywa się *info*):

```
adres_tmp=info.glowa
dopóki (adres_tmp != NULL) wykonuj
{
    jeśli(adres_tmp.wartość == x) to
    {
        wypisz "Znalazłem poszukiwany element"
        opuść procedurę
    }
    w przeciwnym przypadku
        adres_tmp=adres_tmp.następny
}
wypisz "Nie znalazłem poszukiwanego elementu"
```

W dalszej części rozdziału będziemy przeplatać opis algorytmów w pseudojęzyku programowania (takim jak wyżej) z gotowym kodem C++; kryterium wyboru będzie czytelność procedur. Oczywiście nawet jeśli prezentacja algorytmu zostanie dokonana w pseudokodzie, to wersja dyskietykowa będzie zawierała pełne wersje w języku C++, gotowe do kompilacji i uruchomienia.

Realizacja struktur danych listy jednokierunkowej

Poniższa implementacja struktur potrzebnych do programowej obsługi listy jednokierunkowej jest dokładnym odzwierciedleniem rysunku 5.2 i nie należy się tu spodziewać szczególnych niespodzianek. Osoby, które nie znają jeszcze zbyt dobrze składni języka C++, powinny dobrze zapamiętać sposób deklaracji typów danych rekurencyjnych (tzn. zawierających wskaźniki do elementów swojego typu). Różni się on bowiem odrobinę od sposobu używanego na przykład w Pascalu (patrz również dodatek A).



lista.h

```
// definicje typów danych
enum szukanie {PORAZKA=0, SUKCES=1};
typedef struct rob
{
    int wartosc;
    struct rob *następny; // wskaźnik do następnego elementu
}ELEMENT;

// początek deklaracji klasy LISTA
class LISTA
{
public:
    // nagłówki:
    friend LISTA& operator +(LISTA&,LISTA&); // sumuje dwie listy
    friend void fuzja(LISTA &x,LISTA &y);
    void wypisz(); // wypisuje zawartość listy
    int szukaj(int x); // szuka elementu x na liście
    void dorzuc1(int x); // dorzuca bez sortowania
    void dorzuc2(int x); // dorzuca z sortowaniem
    LISTA& operator --(int); // usuwa ostatni element listy

    // kilka prostych metod:
    bool pusta() // czy lista coś zawiera?
    {
        return (inf.glowa==NULL);
    }
}
```

```

void zeruj()           // zeruje listę bez wykonywania „delete”
{
    inf.glowa=inf.ogon=NULL;
}

LISTA()               // konstruktor
{
    inf.glowa=inf.ogon=NULL;
}
~LISTA()
{
    // destruktor, który używa przeddefiniowanego operatora --
    while (!pusta()) (*this)--;
}
private:
// struktura informacyjna zapewni dostęp do listy
typedef struct
{
    ELEMENT *glowa;
    ELEMENT *ogon;
} INFO;
INFO inf;
}; //koniec deklaracji klasy LISTA

```

Pole wartość w naszym przykładzie jest typu `int`, ale w praktyce może to być bardzo złożony rekord informacyjny (np. zawierający *imię*, *nazwisko*, *wiek* itd.).

Kilka prostych metod klasy `LISTA` zdefiniowaliśmy już w pliku nagłówkowym. Jeśli metoda jest trywialna i charakteryzuje się małymi rozmiarami, to często definiuje się ją wprost w ciele klasy. Przykładem może być miniaturowa funkcja usługowa `pusta`, która pomimo swej prostoty ma szansę być dość często używana w praktyce.

Klasa `LISTA` nie jest zbyt rozbudowana, jednak zawiera kilka rozwiązań, które wymagają dość szczegółowego komentarza. Osoby słabiej znające C++ mogą mieć kłopot ze zrozumieniem deklaracji:

```
LISTA& operator --(int);
```

Jest to oczywiście nagłówek metody, która przeddefiniuje operator `--`, ale do czego służy ten dodatkowy parametr typu `int`? Otóż wynika on z wymogów normalizacyjnych języka C++. Parametr ten jest sztuczny i wyłącznie informuje kompilator o tym, że przeddefiniujemy w tej metodzie operator „przyrostkowy”.

Kwestią otwartą pozostaje wybór ewentualnego utajnienia typów danych (patrz deklaracje `public` i `private`); programista musi sam podjąć odpowiednią decyzję, mając na uwadze takie aspekty, jak: *sens* ujawniania/ukrywania atrybutów, parametry „sprawnościowe” metod, późniejsze dziedziczenie, etc. Propozycje przedstawione w tym rozdziale w żadnym razie nie pretendują do miana rozwiązań wzorcowych — takie bowiem nie istnieją wobec nieskończonej w zasadzie liczby nowych sytuacji i problemów, z którymi może się w praktyce spotkać programista. Staraniem autora było raczej pokazanie istniejącej różnorodności, a nie przekonywanie do jednych rozwiązań przy jednoczesnym pomijaniu innych.

W następnych paragrafach zostaną przedstawione wszystkie metody, które były wyżej wymienione jedynie poprzez swoje nagłówki.

Tworzenie listy jednokierunkowej

Niewątpliwie najwyższa już pora na przedstawienie sposobu dołączania elementów do listy. Posłużymy nam do tego celu kilka funkcji o mniejszym lub większym stopniu skomplikowania. Potrzeba sprawdzania, czy jakieś elementy już są zapamiętane na liście, wystąpi przykładowo w funkcji `dorzuc1`, która dołącza nowy element do listy.

Podczas dokładania nowego elementu możliwe są dwa podejścia: albo będziemy traktować listę jako zwykły worek do gromadzenia danych nieuporządkowanych (będzie to wówczas naukowy sposób na zwiększanie bałaganu), albo też przyjmiemy założenie, że nowe elementy dokładane będą w liście we właściwym, ustalonym przez nas porządku — na przykład sortowane od razu w kierunku wartości niemalejących.

Pierwszy przypadek jest trywialny — odpowiadająca mu procedura `dorzuc1` jest przedstawiona poniżej:



lista.cpp

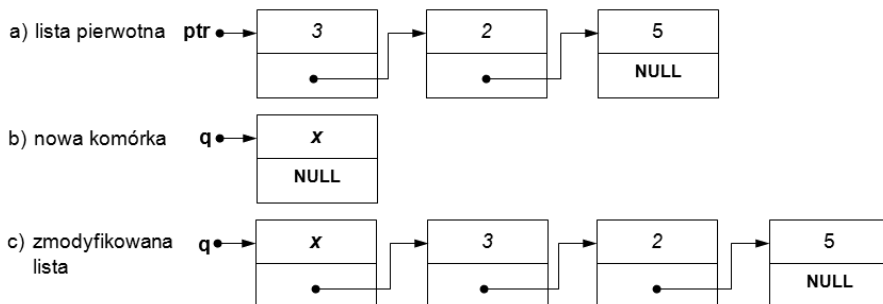
```
// Fragment pliku lista.cpp, na razie bez funkcji main:
#include <iostream>
using namespace std;
#include "LISTA.H"           // bez tej dyrektywy kompilator nie zrozumie
                             // nowego typu danych LISTA
void LISTA::dorzuc1(int x)   // dołączamy rekord na koniec listy
{                             // bez sortowania; operator :: jest
    ELEMENT *q=new ELEMENT; // niezbędny, bowiem definiujemy kod
    q->wartosc=x;             // metody poza „ciałem” klasy
    q->nastepny=NULL;
    if (pusta())             // lista pusta?
        inf.glowa=inf.ogon=q;
    else
        // coś jest w liście
        {
            (inf.ogon->nastepny=q; // pole następny jest
            inf.ogon=q;           // wskaźnikiem, stąd -> a nie . (kropka)
        }
}
```

Działanie funkcji `dorzuc1` jest następujące: w przypadku pustej listy oba pola struktury informacyjnej są inicjowane wskaźnikiem na nowo powstały element. W przeciwnym wypadku nowy element zostaje podpięty do końca, stając się tym samym ogonem listy.

Oczywiście możliwe jest dokładanie nowego rekordu przez *pierwszy* element listy (wskazywanej zawsze przez pewien łatwo dostępny wskaźnik, powiedzmy: `ptr`), stawałby się on wówczas automatycznie głową listy i musiałby zostać zapamiętany przez program, aby nie stracić dostępu do danych:

```
ELEMENT    *q=new ELEMENT;    // a)
            q->wartosc=x;       // b)
            q->nastepny=ptr;    // c)
```

Kod ten może być zilustrowany schematem z rysunku 5.4.



Rysunek 5.4. Dołączanie nowego elementu listy na jej początek



Ostrzeżenie

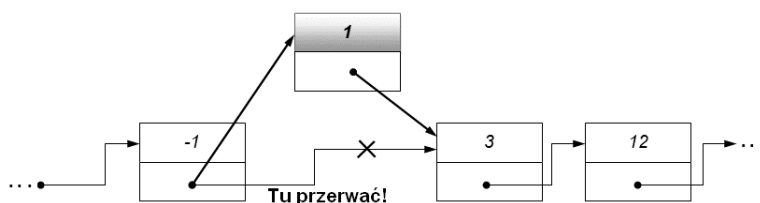
W tym i dalszych przykładach przyjmowane jest założenie, że przydział pamięci ZAWSZE kończy się sukcesem. W rzeczywistych programach jest to przypuszczenie dość niebezpieczne i warto sprawdzać, czy istotnie po użyciu instrukcji używającej `new`, np. `ELEMENT *q=new ELEMENT;` wartości `q` nie zostało przypisane `NULL`! Z uwagi na chęć zapewnienia klarowności prezentowanych algorytmów tego typu kontrola zostanie w książce pominięta; podczas realizacji „prawdziwego” programu takie niedopatrzenie może się okazać dość przykre w skutkach.

Sposób podany powyżej jest poprawny, ale pamiętajmy, że dokładając nowe elementy zawsze na *początek* listy, tracimy istotną czasami informację na temat *kolejności* nadchodzenia elementów!

O wiele bardziej złożona jest funkcja dołączająca nowy element w takie miejsce, aby całość listy była widziana jako posortowana (tutaj: w kierunku wartości niemalejących). Ideę przedstawia rysunek 5.5, gdzie możemy zobaczyć sposób dołączania liczby 1 do już istniejącej listy złożonej z elementów -1, 3 i 12.

Rysunek 5.5.

Dołączanie elementu listy z sortowaniem



Nowy element (narysowany pogrubioną kreską i zacieniowany) może zostać wstawiony na początek (*a*), koniec (*b*) listy, jak również gdzieś w jej środku (*c*). W każdym z tych przypadków w istniejącej liście trzeba znaleźć miejsce wstawienia, tzn. zapamiętać dwa wskaźniki: element, *przed którym* mamy wstawić nową komórkę, i element, *za którym* mamy to zrobić. Do zapamiętania tych istotnych informacji posłużą nam zmienne *przed* i *po*.

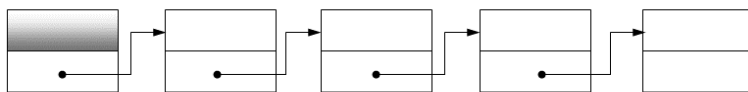
Następnie, gdy dowiemy się, gdzie jesteśmy, możemy dokonać wstawienia nowego elementu do listy. Sposób, w jaki tego dokonamy, zależy oczywiście od miejsca wstawienia i od tego, czy lista przypadkiem nie jest jeszcze pusta. Krótko mówiąc, realizacja jest niestety dość złożona. Pewne skomplikowanie funkcji `dorzuc2` wynika z połączenia w niej poszukiwania miejsca wstawienia z samym dołączeniem elementu. Równie dobrze można by te dwie czynności rozbić na osobne funkcje — nie zostało to jednak uczynione w obecnej wersji.

Istnieją 3 przypadki „współrzędnych” nowego elementu w liście, symbolicznie przedstawione na rysunku 5.6 (zakładamy, że lista już coś zawiera).

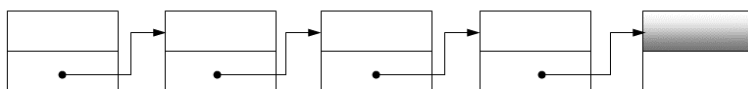
Rysunek 5.6.

Wstawianie nowego elementu do listy — analiza przypadków

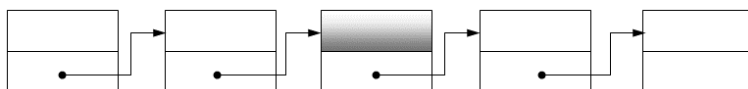
a) Wstawiamy na początek listy (*przed=NULL*)



b) Wstawiamy na koniec listy (*po=NULL*)



c) Wstawiamy w środek listy (*przed≠NULL, po≠NULL*)



W zależności od ich wystąpienia zmienia się sposób dołączenia elementu do listy. Oto pełny tekst funkcji `dorzuc2`, która swoje działanie opiera właśnie na idei przedstawionej na rysunku 5.6:

```
void LISTA::dorzuc2(int x)    // dołączamy rekord na właściwe miejsce
{
    ELEMENT *q=new ELEMENT;  // z sortowaniem
    q->wartosc=x;
    // Poszukiwanie właściwej pozycji na wstawienie elementu
    if (pusta())
    {
        inf.glowa=inf.ogon=q;
        q->nastepny=NULL;
    }
    else //szukamy miejsca na wstawienie
    {
        ELEMENT *przed=NULL, *po=inf.glowa;
        // zmienna wyliczeniowa
        enum {SZUKAJ,ZAKONCZ} stan=SZUKAJ;
        while ((stan==SZUKAJ) && (po!=NULL))
            if (po->wartosc==x)
                stan=ZAKONCZ; // znaleźliśmy właściwe miejsce!
            else //przemieszczamy się w poszukiwaniach
            {
                // właściwego miejsca
                przed=po;      // wskaźniki „przed” i „po”
                po=po->nastepny; // zapamiętaj miejsce wstawiania
            }
        if (przed==NULL) // wstawiamy na początek listy
        {
            inf.glowa=q;
            q->nastepny=po;
        } else
        if (po==NULL) // wstawiamy na koniec listy
        {
            inf.ogon->nastepny=q;
            q->nastepny=NULL;
            inf.ogon=q;
        }
        else // wstawiamy gdzieś w środku
        {
            przed->nastepny=q;
            q->nastepny=po;
        }
    }
}
```

Kolejne ważne, choć skrajnie nieskomplikowane, metody są niemalże identyczne koncepcyjnie. W celu znalezienia w liście pewnego elementu `x` należy przejrzeć ją za pomocą zwykłej pętli `while`:

```
int LISTA::szukaj(int x)
{
    ELEMENT *q=inf.glowa;
    while (q != NULL)
    {
        if (q->wartosc==x)
            return SUKCES;
        q=q->nastepny;
    }
    return PORAZKA;
}
```

Identyczną strukturę posiada metoda `wypisz`, służąca do wypisywania zawartości listy:

```
void LISTA::wypisz()
{
    ELEMENT *q=inf.glowa;
```

```

if (pusta()) cout << "(lista pusta)";
else
    while (q != NULL)
    {
        cout << q->wartosc << " ";
        q=q->nastepny;
    }
cout << "\n";
}

```

Pora na nieco trudniejsze fragmenty kodu.

Zacniemy od operacji usuwania *ostatniego* elementu listy, do której to czynności zatrudniliśmy przededefiniowany operator dekrementacji (--). Tak jak już wcześniej wspominałem, dodatkowy parametr typu `int` wynika z wymogów normalizacyjnych języka C++, jest sztuczny i wyłącznie informuje kompilator o tym, że przeddefiniujemy w tej metodzie operator „przyrostkowy”.

Funkcja, która się za nim ukrywa, jest relatywnie prosta: jeśli na liście jest tylko jeden element, to modyfikacji ulegnie zarówno pole `glowa`, jak i pole `ogon` struktury informacyjnej. Oba te pola, po uprzednim usunięciu jedynego elementu listy, zostaną zainicjowane wartością `NULL`.

Nieco trudniejszy jest przypadek, gdy lista zawiera więcej niż jeden element. Należy wówczas odszukać *przedostatni* jej element, aby móc odpowiednio zmodyfikować wskaźnik `ogon` struktury informacyjnej. Znajomość przedostatniego elementu listy umożliwi nam łatwe usunięcie ostatniego jej elementu. Poniżej jest zamieszczony pełny tekst funkcji wykonującej to zadanie.

```

LISTA& LISTA::operator --(int) // int jest parametrem sztucznym
{
    // (operator -- będzie przyrostkowy)
    if (inf.glowa==inf.ogon) // jeden element (lub lista pusta)
    {
        delete inf.glowa;
        inf.glowa=inf.ogon=NULL;
    } else
    {
        ELEMENT *temp=inf.glowa;
        while ((temp->nastepny) != inf.ogon) // szukamy przedostatniego
            temp=temp->nastepny; // elementu listy...
        inf.ogon=temp;
        delete temp->nastepny; // ... i usuwamy go
        temp->nastepny=NULL;
    }
    return (*this); // zwracamy zmodyfikowany obiekt
}

```

Obiekt jest zwracany poprzez swój adres, czyli może posłużyć jako argument dowolnej dozwolonej na nim operacji. Przykładowo: możemy utworzyć wyrażenie `(l2--).wypisz()`. Mimo groźnego wyglądu działanie tej instrukcji jest trywialne: pierwsza dekrementacja zwraca prawdziwy, fizycznie istniejący obiekt, który jest poddawany od razu drugiej dekrementacji. Rezultat tej ostatniej — jako pełnoprawny obiekt — może aktywować dowolną metodę swojej klasy, czyli przykładowo sprawdzić swoją zawartość za pomocą funkcji `wypisz`.

Przy okazji omawiania operatora dekrementacji spójrzmy jeszcze na inne jego zastosowanie. W definicji klasy został zawarty jej destruktor. Przypomnijmy, że destruktor jest specjalną funkcją wywoływaną automatycznie podczas niszczenia obiektu. To niszczenie może być bezpośrednie, np. za pomocą operatora `delete`:

```

LISTA *p=new LISTA; // tworzymy nowy obiekt...
...
delete p; // ...i niszczymy go!

```

lub też pośrednie, w momencie gdy obiekt przestaje być dostępny. Przykładem tej drugiej sytuacji niech będzie następujący fragment programu:

```

if (warunek)
{
LISTA p; //tworzymy obiekt lokalny
...      //widoczny tylko w tej instrukcji if
}

```

Obiekt `p` zadeklarowany w ciele instrukcji `if` jest dla niej całkowicie lokalny. Żaden inny fragment programu nie ma prawa dostępu do niego. Z takim tymczasowym obiektem wiąże się czasem dość sporo pamięci zarezerwowanej tylko dla niego. Otóż gdyby nie było destruktoru, programista nie miałby wcale pewności, czy ta pamięć została w całości zwrócona systemowi operacyjnemu. Celowo podkreślam, że w całości, bowiem automatyczne zwalnianie pamięci jest możliwe tylko w przypadku tych zmiennych, które są z założenia lokowane na stosie. Dotyczy to np. zwykłych pól obiektu, ale nie jest możliwe w przypadku struktur dynamicznych, które są nierzadko rozsiane po dość sporym obszarze pamięci komputera. Tak jest w przypadku list, drzew, tablic dynamicznych, etc. W takim przypadku programista musi sam napisać jawny destruktor, który znając⁶ doskonale sposób, w jaki pamięć została przydzielona obiektowi, będzie ją umiał prawidłowo zwrócić.

Tak też się dzieje w naszym przykładzie. Destruktor ma zaskakująco prostą budowę:

```

~LISTA()
//destruktor, który używa przededefiniowanego operatora --
{
while (!pusta()) (*this)--;
}

```

Jest to zwykła pętla `while`, która tak długo usuwa elementy z listy, aż stanie się ona pusta. Mimo iż nie jest to optymalny sposób na zwolnienie pamięci, został jednak zastosowany w celu ukazania możliwych zastosowań wskaźnika `this`, który — jak wiemy — wskazuje na własny obiekt. Linia `(*this)--` oznacza dla danego obiektu wykonanie na sobie operacji dekrementacji. Obiekt ulegający z pewnych powodów destrukcji (typowe przypadki zostały wzmiankowane wcześniej) wywoła swój destruktor, który zaaplikuje na sobie tyle razy funkcję dekrementacji, aby całkowicie zwolnić pamięć wcześniej przydzieloną liście.

Kolejna porcja kodu do omówienia dotyczy redefinicji operatora `+` (plus). Naszym celem jest zbudowanie takiej funkcji, która umożliwi *dodawanie* list w jak najbardziej dosłownym znaczeniu tego słowa. Chcemy, aby w wyniku następujących instrukcji:

```

LISTA x,y,z; //tworzymy 3 puste listy.
x.dorzuc2(3); x.dorzuc2(2); x.dorzuc2(1);
y.dorzuc2(6); y.dorzuc2(5); y.dorzuc2(4);
z=x+y;

```

Lista wynikowa z zawierała wszystkie elementy list `x` i `y`, tzn.: 1, 2, 3, 4, 5 i 6 (posortowane!). Najprostszą metodą jest przekopiowanie wszystkich elementów z list `x` i `y` do listy `z` i jednocześnie aktywowanie na rzecz tej ostatniej metody `dorzuc2`. Zapewni to utworzenie listy już posortowanej:

```

LISTA& operator +(LISTA &x,LISTA &y)
{
LISTA *temp=new LISTA;
ELEMENT *q1=(x.inf).glowa; //wskaźniki robocze
ELEMENT *q2=(y.inf).glowa;
while (q1 != NULL) //przekopiowanie listy x do temp
{
temp->dorzuc2(q1->wartosc);
q1=q1->nastepny;
}
}

```

⁶ De facto to *my* go znamy i dzielimy się tą cenną wiedzą z destruktorom. Występujące tu i ówdzie personifikacje są nie do uniknięcia w tego typu opisach!

```

while (q2 != NULL) //przekopiowanie listy y do temp
{
    temp->dorzuc2(q2->wartosc);
    q2=q2->nastepny;
}
return (*temp);
}

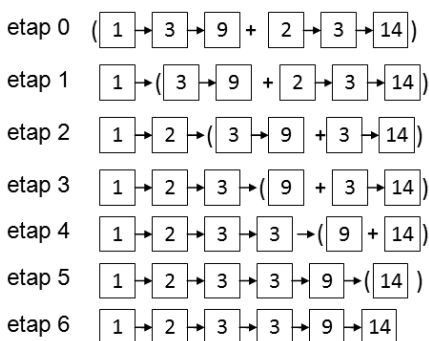
```

Czy jest to najlepsza metoda? Chyba nie, chociażby z uwagi na niepotrzebne dublowanie danych. Ideałem byłoby posiadanie metody, która wykorzystując fakt, iż listy są już posortowane⁷, dokona ich zespolenia ze sobą (tzw. *fuzji*) przy użyciu wyłącznie istniejących komórek pamięci, bez tworzenia nowych. Inaczej mówiąc, będziemy zmuszeni do manipulowania wyłącznie wskaźnikami i to jest jedyne narzędzie, jakie dostaniemy do ręki!

Na rysunku 5.7 możemy przykładowo prześledzić, jak powinna być wykonywana fuzja pewnych dwóch list $x=(1,3,9)$ i $y=(2,3,14)$, tak aby w jednej z nich znalazły się wszystkie elementy x i y — oczywiście posortowane (w naszym przykładzie niemalejąco).

Rysunek 5.7.

Fuzja list na przykładzie



Najmniejszym z dwóch pierwszych elementów list jest 1 i on też będzie stanowił początek nowej listy. Następnikiem tego elementu będzie fuzja dwóch list: $x' = (3, 9)$ i $y = (2, 3, 14)$.

Jak dokonać fuzji list x' i y ? Dokładnie tak samo: bierzemy element 2, który jest najmniejszym z dwóch pierwszych elementów list x' i y ... Można tak *rekurencyjnie* kontynuować aż do momentu, gdy natrafimy na przypadki elementarne: jeśli jedna z list jest pusta, to fuzją ich obu będzie oczywiście ta druga lista. Na tej zasadzie jest skonstruowana procedura `fuzja(ob1, ob2)`, która wywołana z dwoma parametrami `ob1` i `ob2` zwróci w liście `ob1` sumę elementów list `ob1` i `ob2`. Lista `ob2` jest w wyniku tej operacji zerowana, choć jej całkowite usunięcie pozostaje ciągle w gestii programisty (taki jest nasz wybór — równie dobrze można by to zrobić od razu).

Nasze zadanie wykonamy w dość nietypowy dla C++ sposób, który ma na celu ukazanie zakresu możliwych zastosowań tzw. *funkcji zaprzyjaźnionych* z klasą. Przypomnijmy, iż są to funkcje (lub procedury), które — nie będąc metodami danej klasy — mają dostęp do zastrzeżonych pól `private` i `protected` obiektu, którego adres został im przekazany jako jeden z parametrów wywołania. Ponieważ nie są to metody, nie mogą być wywoływane w ramach notacji z kropką, a ponadto obiekt, na który mają działać, musi im zostać przekazany w sposób jawny — na przykład poprzez swój adres.

Fuzję list wykonamy w dwóch etapach. Najpierw przygotujemy prostą funkcję, która — otrzymując dwie posortowane listy a i b — zwróci jako wynik listę będącą ich fuzją. Rekurencyjny zapis tego procesu jest bardzo prosty i zbliżony stylem do rozwiązywania problemów listowych w takich językach jak LISP lub PROLOG:

⁷ Zakładamy tym samym użycie metody `dorzuc2` podczas tworzenia listy.

```

ELEMENT *sortuj(ELEMENT *a, ELEMENT *b)
{
    if (a==NULL)
        return b;
    if (b==NULL)
        return a;
    if (a->wartosc<=b->wartosc)
    {
        a->nastepny=sortuj(a->nastepny,b);
        return a;
    }else
    {
        b->nastepny=sortuj(b->nastepny,a);
        return b;
    }
}

```

Dysponując już funkcją `sortuj`, możemy zastosować ją w procedurze `fuzja`. Będąc zaprzyjaźnioną z klasą `LISTA`, może ona dowolnie manipulować prywatnymi komponentami list x i y , które zostały jej przekazane w wywołaniu⁸.

```

void fuzja(LISTA &x,LISTA &y)
{ // a i b muszą być posortowane
    ELEMENT *a=x.inf.glowa,*b=y.inf.glowa;
    ELEMENT *wynik=sortuj(a,b);
    x.inf.glowa=wynik;
    if(x.inf.ogon->wartosc <= y.inf.ogon->wartosc)
        x.inf.ogon=y.inf.ogon;
    else x.inf.ogon=x.inf.ogon;
        y.zeruj();
}

```

Celowo znacznie rozbudowana funkcja `main` ilustruje sposób korzystania z opisanych wyżej funkcji. Do obu list są dołączane elementy tablic, następnie ma miejsce testowanie niektórych metod oraz sortowanie dwóch list poprzez ich fuzję.

```

int main()
{
    LISTA l1,l2;
    const int n=6;
    int tab1[n]={2,5,-11,4,14,12};
    // każdy element tablicy zostanie wstawiony do listy
    cout << "\nL1 = ";
    for (int i=0; i<n; l1.dorzuc2(tab1[i++]));
        l1.wypisz();
    int tab2[n]={9,6,77,1,7,4};
    cout << "\nL2 = ";
    for (int i=0; i<n; l2.dorzuc2(tab2[i++]));
        l2.wypisz();
    cout << "Efekt poszukiwań liczby 14 w l1: " << l1.szukaj(14) << endl;
    cout << "Efekt poszukiwań liczby 0 w l1: " << l1.szukaj(0) << endl;
    cout << "Oto lista będąca sumą dwóch poprzednich\nL3 = ";
        LISTA l3=l1+l2;
        l3.wypisz();
    cout << "Listy L1 i L2 pozostały bez zmian:\nL1 = ";
        l1.wypisz();
    cout << "\nL2 = ";
        l2.wypisz();
    cout << "Lista L1 bez dwóch ostatnich elementów:\nL1 = ";
        (l1--)--.wypisz();
    cout << "Efekt fuzji L1 z L2:\n";
        fuzja(l1,l2);
}

```

⁸ Patrz też dodatek A, w którym zostało omówione na innym przykładzie pojęcie funkcji zaprzyjaźnionej.

```

cout << "L1 = ";
l1.wypisz();
cout << "L2 = ";
l2.wypisz();
l1.dorzuc2(80);l1.dorzuc2(8);
cout << "dorzucamy do L1 liczby 80 i 8\nL1 = ";
l1.wypisz();
}

```

Oto wyniki uruchomienia programu:

```

L1 = -11  2  4  5  12  14
L2 = 1  4  6  7  9  77
Efekt poszukiwań liczby 14 w l1: 1
Efekt poszukiwań liczby 0 w l1: 0
Oto lista będąca sumą dwóch poprzednich
L3 = -11  1  2  4  4  5  6  7  9  12  14  77
Listy L1 i L2 pozostały bez zmian:
L1 = -11  2  4  5  12  14
L2 = 1  4  6  7  9  77
Lista L1 bez dwóch ostatnich elementów:
L1 = -11  2  4  5
Efekt fuzji L1 z L2:
L1 = -11  1  2  4  4  5  6  7  9  77
L2 = (lista pusta)
dorzucamy do L1 liczby 80 i 8
L1 = -11  1  2  4  4  5  6  7  8  9  77  80

```

Listy jednokierunkowe — teoria i rzeczywistość

Oprócz pięknie brzmiących rozważań teoretycznych istnieje jeszcze twarda rzeczywistość, w której mają wykonywać się nasze pieczołowicie przygotowane programy⁹.

Spójrzmy obiektywnie na listy jednokierunkowe pod kątem ich wad i zalet:

- ◆ *Wady*: nienaturalny dostęp do elementów, niełatwe sortowanie, utrudniona analiza zawartości i ocena wielkości listy.
- ◆ *Zalety*: efektywne zużycie pamięci, elastyczność.

Przeanalizujmy szczególnie uważnie zagadnienie sortowania danych, będących elementami listy. Wyobraźmy sobie zapewne, że posortowanie w pamięci struktury danych, która nie jest w niej rozłożona liniowo (tak jak ma to miejsce w przypadku tablicy), jest dość złożone.

Lista, do której nowe elementy są wstawiane, już na samym początku konsekwentnie w określonym porządku służy, oprócz swojej podstawowej roli gromadzenia danych, także do ich porządkowania. Jest to piękna właściwość: sama struktura danych dba o sortowanie! W sytuacji, gdy istnieje tylko jedno kryterium sortowania (np. w kierunku wartości niemalejących pewnego pola x), możemy mówić o ideale. Cóż jednak mamy począć, gdy elementami listy są rekordy o bardziej skomplikowanej strukturze, np.:

```

struct
{
char imie[20];
char nazwisko[30];
int wiek;
int kod_pracownika;
}

```

Raz możemy zechcieć dysponować taką listą uporządkowaną alfabetycznie, wg nazwisk, innym razem będzie nas interesował wiek pracownika itd. Czy należy w takim przypadku dysponować dwiema wersjami tych list — co pochłania cenną pamięć komputera — czy też może

⁹ Wbrew wszelkim przesłankom nie jest to definicja systemu operacyjnego.

zdecydujemy się na sortowanie listy w pamięci? Jednak uwaga: to drugie rozwiązanie zajmie z kolei cenny czas procesora!

Poruszony powyżej problem był na tyle charakterystyczny dla wielu rzeczywistych programów, że zostało do jego rozwiązania wymyślone pewne sprytne rozwiązanie, które postaram się dość szczegółowo omówić.

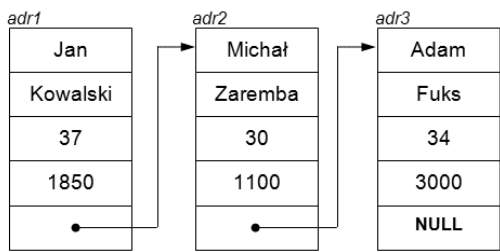
Pomysł polega na uproszczeniu i skomplikowaniu zarazem tego, co poznaliśmy wcześniej. Uproszczenie polega na tym, że *rekordy zapamiętywane w liście nie są w żaden sposób wstępnie sortowane*. Inaczej mówiąc, do zapamiętywania możemy użyć odpowiednika jakże prostej funkcji `dorzuc1` (patrz strona 97). Słowo „odpowiednik” pasuje tutaj najlepiej, bowiem niezbędne okaże się wprowadzenie kilku kosmetycznych zabiegów związanych z ogólną zmianą koncepcji.

Obok listy danych będziemy ponadto dysponować kilkoma listami wskaźników do nich. List tych będzie tyle, ile sobie zażyczymy kryteriów sortowania.

Jak nietrudno się domyślić, jeśli nie zamierzamy sortować listy danych (a jednocześnie chcemy mieć dostęp do danych posortowanych!), to podczas wstawiania nowego adresu do którejś z list wskaźników musimy dokonać jej sortowania. Zadanie jest zbliżone do tego, które wykonywała funkcja `dorzuc2`, z tą tylko różnicą, że dostęp do danych nie odbywa się w sposób bezpośredni.

Podczas sortowania list wskaźników dane nie są w ogóle ruszane — przemieszczaniu w listach będą ulegały wyłącznie same wskaźniki! Na tym etapie ma prawo to wszystko brzmieć dość enigmatycznie, pora zatem na jakiś konkretny przykład. Popatrzmy w tym celu na rysunek 5.8.

Rysunek 5.8.
Sortowanie listy bez przemieszczania jej elementów (1)

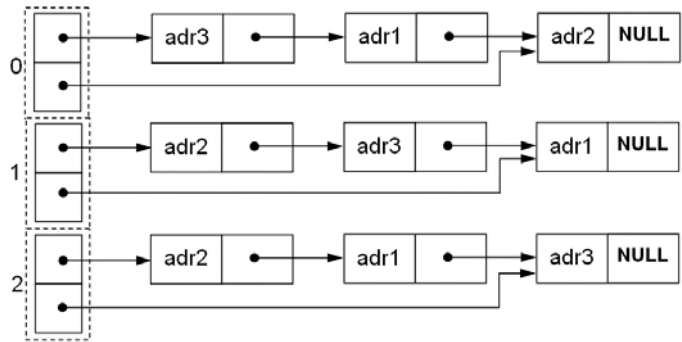


Lista DANE

Zawiera on listę o nazwie DANE, zbudowaną z kilku rekordów, które stanowią zaczątek miniatury bazy danych o pracownikach pewnego przedsiębiorstwa. Przyjmijmy dla uproszczenia, że jedyne istotne informacje, które chcemy zapamiętać, to: imię, nazwisko, pewien kod i oczywiście zarobek. Na rysunku są zaznaczone symbolicznie adresy rekordów: `adr1`, `adr2` i `adr3`, przydzielone przez funkcję `dorzuc1`.

Rysunek 5.9 zawiera już kilka nowości w porównaniu z tym, co mieliśmy okazję do tej pory poznać.

Rysunek 5.9.
Sortowanie listy bez przemieszczania jej elementów (2)



Lista TAB_PTR

Tablica `TAB_PTR` zawiera rekordy informacyjne (tzn. wskaźniki głowa i ogon) do list złożonych z adresów rekordów z listy `DANE` — w naszym przypadku zakładamy 3 listy wskaźników i będą one oczywiście zawierać adresy `adr1`, `adr2` i `adr3` (chwilowo na liście znajdują się trzy elementy; w miarę dokładania nowych elementów do listy `DANE` będą ulegały odpowiedniemu wzrostowi listy wskaźników).

Rozmiar tablicy `TAB_PTR` jest równy liczbie kryteriów sortowania: patrząc od góry, możemy zauważyć, że listy są posortowane kolejno wg nazwiska, kodu i zarobków.

Podsumujmy informacje, które można odczytać z rysunków 5.8 i 5.9:

- ◆ Nieposortowana baza danych, która jest zapamiętana w liście o nazwie `DANE`, zawiera w danym momencie 3 rekordy.
- ◆ Tablica wskaźników `TAB_PTR` zawiera 3 rekordy informacyjne (poznane już poprzednio), których pola `głowa` i `ogon` umożliwiają dostęp do trzech list wskaźników. Każda z tych list jest posortowana wg innego kryterium sortowania:
 - ◆ Lista wskazywana przez `TAB_PTR[0]` jest posortowana alfabetycznie wg nazwisk pracowników (Fuks, Kowalski i Zaremba).
 - ◆ Analogicznie `TAB_PTR[1]` klasyfikuje pracowników wg pewnego kodu używanego w tej fabryce (Zaremba, Fuks i Kowalski).
 - ◆ Ostatnia lista, `TAB_PTR[2]`, grupuje pracowników wg ich zarobków.

Poniżej jest przedstawiona nowa wersja klasy `LISTA`, uwzględniająca już propozycje przedstawione na rysunku 5.8. Aby umożliwić sensowną prezentację w postaci programu przykładowego, pewnemu uproszczeniu uległa struktura danych zawierająca informacje o pracowniku: ograniczymy się tylko do nazwiska i zarobków. (Rozbudowa tych struktur danych nie wniosłaby koncepcyjnie nic nowego, natomiast zagmatwałaby i tak dość pokazyńco objętościowo listing).

Struktury danych prezentują się w nowej wersji następująco:

```
typedef struct rob
{
    char nazwisko[100];
    long zarobek;
    struct rob *nastepny;    // wskaźnik do
                           // następnego elementu
}ELEMENT;

typedef struct rob_ptr      // struktura robocza listy
{                           // wskaźników
    ELEMENT *adres;
    struct rob_ptr *nastepny;
}LPTR;
```

Olbrzymich zmian jak na razie nie ma i uważny Czytelnik mógłby się słusznie zapytać, dlaczego nie zostały wykorzystane mechanizmy dziedziczenia, aby maksymalnie wykorzystać już napisany kod? Powód jest prosty: poprzednia wersja klasy `LISTA` służyła w zasadzie do ukazania mechanizmów i podstawowych algorytmów związanych z listami jednokierunkowymi; jej zastosowanie praktyczne było w związku z tym raczej nikłe i użycie jej jako klasy bazowej byłoby nieco sztuczne. W C++ klasy bazowe powinny być jak najbardziej uniwersalne, tak aby ich użycie do dziedziczenia nie miało sztucznego charakteru.

Obecnie prezentowana wersja struktury listy jednokierunkowej charakteryzuje się bardzo dużą elastycznością użytkowania i to właśnie ona mogłaby ewentualnie posłużyć jako klasa bazowa w dalszej hierarchii dziedziczenia (o ile Czytelnik w istocie będzie w ogóle potrzebował mechanizmów dziedziczenia, mechanizm ten nie powinien być stosowany jako sztuka dla sztuki).

Oto nowa wersja klasy LISTA:



lista2.h

```
const int n=5;           // rozmiar tablic przykładowych
const int kryteria_sort=2; // liczba kryteriów sortowania

typedef struct rob
{
    char nazwisko[100];
    long zarobek;
    struct rob *nastepny; // wskaźnik do
} ELEMENT;               // następnego elementu

typedef struct rob_ptr    // struktura robocza listy
{
    ELEMENT *adres;
    struct rob_ptr *nastepny;
} LPTR;
class LISTA
{
public:
    LISTA();               // konstruktor
    ~LISTA();              // destruktor
    void dorzuc(ELEMENT *); // dołącz nowy element q
    void wypisz(char);      // wypisz zawartość listy
    int usun(ELEMENT*, int(*decyzja)(ELEMENT *, ELEMENT*));
    // usuń element, który jest zgodny z wzorcową komórką podaną
    // jako parametr
private: // prywatne struktury:

    typedef struct        // struktura informacyjna listy danych
    {
        ELEMENT *głowa;
        ELEMENT *ogon;
    }
    INFO;
    typedef struct        // struktura informacyjna listy wskaźników
    {
        LPTR *głowa;
        LPTR *ogon;
    } LPTR_INFO;

    LPTR_INFO inf_ptr[kryteria_sort];
    INFO info_dane;
    // metody „wewnętrzne”, niedostępne publicznie:
    LPTR_INFO *odszukaj_wsk(LPTR_INFO*, ELEMENT*,
        int*)(ELEMENT*, ELEMENT*));
    ELEMENT *usun_wsk(LPTR_INFO*, ELEMENT*, int*)(ELEMENT*, ELEMENT*));
    int usun_dane(ELEMENT*);
    void dorzuc2(int, ELEMENT*, int(*decyzja)(ELEMENT*, ELEMENT*));
    void wypisz1(LPTR_INFO*);
};
```

Tajemnicze metody prywatne, podane wyżej bez żadnego opisu, zostaną szczegółowo omówione w następnych paragrafach.

Analizując procedury i funkcje do obsługi list, można zauważyć, że operacje odszukiwania pewnego elementu wg podanego wzorca (np. „odszukaj pracownika, który zarabia 1 200 zł”) i wyszukiwania miejsca na wstawienie nowego elementu różniły się nieznacznie. Od tego spostrzeżenia do gotowej realizacji programowej jest już tylko jeden krok. Aby go zrobić, musimy dobrze zrozumieć zasady operowania wskaźnikami do funkcji¹⁰ w C++, bowiem ich użycie pozwoli na

¹⁰ Miłośnicy i znawcy języka LISP mogą opuścić ten paragraf.

eleganckie rozwiązanie kilku problemów. Zdając sobie sprawę, że wskaźniki do funkcji są relatywnie rzadko stosowane, niezbędne wydało mi się przypomnienie sposobu ich stosowania w C++. Jest to ułkon głównie w stronę programistów pascalowych, bowiem w ich ulubionym języku ten mechanizm w ogóle nie istnieje.

Przedstawiony poniżej przykład ilustruje sposób użycia wskaźników do funkcji w C++.



wsk_fun.cpp

```
int do_2(int a)
{
    return a*a;
}
int do_4(int a)
{
    return a*a*a*a;
}

int wzor(int x,int(*fun)(int))
{
    return fun(x);
}
int main()
{
    cout << "10 do potęgi 2:"<< wzor(10,do_2) << endl;
    cout << "10 do potęgi 4:"<< wzor(10,do_4) << endl;
}
```

Funkcja `wzor` zwraca — w zależności od tego, czy zostanie wywołana jako `wzor(10,do_2)` czy też `wzor(10,do_4)` — odpowiednio: 100 lub 10000. Mamy tu do czynienia z podobnym fenomenem, jak w przypadku tablic, gdzie nazwa (tablicy, funkcji) jest jednocześnie wskaźnikiem do niej. Bezpośrednią konsekwencją jest dość naturalny sposób użycia, pozwalający na uniknięcie typowych dla C++ operatorów `*` (operator „wyluskania” wartości) i `&` (operator adresowy).

Inny przykład: pewna procedura `f`, która otrzymuje jako parametr liczbę `x` (typu `int`) i wskaźnik do funkcji o nazwie `g` (zwracającej typ `double` i operującej trzema parametrami: `int`, `double`, i `char*`), może zostać zadeklarowana w następujący sposób:

```
void f(int x, double(*g)(int, double, char *))
{
    k=g(12,5.345,"1984");
    cout << k << endl;
}
```

Zakres stosowania wskaźników do funkcji jest dość szeroki i przyczynia się do uogólnienia wielu procedur i funkcji.

Powróćmy teraz do odsuniętych chwilowo na bok list i zajmijmy się problemem wstawiania nowego elementu do listy uprzednio posortowanej. Chcemy znaleźć dwa adresy: przed i po (patrz rysunek 5.6 na stronie 98), które umożliwią nam takie zmodyfikowanie wskaźników, aby cała lista była widziana jako posortowana. W tym celu zmuszeni jesteśmy do użycia pętli `while` po-
 znej na stronie 99:

```
while((stan==SZUKAJ) && (po!=NULL))
    if (po->zarobek >= x)
        stan=ZAKONCZ;
    else
    {
        przed=po;
        po=po->nastepny;
    }
```

Gdybyśmy zaś chcieli usunąć pewien element listy, który spełnia przykładowo warunek, że pole zarobek wynosi 1200 zł, to również będą nam potrzebne wskaźniki przed i po. Odnajdziemy je w sposób następujący:

```
while((stan==SZUKAJ) && (po!=NULL))
    if (po->zarobek == 1200)
        stan=ZAKONCZ;
    else
    {
        przed=po;
        po=po->nastepny;
    }
```

Różnica pomiędzy tymi dwiema pętlami `while` tkwi wyłącznie w warunku instrukcji `if-else`. Idea naszego rozwiązania jest zatem następująca: napiszemy uniwersalną funkcję, która posłuży do odszukiwania wskaźników przed i po w celu ich późniejszego użycia do dokładania elementów do listy, jak również do ich usuwania. Funkcja ta powinna nam zwrócić oba wskaźniki — posłużymy się do tego celu strukturą `LPTR_INFO` (patrz listing LISTA2.H), umawiając się, że pole `głowa` będzie odpowiadało wskaźnikowi przed, a pole `ogon` — wskaźnikowi po.

Łatwo jest zauważyć, że operacje poszukiwania, wstawiania, etc. rozpoczynamy od listy wskaźników, z której zdobędziemy adres rekordu danych (adres ten zostanie zapamiętany w polu adres struktury `LPTR`, która stanowi element składowy listy wskaźników — patrz rysunek 5.9). Dopiero po zmodyfikowaniu wszystkich list wskaźników (a może ich być tyle, ile przyjmujemy kryteriów sortowania) należy zmodyfikować listę danych. Pracy jest — jak widać — mnóstwo, ale jest to cena za wygodę późniejszego użytkowania takiej listy! Pocieszeniem niech będzie fakt, że po jednokrotnym napisaniu odpowiedniego zestawu funkcji bazowych będziemy mogli z nich później wielokrotnie korzystać bez konieczności przypominania sobie, jak one to robią. Przejdźmy już do opisu realizacji funkcji `odszukaj_wsk`, która zajmie się poszukiwaniem wskaźników przed i po, zwracając je w strukturze `LPTR_INFO`:

- ♦ Wskaźnik `inf` do struktury informacyjnej listy wskaźników; adres początku znajduje się w polu `głowa`, a adres końca w polu `ogon`.
- ♦ Wskaźnik `q` do pewnego fizycznie istniejącego rekordu danych. Jest to albo nowy rekord, który chcemy dołączyć do listy, albo po prostu pewien szablon poszukiwań.
- ♦ Wskaźnik decyzyja do funkcji porównawczej, która zostanie włożona do instrukcji `if` w pętli `while`.



lista2.cpp

```
LISTA::LPTR_INFO* LISTA::odszukaj_wsk(LISTA::LPTR_INFO *inf,ELEMENT *q,
                                     int(*decyzja)(ELEMENT *q1,ELEMENT *q2))
{
    LPTR_INFO *res=new LPTR_INFO;
    res->głowa=res->ogon=NULL;
    if (inf->głowa==NULL)
        return (res); // lista pusta!
    else
    {
        LPTR *przed,*pos;
        przed=NULL;
        pos=inf->głowa;
        enum {SZUKAJ,ZAKONCZ} stan=SZUKAJ;
        while ((stan==SZUKAJ) && (pos!=NULL))
            if (decyzja(pos->adres,q))
                stan=ZAKONCZ; // znaleźliśmy miejsce, w którym element
            else // istnieje (albo ma być wstawiony)
            { // przemieszczamy się w poszukiwaniach
```

```

        przed=pos;
        pos=pos->nastepny;
    }
    res->glowa=przed;
    res->ogon=pos;
    return (res);
}

```

Przykładowo, jeśli chcemy odszukać i usunąć pierwszy rekord, który w polu nazwisko zawiera „Kowalski”, to należy stworzyć tymczasowy rekord, który będzie miał odpowiednie pole wypełnione tym nazwiskiem (pozostałe nie będą miały wpływu na poszukiwanie):

```

ELEMENT *f=new ELEMENT;
strcpy(f->nazwisko,"Kowalski");

```

Podobna uwaga należy się pozostałym kryteriom poszukiwań — wg zarobków, imienia, etc. Jeśli poszukiwanie zakończy się sukcesem, to w polu ogon zostanie zwrócony adres fizycznie istniejącego rekordu, który odpowiadał wzorcowi naszych poszukiwań. W przypadku gdyby element taki nie istniał, powinny zostać zwrócone wartości NULL. Znajomość wskaźników przed i po umożliwi nam zwolnienie komórek pamięci zajmowanych dotychczas przez rekord danych, jak również odpowiednie zmodyfikowanie całej listy, tak aby wszystko było na swoim miejscu.

Innym przykładem zastosowania funkcji jest dołączenie nowego elementu do listy. Trzeba wówczas stworzyć nowy rekord, wypełnić jego pola i umieścić go na końcu listy. Następnie należy adres tego elementu wstawić do list wskaźników posortowanych wg zarobków, nazwisk czy też dowolnych innych kryteriów. W każdej z tych list miejsce wstawienia będzie inne, czyli za każdym razem różne mogą być wartości wskaźników przed i po, które zwróci funkcja `odszukaj_wsk`.

Zastosowanie funkcji `odszukaj_wsk` jest, jak widać, bardzo wszechstronne. Taka elastyczność możliwa była do osiągnięcia tylko i wyłącznie poprzez użycie wskaźników do funkcji — we właściwym miejscu i o właściwej porze.

Oto garść funkcji decyzyjnych, które mogą zostać użyte jako parametr:



lista2.h

```

int alfabetycznie(ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 są uporządkowane alfabetycznie?
    return (strcmp(q1->nazwisko,q2->nazwisko)>=0);
}
int wg_zarobkow(ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 są uporządkowane wg zarobków?
    return (q1->zarobek>=q2->zarobek);
}
int ident_nazwiska (ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 mają identyczne nazwiska?
    return (strcmp(q1->nazwisko,q2->nazwisko)==0);
}
int ident_zarobki (ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 mają identyczne zarobki?
    return (q1->zarobek==q2->zarobek);
}

```

Dwie pierwsze funkcje z powyższej listy służą do jej porządkowania, pozostałe ułatwiają proces wyszukiwania elementów. Oczywiście w rzeczywistej aplikacji bazy danych o pracownikach analogiczne funkcje byłyby nieco bardziej skomplikowane — wszystko zależy od tego, jakie kryteria wyszukiwania lub porządkowania zamierzamy zaprogramować oraz jak skomplikowane struktury danych wchodzą w grę.

Po tak rozbudowanych wyjaśnieniach działanie funkcji `odszukaj_wsk` nie powinno stanowić już dla nikogo tajemnicy.

Na 95 stronie mieliśmy okazję zapoznać się z funkcją pusta informującą, czy lista danych coś zawiera. Nic nie stoi na przeszkodzie, aby do kompletu dołożyć jej kolejną wersję, badającą w analityczny sposób listę wskaźników¹¹:

```
inline int pusta(LPTR_INFO *inf)
{
    return (inf->glowa==NULL);
}
```

Ponieważ użyliśmy dwukrotnie tej samej nazwy funkcji, nastąpiło w tym momencie jej *przeciążenie*; podczas wykonywania programu właściwa jej wersja zostanie wybrana w zależności od typu parametru, z którym zostanie wywołana (wskaźnik do struktury `INFO` lub wskaźnik do struktury `LPTR_INFO`).

Mając już komplet funkcji `pusca`, zestaw funkcji decyzyjnych i uniwersalną funkcję `odszukaj_wsk`, możemy pokusić się o napisanie brakującej procedury `dorzuc1`, która będzie służyła do dołączania nowego rekordu do listy danych z jednoczesnym sortowaniem list wskaźników. Załóżmy, że będą tylko dwa kryteria sortowania danych, co implikuje, iż tablica zawierająca „wskaźniki do list wskaźników” będzie miała tylko dwie pozycje (patrz rysunek 5.9).

Adres tej tablicy, jak również wskaźniki do listy danych i do nowo utworzonego elementu, zostaną obowiązkowo przekazane jako parametry:

```
void LISTA::dorzuc(ELEMENT *q)
{
    // rekord dołączamy bez sortowania
    if (info_dane.glowa==NULL) // lista pusta
        info_dane.glowa=info_dane.ogon=q;
    else // coś jest w liście
    {
        (info_dane.ogon->nastepny=q;
        info_dane.ogon=q;
    }
    // dołączamy wskaźnik do rekordu do listy posortowanej alfabetycznie
    dorzuc2(0,q,alfabetycznie);
    // dołączamy wskaźnik do rekordu do listy posortowanej wg zarobków
    dorzuc2(1,q,wg_zarobkow);
}
```

Funkcja jest bardzo prosta, głównie z uwagi na tajemniczą procedurę o nazwie `dorzuc2`. Oczywiście nie jest to jej poprzedniczka ze strony 99, choć różni się od tamtej poprawdy niewiele:

```
void LISTA::dorzuc2(int nr,ELEMENT *q,int(*decyzja)(ELEMENT *q1, ELEMENT *q2))
{
    LPTR *wsk=new LPTR;
    wsk->adres=q; // wpisujemy adres rekordu q
    // Poszukiwanie właściwej pozycji na wstawienie elementu
    if (inf_ptr[nr].glowa==NULL) // pusta
    {
        inf_ptr[nr].glowa=inf_ptr[nr].ogon=wsk;
        wsk->nastepny=NULL;
    }
    else //szukamy miejsca na wstawienie
    {
        LPTR *przed,*po;
        LPTR_INFO *gdzie=odszukaj_wsk(&inf_ptr[nr].q,decyzja);
```

¹¹ Funkcja typu `inline` oznacza tak naprawdę jej wywołanie w formie makra, tj. w miejsce jej wystąpienia zostanie wstawiony cały kod w niej zawarty, wydłużając kod wynikowy. Zaletą `inline` jest znaczące polepszenie czytelności programu, w którym zbiory prymitywnych grup funkcji (np. wielokrotnych przypisań) są zastępowane jednym „aliasem”. Ponadto nie jest to wywołanie funkcyjne, dzięki czemu unikamy całego bagażu związanego z obsługą przez stos.

```

przed=gdzie->glowa;
po=gdzie->ogon;
if (przed==NULL) // wstawiamy na początek listy
{
    inf_ptr[nr].glowa=wsk;
    wsk->nastepny=po;
} else
    if (po==NULL) // wstawiamy na koniec listy
    {
        inf_ptr[nr].ogon->nastepny=wsk;
        wsk->nastepny=NULL;
        inf_ptr[nr].ogon=wsk;
    } else // wstawiamy gdzieś „w środku”
    {
        przed->nastepny=wsk;
        wsk->nastepny=po;
    }
}
}

```

W celu zrozumienia dokonanych modyfikacji właściwe byłoby porównanie obu wersji funkcji `dorzuc2`, aby wykryć różnice, które między nimi istnieją. „Filozoficznie” nie ma ich wiele — w miejsce sortowania danych sortujemy po prostu wskaźniki do nich.

Funkcja zajmująca się usuwaniem rekordów wymaga przesłania m.in. fizycznego adresu elementu do usunięcia. Mając tę informację, należy wyczyścić zarówno listę danych, jak i listy wskaźników:

```

int LISTA::usun(ELEMENT *q, int(*decyzja)(ELEMENT *q1, ELEMENT *q2))
{
    // usuwa całkowicie informacje z obu list (wskaźników i danych)
    ELEMENT *ptr_dane;
    for (int i=0; i<kryteria_sort; i++)
        ptr_dane=usun_wsk(&inf_ptr[i].q,decyzja);
    if (ptr_dane==NULL)
        return(0);
    else
        return usun_dane(ptr_dane);
}

```

Funkcja `usun_wsk` zajmuje się usuwaniem wskaźników danego elementu z list wskaźników — jakakolwiek byłaby ich liczba. Czytelnik może zauważyć z łatwością, że raz jeszcze mamy tu do czynienia z bardzo podobnym do poprzednich schematem algorytmu.

Można nawet odważyć się na stwierdzenie, że listing jest zamieszczany wyłącznie gwoili formalności! Elementarna kontrola błędów jest zapewniana przez wartość zwracaną przez funkcję: w normalnej sytuacji winien to być różny od `NULL` adres fizycznego rekordu przeznaczonego do usunięcia.

```

ELEMENT* LISTA::usun_wsk(LPTR_INFO *inf, ELEMENT *q, int(*decyzja)(ELEMENT *q1, ELEMENT *q2))
{
    if (inf->glowa==NULL) // lista pusta, czyli nie ma czego usuwać!
        return NULL;
    else // szukamy elementu do usunięcia
    {
        LPTR *przed,*pos;
        LPTR_INFO *gdzie=odszukaj_wsk(inf,q,decyzja);
        przed=gdzie->glowa;
        pos=gdzie->ogon;
        if (pos==NULL)
            return NULL; // element nieodnaleziony

        if (pos==inf->glowa) // usuwamy z początku listy
            inf->glowa=pos->nastepny;
        else

```



```

        if (pos->nastepny==NULL) // usuwamy z końca listy
        {
            inf->ogon=przed;
            przed->nastepny=NULL;
        } else // usuwamy gdzieś „ze środka”
            przed->nastepny=pos->nastepny;
        ELEMENT *ret=pos->adres;
        delete pos;
        return ret;
    }
}

```

Funkcja `usun_dane` jest zbudowana wg podobnego schematu, co funkcja `usun_wsk`. Ponieważ przyjmowane jest założenie, że *element, który chcemy usunąć, istnieje*, programista musi zapewnić dokładną kontrolę poprawności wykonywanych operacji. Tak się dzieje w naszym przypadku — ewentualna nieprawidłowość zostanie wykryta już podczas próby usunięcia wskaźnika i wówczas usunięcie rekordu po prostu nie nastąpi.

```

int LISTA::usun_dane(ELEMENT *q)
{ // założenie: q istnieje!
    ELEMENT *przed,*pos;
    przed=NULL;
    pos=info_dane.glowa;
    while ((pos!=q) && (pos!=NULL)) // szukamy elementu „przed”
    {
        przed=pos;
        pos=pos->nastepny;
    }
    if (pos!=q)
        return(0); // element nieodnaleziony?!
    if (pos==info_dane.glowa) // usuwamy z początku listy
    {
        info_dane.glowa=pos->nastepny;
        delete pos;
    } else
        if (pos->nastepny==NULL) // usuwamy z końca listy
        {
            info_dane.ogon=przed;
            przed->nastepny=NULL;
            delete pos;
        } else // usuwamy gdzieś „ze środka”
        {
            przed->nastepny=pos->nastepny;
            delete pos;
        }
    return(1);
}

```

Pomimo wszelkich prób uczynienia powyższych funkcji bezpiecznymi kontrola w nich zastosowana jest ciągle bardzo uproszczona. Czytelnik, który będzie zajmował się implementacją dużego programu w C++, powinien bardzo dokładnie kontrolować poprawność operacji na wskaźnikach. Programy stają się wówczas co prawda mniej czytelne, ale jest to cena za mały, lecz jakże istotny szczegół: ich poprawne działanie.

Poniżej znajduje się rozbudowany przykład użycia nowej wersji listy jednokierunkowej. Jest to dość spory fragment kodu, ale zdecydowałem się na jego zamieszczenie, biorąc pod uwagę względne skomplikowanie omówionego materiału — ktoś nieprzyzwyczajony do sprawnego operowania wskaźnikami miał prawo się nieco zgubić. Zakładam zatem, że szczegółowy przykład zastosowania może mieć duże znaczenie dla ogólnego zrozumienia całości.

Dwie proste funkcje `wypisz1` i `wypisz` zajmują się eleganckim wypisaniem na ekranie zawartości bazy danych w kolejności narzuconej przez odpowiednią listę wskaźników:

```

void LISTA::wypisz1(LPTR_INFO *inf)
{ // wypisujemy zawartość posortowanej listy wskaźników (oczywiście nie interesuje nas
  LPTR *q=inf->glowa; // wypisanie wskaźników, gdyż są to adresy), lecz
  while (q != NULL) // informacji, na które one wskazują
  {
    cout << setw(9)<<q->adres->nazwisko<< " zarabia "<<setw(4)<<
      q->adres->zarobek<<"zł\n";
    q=q->nastepny;
  }
  cout << "\n";
}

void LISTA::wypisz(char c)
{
  if (c=='a') // alfabetycznie
    wypisz1(&inf_ptr[0]);
  else
    wypisz1(&inf_ptr[1]);
}

```

Funkcja main testuje wszystkie nowo poznane mechanizmy:

```

int main()
{
  LISTA l1;
  char *tab1[n]={ "Bec", "Becki", "Fikus", "Pertek", "Czerniak" };
  int tab2[n]={ 1300, 1000, 1200, 2000, 3000 };
  for (int i=0; i<n; i++)
  {
    ELEMENT *nowy=new ELEMENT; // tworzymy fizycznie nowy rekord...
    strcpy(nowy->nazwisko, tab1[i]);
    nowy->zarobek= tab2[i];
    nowy->nastepny=NULL;
    l1.dorzuc(nowy); // ...i dorzucamy go do listy
  }
  cout << "\n*** Baza danych posortowana alfabetycznie ***\n";
  l1.wypisz('a');
  cout << "*** Baza danych posortowana wg zarobków ***\n";
  l1.wypisz('z');
  ELEMENT *f=new ELEMENT;
  f->zarobek=2000;

  cout << "Wynik usunięcia rekordu pracownika zarabiającego 2000zł=
  "<< l1.usun(f, ident_zarobki) <<endl;
  delete f;
  cout << "*** Baza danych posortowana alfabetycznie ***\n";
  l1.wypisz('a');
  cout << "*** Baza danych posortowana wg zarobków ***\n";
  l1.wypisz('z');
}

```

Uruchomienie programu powinno dać następujące wyniki:

```

*** Baza danych posortowana alfabetycznie ***
    Bec zarabia 1300zł
    Becki zarabia 1000zł
    Czerniak zarabia 3000zł
    Fikus zarabia 1200zł
    Pertek zarabia 2000zł
*** Baza danych posortowana wg zarobków ***
    Becki zarabia 1000zł
    Fikus zarabia 1200zł
    Bec zarabia 1300zł
    Pertek zarabia 2000zł
    Czerniak zarabia 3000zł

```

```

Wynik usunięcia rekordu pracownika zarabiającego 2000zł=1
*** Baza danych posortowana alfabetycznie ***
    Bec zarabia 1300zł
    Becki zarabia 1000zł
    Czerniak zarabia 3000zł
    Fikus zarabia 1200zł
*** Baza danych posortowana wg zarobków ***
    Becki zarabia 1000zł
    Fikus zarabia 1200zł
    Bec zarabia 1300zł
    Czerniak zarabia 3000zł

```

Tablicowa implementacja list

Programowanie w C++ zmusza niejako programistę do dobrego poznania operacji na dynamicznych strukturach danych, sprawnego żonglowania wskaźnikami, etc. Nie da się ukryć, że nie wszyscy wskaźniki lubią. Przyczyn tej niechęci należy upatrywać głównie w próbach programowania na przykład struktur listowych bez pełnego zrozumienia tego, co się chce zrobić. Efekty najczęściej są opłakane, a winę w takich przypadkach ponosi rzecz jasna „chłopiec do bicia”, czyli sam język programowania. Tymczasem, podobnie zresztą jak i w życiu, to samo można zrobić na wiele sposobów — o czym niejednokrotnie zapominamy.

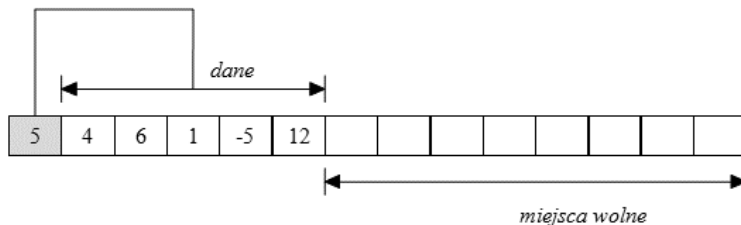
Tak też jest i z listami. Okazuje się, że istnieje kilka sposobów tablicowej implementacji list, niektóre z nich charakteryzują się nawet dość istotnymi zaletami, niemożliwymi do uzyskania w realizacji klasycznej (czyli tej, którą mieliśmy okazję poznać wcześniej). Olbrzymią wadą tablicowych wersji struktur listowych jest marnotrawstwo pamięci — przydzielamy przecież na stałe pewien obszar pamięci, powiedzmy dla 1 000 elementów — bo tyle w porywach będziemy potrzebowali miejsca. Gdyby natomiast nasz program używał listy o długości 200 elementów, to i tak obszar realnie zajmowany wynosiłby 1 000 (!). Jest to jednak cena nie do uniknięcia, płacimy ją za prostotę realizacji.

Klasyczna reprezentacja tablicowa

Jedną z najprostszych metod zamiany tablicy na listę jest umówienie się co do sposobu interpretacji jej zawartości. Jeśli powiemy sobie głośno (i nie zapomnimy o tym zbyt szybko), że i -temu indeksowi tablicy będzie odpowiadać i -ty element listy, to problem mamy prawie z głowy. To „prawie” wynika z tego, że trzeba się umówić, ile maksymalnie elementów zechcemy zapamiętać na liście. Oprócz tego konieczne będzie wybranie jakiejś zmiennej do zapamiętywania aktualnej liczby elementów wstawionych wcześniej do listy.

Ideę ilustruje rysunek 5.10, gdzie możemy zobaczyć tablicową implementację listy 5-elementowej złożonej z elementów: 4, 6, 1, -5 i 12:

Rysunek 5.10.
Tablicowa
implementacja listy



Programowa realizacja jest bardzo prosta — deklaracja przykładowej klasy nie powinna zawierać żadnych niespodzianek dla Czytelnika:



lista_tab.cpp

```
const int MaxTab=200;
class ListaTab
{
public:
    ListaTab() { tab[0]=0; } // konstruktor klasy
                          // metody zdefiniowane nieco dalej:
    void UsunElement(int k);
    void WstawElement(int x);
    void WstawElement(int x, int k);
    void WypiszListe();
private:
    int tab[MaxTab];        // tab[0] zarezerwowane!
};
```

Omówmy błyskawicznie wszystkie funkcje usługowe klasy. Przypuśćmy, że chcemy dysponować możliwością usunięcia k -tego elementu naszej listy. Po zbadaniu sensu takiej operacji (element musi istnieć!) wystarczy przesunąć zawartość tablicy o jeden w lewo od k -tej pozycji. Podczas przesuwania element nr k jest bezpowrotnie „zamazywany” przez swojego sąsiada:

```
void ListaTab::UsunElement(int k)
{ //usuwamy k-ty element listy, k >= 1
  if((k>=1) && (k<=tab[0]))
  {
    for(int i=k;i<tab[0];i++)
      tab[i]=tab[i+1];
    tab[0]--;
  }
}
```

Wariantów przedstawionej wyżej funkcji może być dość sporo. Mam nadzieję, że Czytelnik w miarę swoich specyficznych wymagań będzie mógł je sobie stworzyć i dostosować do konkretnych potrzeb.

Co jednak z *dołączaniem* elementów do listy? Poniżej są omówione dwie wersje odpowiedniej funkcji: pierwsza wstawia na koniec listy, druga na k -tą jej pozycję. Oczywiście w przypadku tej drugiej funkcji niezbędne jest dokonanie odpowiedniego przesunięcia zawartości tablicy, podobnie jak w metodzie UsunElement:

```
void ListaTab::WstawElement(int x)
{ //wstawiamy na koniec listy
  if(tab[0]<MaxTab-1)
    tab[++tab[0]]=x;
}
//-----
void ListaTab::WstawElement(int x,int k)
{ //wstawiamy na k-tą pozycję listy
  if((k>=1) && (k<=tab[0]+1) && (tab[0]<MaxTab-1))
  {
    for(int i=tab[0];i>=k;i--)
      tab[i+1]=tab[i]; //robimy miejsce
    tab[k]=x;
    tab[0]++;
  }
}
```

Zasady posługiwania się taką pseudolistą są już po stworzeniu wszystkich metod identyczne z zasadami pracy z prawdziwą listą jednokierunkową, dlatego też darujemy sobie cytowanie funkcji main.

Możliwe jest oczywiście takie zdefiniowanie klasy ListaTab, aby dołączanie elementów następowało już w porządku malejącym, rosnącym czy też wedle jakiegoś innego klucza — Czytelnik może odpowiednio rozbudować funkcje i metody w ramach nieskomplikowanego ćwiczenia.

Metoda tablic równoległych

W poprzednio poznanej implementacji list za pomocą zwykłej tablicy przypisaliśmy na sztywno *i*-temu elementowi tablicy *i*-ty element listy. W prostych zastosowaniach może to wystarczyć w zupełności, jednak rozwiązanie takie jest o wiele bardziej zbliżone ideowo do tablicy niż do listy. Prawdziwa lista powinna umożliwiać dość dowolne układanie elementów i sortowanie ich przy użyciu tylko i wyłącznie wskaźników. Chcieliśmy jednak od wskaźników, przydziałów pamięci, procedur `new` i `delete` uciec jak najdalej! Czyżby ich użycie było nieuniknione?

Odpowiedź na szczęście brzmi: NIE! Wszystko można w końcu zasymulować, więc czemu nie wskaźniki?! Popularna metoda polega na zadeklarowaniu tablicy rekordów składających się z pola informacyjnego `info` i pola typu całkowitego `nastepny`, które służy do odszukiwania elementu „następnego” na liście. Dobrze znane i klasyczne wręcz rozwiązanie. Idea jest przedstawiona na rysunku 5.11, gdzie można zobaczyć przykładową implementację listy służącej do przechowywania znaków, zawierającej w danym momencie pięć liter układających się w słowo „KOTEK”.

Rysunek 5.11.
Metoda „tablic równoległych” (1)



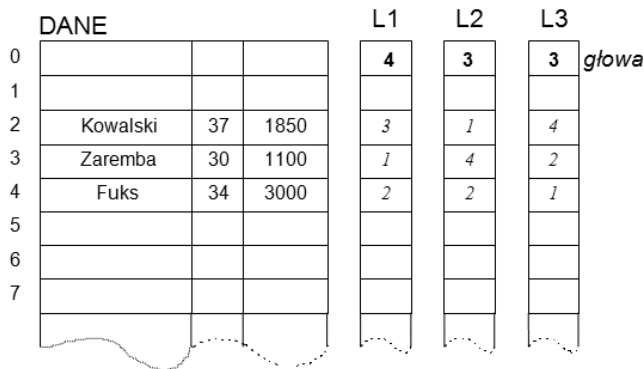
Pierwszy element tablicy (tzn. ten z pozycji 0) pełni rolę wskaźnika początku listy. Jest to zatem zmienna typu `głowa`. Jeśli oznaczymy tablicę jako `t`, to `t[0].nastepny` zawiera indeks pierwszego rzeczywistego elementu listy. W naszym przykładzie jest to 3, zatem w `t[3].info` znajduje się pierwszy element listy — jest nim znak K. Aby dowiedzieć się, co następuje po K, musimy odczytać `t[3].nastepny`. Jest to 2 i tam też jest umieszczona kolejna litera słowa „KOTEK” — etc. Koniec listy jest zaznaczany umownie poprzez wartość -1 w polu `nastepny`.

Rozwiązanie to można uznać za eleganckie i elastyczne. Dopisanie funkcji, które obsługują taką strukturę danych, nie jest trudne. Występuje tu pełna analogia pomiędzy już wcześniej przedstawionymi funkcjami (np. obsługującymi listy jednokierunkowe), dlatego też zadanie ewentualnego opracowania ich pozostawiam Czytelnikowi.

Należy przy okazji zwrócić uwagę na jedną niedogodność: mamy tu do czynienia z bardzo ścisłym połączeniem samej „gołej” informacji z komórkami, które symulują wskaźniki. O ile w przypadku list był to zabieg niezbędny, to przy wykorzystaniu tablic możemy bez wahania oddzielić te dwie rzeczy. Inaczej rzecz ujmując, dobrze byłoby dysponować osobną tablicą na dane i osobną na wskaźniki. Dlaczego jednak nie pójść dalej i nie używać kilku tablic na wskaźniki?! Zbliżylibyśmy się wówczas do wersji zaprezentowanej na rysunku 5.8, otrzymując jednak o wiele prostsze w realizacji zadanie.

Na rysunku 5.12 jest przedstawiona minibaza danych zgrupowana w wyodrębnionej tablicy danych.

Rysunek 5.12.
Metoda „tablic równoległych” (2)



Obok tablicy danych możemy zauważyć trzy osobne tablice wskaźników, które umożliwiają dostęp do danych widzianych jako listy posortowane wedle przeróżnych kryteriów. Tablica DANE zawiera rekordy z danymi, przy czym efektywne informacje zaczynają się od komórki dane[2] w górę. Dlaczego tak dziwnie? Otóż zabieg ten zapewnia nam odpowiedniość „1 do 1” tablicy danych i tablic wskaźników (L1, L2 i L3, które są w rzeczywistości zwykłymi tablicami liczb całkowitych).

W tych tablicach bowiem komórki nr 0 i nr 1 są zarezerwowane odpowiednio na: wskaźnik początku listy i znacznik końca. Należy to rozumieć w ten sposób, że L1[0] zawierający liczbę 4 informuje nas, iż dane[4] są pierwszym rekordem na liście. A jaki jest rekord następny? Oczywiście L1[4]=2, co oznacza, że drugim rekordem na liście danych jest dane[2]. Postępując tak dalej, odtwarzamy całą listę: dane[4], dane[2], dane[3] — łatwo zauważyć, że jest to lista posortowana alfabetycznie wg nazwisk. Skąd jednak wiemy, że dane[3] jest ostatnim rekordem na liście? Otóż L1[3] zawiera 1, co stanowi wg naszej umowy znacznik końca listy.

Analogicznie postępując, możemy odkryć, że L2 jest listą posortowaną wg kodów 2-cyfrowych, a L3 — wg zarobków. Tablicowa reprezentacja list, w której nastąpiło oddzielenie danych od wskaźników, pozwala na zapamiętanie w tym samym obszarze pamięci kilku list jednocześnie — o ile oczywiście ich elementy składowe w jakiś sposób się pokrywają. W aplikacjach, w których występuje taka sytuacja, jest to cenna właściwość przyczyniająca się do zmniejszenia zużycia pamięci. Ponadto wspomniana na samym początku tego paragrafu wada tablic, tzn. zajmowanie przez nie stałego obszaru, może być w łatwy sposób ominięta poprzez sprytne ukrycie dynamicznego zarządzania tablicą w definicji klasy (można np. wykorzystać tzw. wektory, czyli obiekty podobne do tablic, ale dynamicznie powiększające się w miarę potrzeb). Wektory są bardzo czytelnie objaśnione np. w [Eck02].

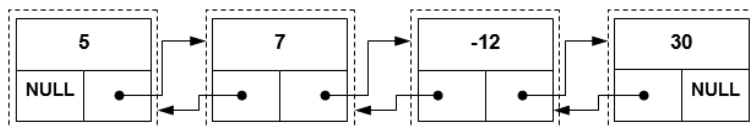
Listy innych typów

Listy jednokierunkowe są bardzo wygodne w stosowaniu i zajmują stosunkowo mało pamięci. Mimo to operacje na nich niekiedy zajmują dużo czasu. Zauważyło ten fakt sporo ludzi i tym sposobem zostały wymyślane inne typy list, np.:

lista dwukierunkowa — komórka robocza zawiera wskaźniki do elementów: poprzedniego i następnego (rysunek 5.13):

Rysunek 5.13.

Lista dwukierunkowa



- ◆ Pierwsza komórka znajdująca się w liście nie posiada swojego poprzednika; zaznaczamy to, wpisując wartość NULL do pola poprzedni.
- ◆ Ostatnia komórka znajdująca się w liście nie posiada swojego następnika; zaznaczamy to, wpisując wartość NULL do pola następny. Lista dwukierunkowa jest dość kosztowna, jeśli chodzi o zajętość pamięci, ale czasem szybkość działania jest ważniejsza niż małe zużycie pamięci — a właśnie szybkość działania jest zaletą listy dwukierunkowej.

Struktura wewnętrzna listy dwukierunkowej jest oczywista:

```
typedef struct rob
{
    int wartosc;
    struct rob *nastepny;
    struct rob *poprzedni;
}ELEMENT;
```

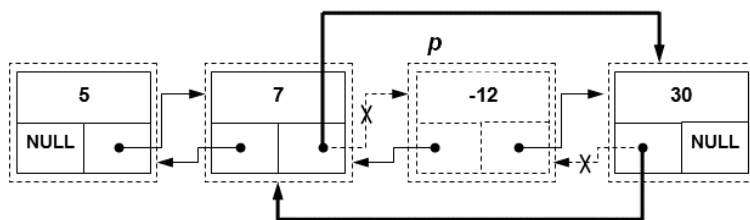
Załóżmy teraz, że podczas przeglądania elementów listy zapamiętaliśmy wskaźnik pozycji bieżącej p. (Przykładowo: szukaliśmy elementu spełniającego pewien warunek i na wskaźniku p nasze poszukiwania zakończyły się sukcesem). Jak usunąć element p z listy? Jak pamiętamy

z paragrafów poprzednich, do prawidłowego wykonania tej operacji niezbędna była znajomość wskaźników przed i po, wskazujących odpowiednio na komórki poprzednią i następną. W przypadku listy dwukierunkowej w komórce wskazywanej przez *p* te dwie informacje już się znajdują i wystarczy tylko po nie sięgnąć:

```
void usun2kier(ELEMENT *p)
{
    if(p->poprzedni!=NULL) //nie jest to element pierwszy
        p->poprzedni->nastepny=p->nastepny;
    if(p->nastepny!=NULL) //nie jest to element ostatni
        p->nastepny->poprzedni=p->poprzedni;
}
```

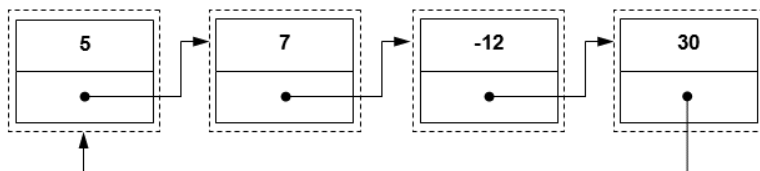
W zależności od konkretnych potrzeb można element *p* fizycznie usunąć z pamięci przez instrukcję `delete` lub też go w niej pozostawić do ewentualnych innych celów. Rysunek 5.14 jest odbiciem procedury `usun2kier` (potrzebne modyfikacje wskaźników są zaznaczone linią pogrubioną).

Rysunek 5.14.
Usuwanie danych
z listy dwukierunkowej



lista cykliczna — patrz rysunek 5.15 — jest zamknięta w pierścieniu: wskaźnik ostatniego elementu wskazuje „pierwszy” element.

Rysunek 5.15.
Lista cykliczna



Pewien element określany jest jako „pierwszy” raczej umownie i służy wyłącznie do wejścia w „magiczny” krąg wskaźników listy cyklicznej.

Każda z przedstawionych powyżej list ma swoje wady i zalety. Celem tej prezentacji było ukazanie istniejących rozwiązań, zadaniem zaś Czytelnika będzie wybranie jednego z nich podczas realizacji swojego programu.

Stos

Stos jest ważnym pojęciem w informatyce i — w szczególności — w oprogramowaniu systemowym¹². Choć takie zdanie brzmi bardzo groźnie, to chciałbym zapewnić, że nie kryje się za nim nic strasznego. Krótko mówiąc, jest to struktura danych, która ułatwia rozwiązanie wielu problemów natury algorytmicznej i w tę właśnie stronę wspólnie będziemy zdążać. Zanim dojdziemy do zastosowań stosu, spróbujmy go jednak zaimplementować w języku C++.

¹² Tym ostatnim zagadnieniem w tej książce nie będziemy się zajmowali.

Zasada działania stosu

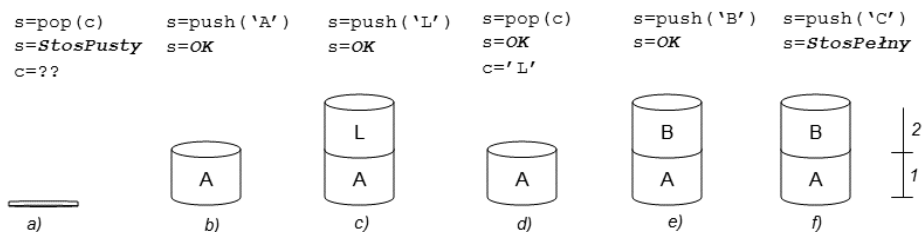
Stos jest strukturą danych, do której dostęp jest możliwy tylko od strony tzw. wierzchołka, czyli pierwszego wolnego miejsca znajdującego się na nim. Z tego też względu jego zasada działania jest bardzo często określana za pomocą angielskiego skrótu LIFO: *Last In First Out*, co w wolnym tłumaczeniu oznacza „ostatni będą pierwszymi”. Do odkładania danych na wierzchołek stosu służy zwyczajowo funkcja o nazwie `push(X)`, gdzie `X` jest daną pewnego typu. Może to być dowolna zmienna prosta lub złożona: liczba, znak, rekord itd.

Podobnie, aby pobrać element ze stosu, używa się funkcji o nazwie `pop(X)`, która załadowuje zmienną `X` daną zdjętą z wierzchołka stosu. Obie te podstawowe funkcje oprócz swojego głównego zadania, które zostało wzmiankowane wyżej, zwracają jeszcze kod błędu¹³. Jest to stała typu całkowitego, która informuje programistę, czy czasem nie nastąpiła sytuacja niepożądana, np. próba zdjęcia czegoś ze stosu w momencie, gdy był on już pusty, lub też próba odłożenia na nim kolejnej danej w sytuacji, gdy brakowało w nim miejsca (brak pamięci). Programowe realizacje stosu różnią się między sobą drobnymi szczegółami (ostateczne słowo w końcu ma programista!), ale ogólna koncepcja jest zbliżona do opisanej wyżej.

Zasada działania stosu może zostać zatem podsumowana następującymi, prostymi regułami:

- ◆ Po wykonaniu operacji `push(X)` element `X` sam staje się nowym wierzchołkiem stosu, przykrywając poprzedni wierzchołek (jeśli oczywiście coś na stosie już było).
- ◆ Jedynym bezpośrednio dostępnym elementem stosu jest jego wierzchołek.
- ◆ Próba wstawienia czegoś na pełny stos powinna zakończyć się błędem.
- ◆ Próba pobrania elementu z pustego stosu powinna zakończyć się błędem.

Dla dokładniejszego zobrazowania zasady działania stosu i powyższych reguł proszę prześledzić kilka operacji dokonanych na nim i efekt ich działania — patrz rysunek 5.16.



Rysunek 5.16. Stos i podstawowe operacje na nim

Rysunek przedstawia stos służący do zapamiętywania znaków. Stałe symboliczne `StosPusty`, `OK` i `StosPełny` są zdefiniowane przez programistę w module zawierającym deklarację stosu jako struktury danych. Wyrażają się one w wartościach typu całkowitego (co akurat nie ma specjalnego znaczenia). Nasz stos ma pojemność dwóch elementów, co jest oczywiście absurdalne, ale zostało przyjęte na użytek naszego przykładu, aby zilustrować efekt przepełnienia.

Symboliczny stos znajdujący się pod każdą z sześciu grup instrukcji ukazuje zawsze stan po wykonaniu „swojej” grupy instrukcji. Jak można łatwo zauważyć, operacje na stosie przebiegały pomyślnie do momentu osiągnięcia jego całkowitej pojemności; wówczas stos zasygnalizował sytuację błędną.

Jakie są typowe realizacje stosu? Najpopularniejszym sposobem jest użycie tablicy i zarezerwowanie jednej zmiennej w celu zapamiętania liczby danych aktualnie znajdujących się na stosie. Jest to dokładnie taki sam pomysł, jak ten zaprezentowany na rysunku 5.10, z jednym

¹³ Nie jest to bynajmniej obowiązkowe!

zastrzeżeniem: mimo iż wiemy, jak stos jest zbudowany od środka, nie zezwalamy nikomu na bezpośredni dostęp do niego. Wszelkie operacje odkładania i zdejmowania danych ze stosu muszą się odbywać za pośrednictwem metod `push` i `pop`. Jeśli zdecydujemy się na zamknięcie danych i funkcji służących do ich obsługi w postaci klasy¹⁴, to wówczas automatycznie uzyskamy bezpieczeństwo użytkowania — zapewni je sama koncepcja programowania zorientowanego obiektowo. Taki właśnie sposób postępowania obierzemy.

Możliwych sposobów realizacji stosu jest mnóstwo; wynika to z faktu, iż ta struktura danych nadaje się doskonale do ilustracji wielu zagadnień algorytmicznych. Dla naszych potrzeb ograniczymy się do bardzo prostej realizacji tablicowej, która powinna być uważana raczej za punkt wyjścia niż za gotową implementację.

W związku z założonym powyżej celowym uproszczeniem definicja klasy `STOS` jest bardzo krótka:



stos.h

```
const int DLUGOSC_MAX=2;
enum stan {OK=0, STOS_PELNY=1, STOS_PUSTY=2};

template <class TypPodst> class STOS
{
public:
    STOS() { szczyt=0;} // szczyt = pierwsza WOLNA komórka // konstruktor
    void clear() { szczyt=0;} // zerowanie stosu
    int push(TypPodst x);
    int pop (TypPodst &w);
    int StanStosu();
private:
    TypPodst t[DLUGOSC_MAX]; // stos = t[0]...t[DLUGOSC_MAX-1]
    int szczyt;
}; // koniec definicji klasy STOS
```

Nasz stos będzie mógł potencjalnie służyć do przechowywania danych wszelakiego rodzaju, z tego też powodu celowe wydało się zadeklarowanie go w postaci tzw. *klasy szablonowej*, co zostało zaznaczone przez słowo kluczowe `template`.

Idea klasy szablonowej polega na stworzeniu wzorcowego kodu, w którym typ pewnych danych (zmiennych, wartości zwracanych przez funkcje itp.) nie zostaje precyzyjnie określony, ale jest zastąpiony pewną stałą symboliczną. W naszym przypadku jest to stała `TypPodst`.

Zaletą tego typu postępowania jest dość duża uniwersalność tworzonej klasy, gdyż dopiero we właściwym kodzie (np. funkcji `main`) określamy, że np. `TypPodst` powinien zostać zamieniony na np. `float`, `char*` lub jakiś złożony typ strukturalny. Wadą klasy szablonowej jest jednak dość dziwna składnia, której musimy się trzymać, chcąc zdefiniować jej metody. O ile jeszcze definicje znajdują się w ciele klasy (tzn. pomiędzy jej nawiasami klamrowymi), to składnia przypomina normalny kod C++. W momencie jednak gdy chcemy definicje metody umieścić poza klasą (tzn. poza nawiasami klamrowymi zamykającymi jej definicję), to otrzymujemy tego rodzaju dziwolaży¹⁵:

¹⁴ Czyli dokonamy tzw. *hermetyzacji*.

¹⁵ Oczywiście zawsze można się pocieszać, że ewentualnie mogłoby to zostać jeszcze bardziej skomplikowane. Ale żarty na bok, powyższe problemy wynikają z prostego faktu: C++ należy do grupy języków, których kompilatory muszą znać precyzyjnie typ danych, jakie wchodzi w grę podczas programowania, stąd też każdy zabieg służący uczynieniu go pozornie nieczułym na typy danych musi być nieco sztuczny. Warto wspomnieć przy okazji, że istnieją języki z zasady pozbawione pojęcia typu danych, np. *Smalltalk-80* (jest to język obiektowy o zupełnie innej filozofii niż C++, który wydaje się przy nim swego rodzaju *assemblerem obiektowym*).

```
template <class TypPodst> int STOS<TypPodst>::push(TypPodst x)
{
    if (szczyt<DLUGOSC_MAX)
    {
        t[szczyt++]=x;
        return (OK);
    }
    else
        return (STOS_PELNY);
}
```

Metoda `push`, bowiem to jej kod mamy przed oczami, jest bardzo prosta, co jest zresztą cechą wszelkich realizacji tablicowych. Nowy element `x` (jakikolwiek by był jego typ) jest zapisywany na szczycie stosu, który jest wskazywany w prywatnej dla klasy zmiennej `szczyt`. Następnie wartość szczytu stosu jest inkrementowana — to wszystko pod warunkiem, że stos nie jest już zapelniony!

Metoda `pop` wykonuje odwrotne zadanie — zdejmowany ze stosu element jest zapamiętywany w zmiennej `w` (przekazanej w wywołaniu przez referencję); zmienna `szczyt` jest oczywiście dekrementowana — pod warunkiem, że stos nie był pusty (z próżnego to nawet i programista nie... należy?):

```
template <class TypPodst> int STOS<TypPodst>::pop(TypPodst &w)
{ // „w” zostanie „załadowane” wartością zdejętą ze stosu
    if (szczyt>0)
    {
        w=t[--szczyt];
        return (OK);
    }else
        return (STOS_PUSTY);
}
```

Od czasu do czasu może zająć potrzeba zbadania stanu stosu bez wykonywania na nim żadnych operacji. Użyteczna może być wówczas następująca funkcja:

```
template <class TypPodst> int STOS<TypPodst>::StanStosu()
{
    // zwraca informacje o stanie stosu
    switch(szczyt)
    {
        case 0 :return (STOS_PUSTY);
        case DLUGOSC_MAX :return (STOS_PELNY);
        default :return (OK);
    }
}
```

Jakie są inne możliwe sposoby zdefiniowania stosu? Nie powinno dla nikogo stanowić niespodzianki, że logicznym następstwem użycia tablic są struktury dynamiczne, np. listy. Bezpośrednie wbudowanie listy do stosu, zamiast na przykład tablicy `t`, tak jak wyżej, byłoby jednakże nieefektywne — warto poświęcić odrobinę wolnego czasu i stworzyć osobną klasę od samego początku.

Chwilę uwagi należy jeszcze poświęcić wykorzystaniu stosu. Zasadniczą kwestią jest składnia użycia klasy szablonowej w funkcji `main`. Deklaracja stosu `s`, który ma posłużyć do przechowywania zmiennych typu np. `char*`, dokonuje się poprzez np.:

```
STOS<char*> s;
```

Podobnie dzieje się w przypadku każdego innego typu danych, wystarczy go tylko umieścić w nawiasach `< >`.

**stos.cpp**

```
int main()
{
    STOS<char*> s1; // stos do przechowywania tekstów
    STOS<double> s2; // stos do przechowywania liczb zmiennopozycyjnych
    cout << "Odkładam na stos s1:\n";
    for(int i=0; i<3;i++)
    {
        if (s1.StanStosu() != STOS_PELNY)
        {
            cout << "Próbuję włożyć:" << tabl[i] << ", ";
            s1.push(tabl[i]);
        } else
        {
            cout << "Stos pełny, stop!\n";
            break;
        }
    }
    for(int i=0; i<3;i++)
    {
        char *z;
        if (s1.pop(z)==OK)
            cout << "\nZdejmuję ze stosu s1: " << z << endl;
        else
        {
            cout << "Stos pusty, koniec!\n";
            break;
        }
    }
}
```

Oto wyniki naszego programu (wersja na *ftp* jest nieco bardziej rozbudowana):

```
Odkładam na stos s1:
Próbuję włożyć:Ala, Próbuję włożyć:ma, Stos pełny, stop!
Zdejmuję ze stosu s1: ma
Zdejmuję ze stosu s1: Ala
Stos pusty, koniec!
```

Kolejki FIFO

Kolejki typu FIFO(ang. *First In First Out*, co w wolnym tłumaczeniu oznacza: *kto pierwszy, ten lepszy*) będą kolejnym omawianym typem danych. Struktura ta działa na zasadzie obsługi ogonka ludzi przed kasą sklepową, oczywiście zakładając brak wyjątków typu klient uprzywilejowany.

Podobnie jak i stos, jest to struktura danych o dostępie ograniczonym. Zakłada ona dwie podstawowe operacje:

- ♦ wstaw — wprowadź dane (klienta) na ogon kolejki;
- ♦ obsłuż — usuń dane (klienta) z czoła kolejki.

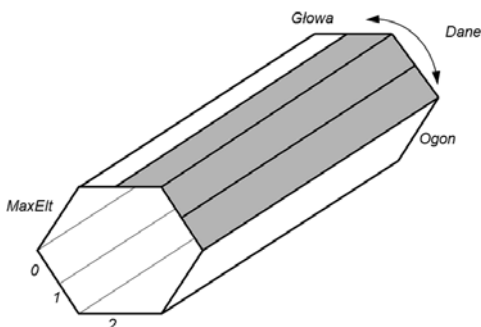
W porównaniu ze stosem kolejki są rzadziej stosowane w praktyce programowania. Pewne zagadnienia natury algorytmicznej dają się jednak relatywnie łatwo rozwiązywać właśnie przy użyciu tej struktury danych i to jest głównie powód niniejszej prezentacji.

Jak to zwykle bywa, możliwych implementacji kolejek jest co najmniej kilka. Realizacja efektywna czasowo za pomocą list jednokierunkowych jest zbliżona do tej wzmiankowanej przy okazji omawiania stosu. Nie będzie ona stanowiła przedmiotu naszej dyskusji, ograniczymy się jedynie do prostej implementacji tablicowej (zbliżonej zresztą ideowo do tablicowej realizacji stosu).

Tablicowa implementacja kolejki FIFO jest wyjaśniona na rysunku 5.17.

Rysunek 5.17.

Tablicowa realizacja
kolejki FIFO
— koncepcja



Zawartość kolejki stanowią elementy pomiędzy *głową* i *ogonem* — te dwie zmienne będą oczywiście zmiennymi prywatnymi klasy FIFO. Dojście nowego elementu do kolejki wiąże się z inkrementacją zmiennej *ogon* i dopisaniem elementu u dołu „szarej strefy”. Oczywiście w pewnym momencie może się okazać, że *ogon* osiągnął koniec tablicy — wówczas pojęcie dołu odwróci się i to dosłownie!

W takim przypadku cała szara strefa zawinie się wokół elementu zerowego tablicy. Obsługa „klienta” będącego aktualnie na początku kolejki wiąże się z zapamiętaniem elementu czołowego i z inkrementacją zmiennej *głowa*. Trzeba się ponadto umówić, jak interpretować stwierdzenie, że kolejka jest pusta?

Zamiast komplikować sobie życie specjalnymi testami zawartości tablicy, można po prostu założyć, że gdy *głowa*=*ogon*, to kolejka jest pusta. Tym samym trzeba zarezerwować jeden dodatkowy element tablicy, który nigdy nie będzie wykorzystany z uwagi na sposób działania metody *wstaw*. Po tych rozbudowanych wyjaśnieniach programowa realizacja kolejki nie powinna już stanowić żadnej niespodzianki dla Czytelnika:



kolejka.h

```
enum stan {OK, BLAD};
template <class TypPodst> class FIFO
{
private:
    TypPodst *t;
    int glowa,ogon,MaxElt;
public:
    FIFO(int n)
    {
        MaxElt=n;
        glowa=ogon=0;
        t=new TypPodst[MaxElt+1];
    }

    void wstaw(TypPodst x)
    {
        t[ogon++]=x;
        if(ogon>MaxElt) ogon=0;
    }

    int obsluz(TypPodst &w)
    {
        if (glowa==ogon)
            return BLAD; // informacja o błędzie operacji
        w=t[glowa++];
    }
};
```

```

        if(glowa>MaxElt) glowa=0;
        return OK;
    }

    bool pusta()
    { // czy kolejka jest pusta?
    if (glowa==ogon)
        return true; // kolejka pusta
    else
        return false;
    }
};

```

Podobnie jak w przypadku stosu zdefiniowaliśmy nowy typ danych w postaci klasy szablonowej. Umożliwia to łatwe definiowanie rozmaitych kolejek obsługujących różnorodne typy danych. Definicja klasy FIFO nie jest kompletna: brakuje w niej na przykład jawnego destruktora, ponadto kontrola operacji mogłaby być nieco bardziej rozbudowana... Te dodatki są jednak pozostawione Czytelnikowi jako proste ćwiczenie programistyczne.

Popatrzmy, jak wygląda w praktyce korzystanie z nowej struktury danych:



kolejka.cpp

```

#include <iostream>
using namespace std;
#include "kolejka.h"
static char *tab[]={"Kowalska","Fronczak","Becki","Pigwa"};
int main()
{
    int i;
    FIFO<char*> kolejka(5); // kolejka 5-osobowa
    for(i=0; i<4; i++)
        kolejka.wstaw(tab[i]);
    char *s;
    for(i=0; i<5;i++)
    {
        int res=kolejka.obsluz(s);
        if (res==OK)
            cout << "Obsłużony został klient: "<<s<<endl;
        else
            cout << "Kolejka pusta!\n";
    }
}

```

Zasada obsługi kolejki (w krajach cywilizowanych) polega na uwzględnianiu w pierwszej kolejności osób, które zjawily się na samym początku. Tak też jest w naszym przykładzie, o czym najbardziej świadczą rezultaty wykonania programu:

```

Obsłużony został klient: Kowalska
Obsłużony został klient: Fronczak
Obsłużony został klient: Becki
Obsłużony został klient: Pigwa
Kolejka pusta!

```

Stery i kolejki priorytetowe

W paragrafach poprzednich mieliśmy okazję zapoznać się m.in. z dwiema strukturami danych stanowiącymi swoje ideowe skrajności :

- ♦ *Kolejką* — usuwało się z niej w pierwszej kolejności „najstarszy” element.
- ♦ *Stosem* — usuwało się z niego w pierwszej kolejności „najmłodszy” element.

Były to struktury danych służące z zasady do zapamiętywania danych nieuporządkowanych, co zdecydowanie upraszczało wszelkie operacje! Kolejna zaś struktura danych, którą będziemy się zajmować — kolejki priorytetowe — działa wg zupełnie odmiennej filozofii, choć zachowuje ciągle zaletę operowania nieuporządkowanym zbiorem danych. (Stwierdzenie o nieuporządkowaniu jest prawdą w sensie globalnym — lokalnie fragmenty sterty są w pewien szczególny sposób uporządkowane, o czym przekonamy się już za moment). Dwie podstawowe operacje wykonywane na kolejkach priorytetowych polegają na:

- ◆ wstawianiu nowego elementu;
- ◆ usuwaniu największego elementu¹⁶.

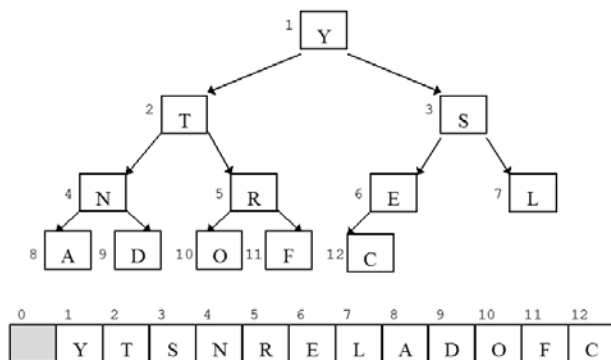
Jednym z najłatwiejszych sposobów realizacji kolejek priorytetowych jest użycie struktury danych zwanej *stertą* (ang. *heap* — inna spotykana polska nazwa to *stóg* lub *kopiec*). O tej strukturze danych napomknęliśmy już przy okazji sortowania (patrz rozdział 4.), obecnie ten temat zostanie rozwinięty w nieco innym ujęciu.

Sterta jest swego rodzaju drzewem binarnym, które ze względu na szczególne własności warto omówić osobno. (Kwestia terminologiczna: zarówno sterta, jak i kolejki priorytetowe są strukturami danych, jednakże tylko kolejka priorytetowa ma charakter czysto abstrakcyjny).

Uporządkowanie elementów wchodzących w skład sterty można zaobserwować na rysunku 5.18 przedstawiającym 12-elementową stertę. Jest to również przykład tzw. *kompletnego drzewa binarnego*. Stosując pewne uproszczenie definicyjne, można także powiedzieć, iż jest to „drzewo bez dziur”. Jeśli spojrzeć na numery przypisane węzłom drzewa, to widać, że ich kolejność definiuje pewien charakterystyczny porządek wypełniania go: pod istniejące węzły przywieszamy maksymalnie po dwa nowe aż do ułożenia wszystkich 12 elementów. Można to oczywiście wyrazić nieco bardziej formalnie, ale zapewniam, że zdecydowanie mniej zrozumiale.

Rysunek 5.18.

Tablicowa realizacja
kolejki FIFO
— przykład



Liniowy porządek wypełniania drzewa automatycznie sugeruje sposób jego składowania w tablicy¹⁷:

- ◆ Wierzchołek (czyli de facto korzeń, bo drzewo jest odwrócone) = 1.
- ◆ Lewy potomek i -tego węzła jest schowany pod indeksem $2*i$.
- ◆ Prawy potomek i -tego węzła jest schowany pod indeksem $2*i+1$.



Uwaga

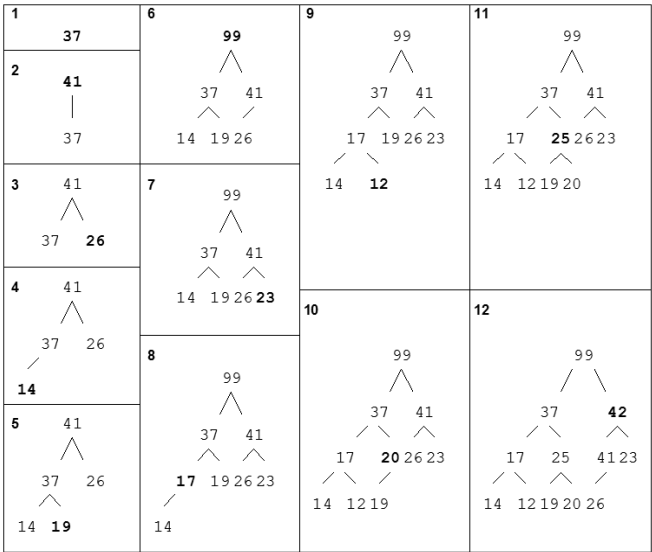
Dany węzeł może mieć od 0 do 2 potomków.

¹⁶ Jeśli w kolejce priorytetowej będą składowane rekordy o pewnej strukturze, to jednym z pól rekordu będzie jego priorytet wyrażony w postaci liczby całkowitej dodatniej lub ujemnej. W naszych przykładach dla prostoty ograniczymy się tylko do przypadku składowania liczb całkowitych.

¹⁷ Zerowa komórka tablicy nie jest używana do składowania danych.

Powyżej zdefiniowaliśmy sposób składowania danych, nic jednak nie powiedzieliśmy o zależnościach istniejących pomiędzy nimi. Otóż cechą charakterystyczną sterty jest to, iż *wartość każdego węzła jest większa¹⁸ od wartości węzłów jego dwóch potomków* — jeśli oczywiście istnieje. Sposób organizacji drzewa (jak również w konsekwencji tablicy) ułatwia operacje wstawiania i usuwania elementów. Możemy bowiem nowy element bez problemu dopisać na końcu tablicy (co oczywiście zburzy nam ład wcześniej tam panujący), następnie za pomocą dość prostych modyfikacji tablicy przywrócić z powrotem tablicy (drzewu) własności sterty. Popatrzmy na przykładzie, w jaki sposób jest konstruowana sarta z elementów: 37, 41, 26, 14, 19, 99, 23, 17, 12, 20, 25 i 42 — dołączanych sukcesywnie do drzewa. Cały proces jest pokazany na rysunku 5.19.

Rysunek 5.19.
Konstrukcja sterty
na przykładzie



Na rysunku widzimy, gdzie wędruje nowy — zaznaczony wytłuszczoną czcionką — element. Po-
przez porównanie z etapem poprzednim łatwo zauważamy modyfikacje struktury drzewa. Załóżmy,
że dokładamy na koniec drzewa liczbę 99 (patrz etap 5.). Drzewo ma już 5 elementów, zatem
nowy powędruje na miejsce nr 6 w tablicy — pod 26. W tym momencie jednak zostaje złamana
zasada konstrukcji sterty: potomek węzła ma większą wartość niż sam węzeł, do którego jest on
przywieszony! Co możemy zrobić, aby przywrócić porządek? W tym miejscu wystarczy zwyczaj-
nie zamienić 26 i 99 miejscami, aby wszystko się *lokalnie* uspokoiło. Zauważmy, że taka lokalna
zamiana przywraca porządek jedynie na aktualnie analizowanym poziomie — burząc go być
może na następnym! Zatem, aby w całej stercie zapanował porządek, należy proces zamieniania
kontynuować¹⁹ w górę aż do osiągnięcia korzenia. (W naszym przykładzie konieczna będzie
jeszcze zamiana liczb 99 i 41). Programową realizację opisaną powyżej czynności wykona pro-
cedura o nazwie DoGory. Opisaną sytuację ilustruje rysunek 5.20.

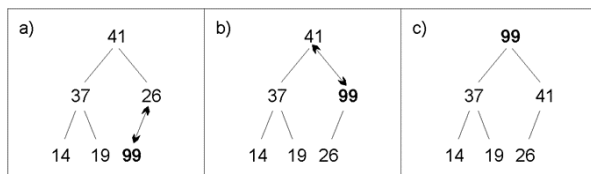
Teraz, gdy już wiemy, CZYM jest sarta i JAK się ją tworzy, pora wyjaśnić wreszcie, dlaczego
sterta umożliwia łatwe tworzenie kolejek priorytetowych. Wiemy już, że istotną cechą wy-
różniającą kolejki priorytetowe od innych podobnych struktur danych stanowi to, że pierwszym
obsługiwanym „klientem” jest ten, który ma największą wartość (lub też, w przypadku rekordów,
największą wartość pewnego wybranego pola).

¹⁸ Spotyka się również implementacje, w których jest to wartość nie większa.

¹⁹ Jeśli oczywiście zachodzi potrzeba.

Rysunek 5.20.

Poprawne wstawianie
nowego elementu
do sterty



Jeśli trzymać się ciągle analogii kolejki do kasy sklepowej, to można by powiedzieć, że wszyscy ustawiają się elegancko na końcu „ogonka”, ale to kasjerka patrzy klientom w oczy i wybiera do obsługi tych najbardziej uprzywilejowanych (ewentualnie najprzystojniejszych...).

W przypadku list i zwykłych tablic problemem byłoby znalezienie właśnie tego największego elementu — należałoby w tym celu dokonać przeszukania, które zajmuje czas proporcjonalny do N (wielkości tablicy lub listy). A jak to wygląda w naszym przypadku? Spójrzmy raz jeszcze na tablicę z rysunku 5.18 dla upewnienia się: TAK, my w ogóle nie musimy szukać największego elementu, bowiem z założenia znajduje się on w komórce tablicy o indeksie 1!

Po euforii powinna jednak przyjść chwila zastanowienia: a co z wstawianiem? Elementy są co prawda zawsze dokładane na koniec, ale potem zawsze trzeba wywołać procedurę `DoGory`, która przywróci stercie zachwiany (ewentualnie) porządek. Czy czasem owa procedura nie jest na tyle kosztowna, że ewentualny zysk z użycia sterty nie jest już tak oczywisty? Na szczęście okazuje się, że nie. Wszelkie algorytmy operujące na sterce wykonują się wprost proporcjonalnie do długości drogi odwiedzanej podczas przechodzenia przez drzewo binarne reprezentujące stertę. Co można powiedzieć o tej długości, wiedząc, że drzewo binarne jest kompletne? Na przykład to, iż dowolny wierzchołek jest odległy od wierzchołka (korzenia) o co najwyżej $\log_2 N$ węzłów! Z tego właśnie powodu algorytmy stertowe wykonują się na ogół w czasie „logarytmicznym”. Jest to dobry wynik, decydujący często o użyciu tej, a nie innej struktury danych.

Po tak długim wstępie warto wreszcie zaprezentować kilka linii kodu w C++, które przemówią lepiej niż rozwlekłe wyjaśnienia. Definicja klasy `Sterna`²⁰ jest następująca:

***sterta.h***

```

#include <iostream>
using namespace std;

class Sterta
{
public:
    Sterta(int nMax)
    {
        t=new int[nMax+1];
        L=0;
    }
    void wstaw(int x);
    int  obsluz();
    void DoGory();
    void NaDol();
    void pisz();
private:
    int *t;
    int L; //liczba elementów
}; // koniec definicji klasy Sterta
  
```

²⁰ Dla uproszczenia podany zostanie przykład dla sterty liczb całkowitych.

Konstruktor klasy tworzy tablicę, w której będą zapamiętywane elementy — `t[0]` jest nieużywane, stąd deklaracja tablicy o rozmiarze `nMax+1`, a nie `nMax` (jest to szczególnie implementacyjny ukryty przed użytkownikiem).

Na początek zajmijmy się wstawieniem nowego elementu do sterty:

```
void Sterta::wstaw(int x)
{
    t[++L]=x;
    DoGory();
}
```

Procedura `DoGory` była już wcześniej wzmiankowana: zajmuje się ona przywróceniem porządku w stercie po dołączeniu na koniec tablicy `t` nowego elementu.

Treść procedury `DoGory` nie powinna stanowić niespodzianki. Jedyną różnicą pomiędzy wskazaną na rysunku 5.20 zamianą elementów jest... jej brak! W praktyce szybsze okazuje się przesunięcie elementów w drzewie, tak aby zrobić miejsce na unoszony do góry ostatni element tablicy:

```
void Sterta::DoGory()
{
    int temp=t[L];
    int n=L;
    while( (n!=1) && (t[n/2]<=temp) )
    {
        t[n]=t[n/2];
        n=n/2;
    }
    t[n]=temp;
}
```

Jest to być może zbędna sztuczka, biorąc pod uwagę oryginalny algorytm polegający na systematycznym zamienianiu elementów ze sobą (w miarę potrzeby) podczas przechodzenia przez węzły drzewa, jednak pozwala ona nieco przyspieszyć procedurę²¹.

Nawiązując do kolejek priorytetowych, wspomnieliśmy, że są one łatwo implementowalne za pomocą sterty. Wstawianie „klienta” do kolejki priorytetowej (czyli sterty) na sam jej koniec zostało zrealizowane powyżej. Jak pamiętamy, pierwszym obsługiwanym „klientem” w kolejce priorytetowej był ten, który miał największą wartość — `t[1]`. Ponieważ po usunięciu tego elementu w tablicy robi się dziura, ostatni element tablicy wstawiamy na miejsce korzenia, dekrementujemy `L` i wywołujemy procedurę `NaDol`, która skoryguje w odpowiedni sposób stertę, której porządek mógł zostać zaburzony:

```
int Sterta::obsluz()
{
    int x=t[1];
    t[1]=t[L--]; // brak kontroli błędów!!
    NaDol();
    return x;
}
```

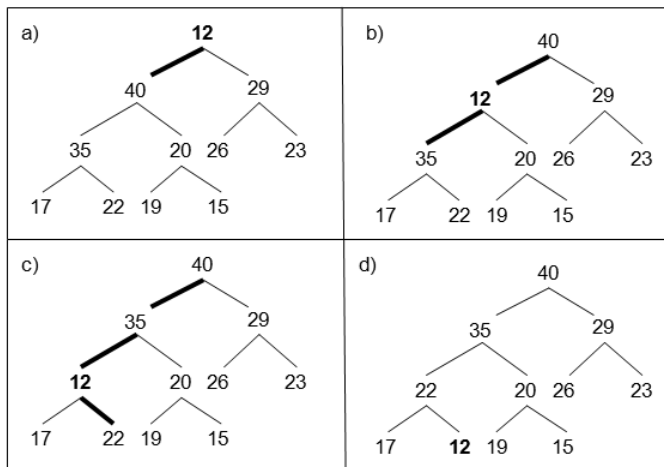
(Czytelnik powinien samodzielnie rozbudować powyższą metodę, wzbogacając ją o elementarną kontrolę błędów).

Jak powinna działać procedura `NaDol`? Zmiana wartości w korzeniu mogła zaburzyć spokój zarówno w lewym, jak i prawym jego potomku. Nowy korzeń należy za pomocą zamiany z większym z jego potomków przesiać w dół drzewa aż do momentu znalezienia właściwego dlań miejsca. Popatrzmy na efekt zadziałania procedury `NaDol` wykonanej na pewnej stercie (patrz rysunek 5.21).

²¹ Która oczywiście pozostanie w dalszym ciągu logarytmiczna — cudów bowiem w informatyce nie ma!

Rysunek 5.21.

Ilustracja procedury
NaDo1



Element 12 został zaznaczony wytłuszczoną czcionką. Za pomocą pogrubionej kreski zaprezentowano drogę, po której zstępował element 12 w stronę swojego... miejsca ostatecznego spoczynku!

Oto jak można sposób zrealizować procedurę NaDo1:

```
void Sterta::NaDo1()
{
    int i=1;
    while(1)
    {
        int p=2*i;    // lewy potomek węzła 'i' to (p), prawy to (p+1)
        if(p>L)
            break;
        if(p+1<=L)    //prawy potomek niekoniecznie musi istnieć!
            if(t[p]<t[p+1]) p++; //przesuwamy się do następnego
        if(t[i]>=t[p]) break;
        int temp=t[p]; //zamiana
        t[p]=t[i];
        t[i]=temp;
        i=p;
    }
}
```

Sposób korzystania ze sterty jest zbliżony do poprzednio opisanych struktur danych i nie powinien sprawić Czytelnikowi żadnych problemów. Nieco bardziej interesujące jest ukazanie efektywnego zastosowania sterty do... sortowania danych.

Wystarczy bowiem dowolną tablicę do posortowania wpierw zapamiętać w sterce, używając metody wstaw, a następnie zapisać ją od tyłu w miarę obsługiwaną za pomocą metody obsluz:

```
#include "sterta.h"
int main()
{
    int i, tab[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
    Sterta s(14);
    for (i=0;i<14;i++)
        s.wstaw(tab[i]);
    for (i=14;i>0;i--)
        tab[i-1]=s.obsluz();
    cout<<"Tablica posortowana:\n";
    for (i=0;i<14;i++)
        cout << " " << tab[i];
}
```

Jest to oczywiście jedno z możliwych zastosowań sterty — prosta i efektywna metoda sortowania danych, średnio zaledwie dwa razy wolniejsza od sortowania szybkiego poznanego w poprzednim rozdziale (algorytmu Quicksort).

Powyższa procedura może być jeszcze bardziej przyspieszona poprzez włączenie kodu metod wstaw i obsłuz wprost do funkcji sortującej, tak aby uniknąć zbędnych i kosztownych wywołań proceduralnych. W tym przypadku zachodzi jednak potrzeba zaglądania do prywatnych informacji klasy — tablicy `t` (patrz plik `sterta.h`), zatem procedura sortująca musiałaby być funkcją zaprzyjaźnioną. Łamiemy jednak w tym momencie koncepcję programowania obiektowego (separacja prywatnego wnętrza klasy od jej zewnętrznego interfejsu)!

Jest to cena, którą płacimy za efektywność — funkcje zaprzyjaźnione zostały wprowadzone do C++ zapewne również z uwagi na użycie tego języka do programowania aplikacji wyjściowych, a nie tylko do prezentacji algorytmów (jak to jest w przypadku Pascala, który zawiera celowe mechanizmy zabezpieczające przed używaniem dziwnych sztuczek, bez których programy działałyby zbyt wolno na rzeczywistych komputerach).



Patrz także

Patrz także rozdział 4., gdzie zamieściłem bardziej zwięzłą wersję samego algorytmu sortowania przez kopcowanie.

Drzewa i ich reprezentacje

Dyskusją na temat tzw. *drzew* można by z łatwością wypełnić kilka rozdziałów. Temat jest bardzo rozległy i różnorodność aspektów związanych z drzewami znacznie utrudnia decyzję dotyczącą tego, co wybrać, a co pominąć. W ostatecznym rozrachunku zwyciężyły względy praktyczne: zostaną szczegółowo omówione te zagadnienia, które Czytelnik będzie mógł z dużym prawdopodobieństwem wykorzystać w codziennej praktyce programowania. Bardziej szczegółowe rozważania dotyczące drzew można znaleźć w zasadzie w większości książek poświęconych ogólnie strukturom danych. Ponieważ jednak te ostatnie nie są celem samym w sobie (o czym bardzo często autorzy książek o algorytmice zapominają), to wierzę, że bardziej praktyczne podejście do tematu zostanie przez większość Czytelników zaakceptowane.

Rozważania na temat drzew będą głównie ilustrowane przy pomocy najpopularniejszych i najczęściej używanych *drzew binarnych*, których użyteczność w rozwiązywaniu przeróżnych zagadnień algorytmicznych jest niezaprzeczalna.

Drzewo jako reprezentacja np. struktur organizacyjnych, systemu folderów w systemie operacyjnym, genealogii rodzinnych itp. jest doskonale znane każdemu z nas. Już tak szerokie rozpo-

wszechnienie drzew w świecie rzeczywistym sugeruje dużą użyteczność tych struktur.

Drzewo jest reprezentowane przez *zbiór węzłów*, połączonych ze sobą relacją, nazwijmy to, „rodzicielską”: umawiany się, że z jednego węzła może się wywodzić kolejny węzeł potomny (jeden lub więcej).

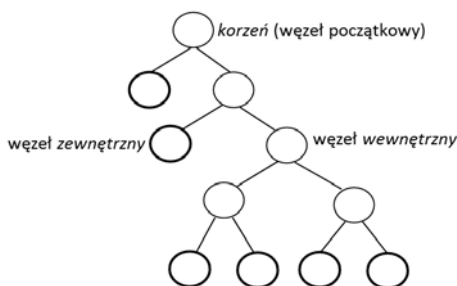
Wyróżniony węzeł drzewa, od którego umownie wywodzą się pozostałe węzły, nazywamy *korzeniem*. Węzeł jest przedstawiany jako kółeczko lub inny kształt, w zależności od naszych potrzeb. Prosty przykład znajduje się na rysunku 5.22.

Krawędzie łączące węzły drzewa mogą być wyposażone w strzałki, ale taka notacja jest w zasadzie nadmiarowa. Zazwyczaj strzałki na rysunkach drzew są pomijane, gdyż kierunek przechodzenia wynika z zasady „rodzicielskiej”, ale z drugiej strony same w sobie strzałki nie szkodzą i czasami się je pokazuje.

Węzły nieposiadające potomków nazywane są *liśćmi* lub, już bez nawiązywania do analogii biologicznych, *węzłami końcowymi* (na rysunku 5.22 przykładowy węzeł oznaczony jako węzeł zewnętrzny).

Rysunek 5.22.

Drzewo i podstawowe
pojęcia z nim związane



Wysokością węzła nazywamy długość najdłuższej ze ścieżek prowadzących od tego węzła do liści.

Głębokością węzła nazywamy długość ścieżki łączącej go z korzeniem.

Drzewo, którym od węzła musi odchodzić określona liczba potomków, zwane jest *m*-drzewem. Najbardziej znanym przykładem takiego typu drzewa są drzewa binarne, w których od danego węzła odchodzą dwa węzły potomne — lewy i prawy. Drzewo, w którym kolejność wymieniania węzłów-potomków ma dla nas znaczenie, zwane jest *uporządkowanym*.

W *m*-drzewach czasami wyróżnia się dwa rodzaje węzłów: wewnętrzne i zewnętrzne — te ostatnie nie posiadają potomków. Zwróćmy uwagę na drobny niuans terminologiczny: w *m*-drzewie „liść” jest tak naprawdę węzłem wewnętrznym (!), którego wszyscy potomkowie są zewnętrznymi.

W przypadku drzew binarnych warto wiedzieć o ich właściwościach matematycznych, które ułatwiają analizę złożoności algorytmów opartych na tych strukturach:

- ♦ Drzewo binarne zawierające n węzłów wewnętrznych ma $n+1$ węzłów zewnętrznych.
- ♦ Wysokość drzewa binarnego zawierającego n węzłów wewnętrznych wynosi co najmniej $\lg n$, a co najwyżej $n-1$.

W rozdziale 10. poznamy uogólnioną strukturę drzewiastą, zwaną grafem. Ponieważ tematowi grafów poświęcony jest wyodrębniony rozdział, nie będę tu wnikał w relacje pomiędzy grafami a drzewami, przyjmijmy po prostu, że w najbardziej ogólnym przypadku zawsze mamy do czynienia z drzewami. Jeśli w drzewie nie wyróżnimy korzenia i nie wprowadzimy ograniczeń na liczbę węzłów przylegających, to nagle zaczynamy mówić o strukturze grafowej!

Czym w praktyce są drzewa binarne? Są to struktury bardzo podobne do list jednokierunkowych, ale wzbogacone o jeszcze jeden wymiar (lub *kierunek*, jak kto woli).

Podstawowa komórka służąca do konstrukcji drzewa binarnego ma postać:

```
struct wezel
{
    int info; // lub dowolny inny typ danych
    struct wezel *lewy, *prawy;
}
```

Jak łatwo jest zauważyć, w miejsce jednego wskaźnika następny (jak w liście jednokierunkowej) mamy do czynienia z dwoma wskaźnikami o nazwach *lewy* i *prawy*, będącymi wskaźnikami do lewej i prawej gałęzi drzewa binarnego. Jeśli nasz algorytm wymaga umiejętności „przechadzania się” w obu kierunkach (od korzenia w dół, do kolejnych węzłów potomnych i na odwrót), struktura danych powinna zawierać także wskaźnik do „przodka”, od którego wywodzi się węzeł potomny, np.:

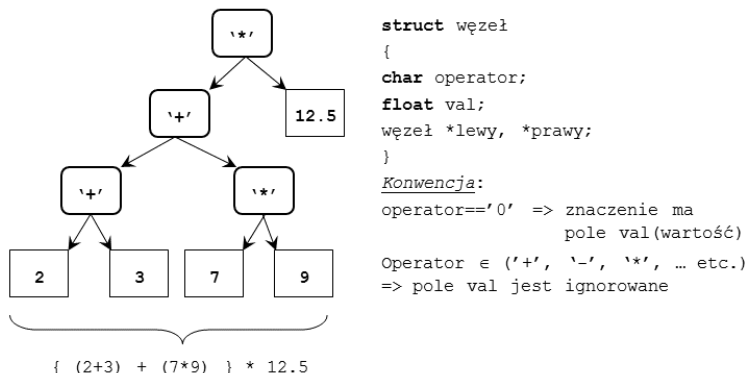
```
struct wezel
{
    int info; // lub dowolny inny typ danych
    struct wezel *lewy, *prawy; // węzły potomne
    struct wezel *przodek; // „ojciec”
}
```

Wskazanie na węzeł przodek jest zbędne, jeśli do przechadzania się po drzewach używamy rekurencji, która z natury rzeczy zapamiętuje, skąd nastąpiło przejście podczas przechadzania się po drzewie.

Aby dobrze zrozumieć sposób działania i użyteczność drzew binarnych, popatrzmy na rysunek 5.23.

Rysunek 5.23.

*Drzewa binarne
i wyrażenia arytmetyczne*



Rysunek pokazuje jeden z możliwych przykładów zastosowania drzew binarnych do reprezentowania wyrażeń arytmetycznych. Do tego przykładu jeszcze powrócimy w dalszych paragrafach, na razie wystarczy ogólny opis sposobu korzystania z takiej reprezentacji. Otóż dowolne wyrażenie arytmetyczne może być zapisane w kilku odmiennych postaciach związanych z położeniem operatorów: *przed swoimi argumentami*, *po nich* oraz klasycznie *pomiędzy nimi* (jeśli oczywiście mamy do czynienia tylko z wyrażeniami dwuargumentowymi, co pozwolimy sobie tutaj dla uproszczenia przykładów założyć).

Struktura danych z tego rysunku jest zwykłym drzewem binarnym, posiadającym dwa pola przeznaczone do przechowywania danych (operator i val) oraz tradycyjne wskaźniki do lewego i prawego odgałęzienia naszego odwróconego do góry nogami drzewa. Umówimy się ponadto, że w przypadku gdy pole operator zostanie zainicjowane jakąś bezsensowną wartością (tutaj 0; nie jest to żaden znany operator), to wówczas pole val ma jakąś wartość, którą możemy uznać za sensowną. Taka dualna reprezentacja może posłużyć do łatwego rozróżnienia przy użyciu tylko jednego typu rekordów dwóch typów węzłów:

- ♦ *wartości* („liście” drzewa),
- ♦ *operatora arytmetycznego*, wiążącego w ogólnym przypadku trzy typy węzłów:
 - ♦ Lewy i prawy potomek stanowią wyrażenia.
 - ♦ Lewy potomek jest wyrażeniem, a prawy wartością.
 - ♦ Prawy potomek jest wyrażeniem, a lewy wartością.

Jeśli napiszemy odpowiednie funkcje obsługujące powyższą strukturę danych wedle przyjętych przez nas reguł postępowania, to za pomocą takiej prostej reprezentacji możemy wyrazić dowolnie skomplikowane wyrażenia arytmetyczne, wykonywać na nich operacje, różniczkować je, etc. Wszystko zależy wyłącznie od tego, co zamierzamy uzyskać — możliwych zastosowań jest dość sporo, a ponadto, jak się okaże już wkrótce, jeśli do pracy zaprzęgniemy rekurencję, to algorytmy obsługi drzew binarnych (i nie tylko) staną się bardzo proste i zrozumiałe na pierwszy rzut oka.

Czy reprezentacja za pomocą rekurencyjnych struktur danych jest optymalna? Na to pytanie można odpowiedzieć sensownie jedynie, mając przed oczami ostateczne zastosowanie implementowanego drzewa: jeśli nie dbamy zbyt wiele o zajętość pamięci, a zależy nam na łatwości implementacji, to reprezentacja tablicowa może okazać się nawet lepsza od tej *klasycznej*, zaprezentowanej powyżej.

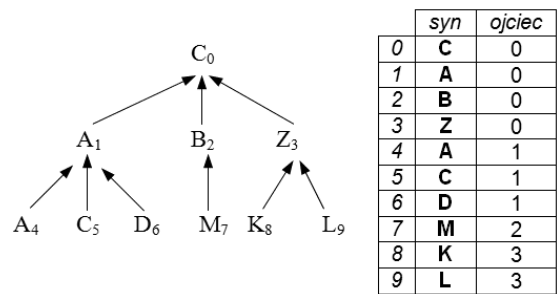
Jak zapamiętać drzewo w tablicy? Nie jest to bynajmniej dla nas problem nowy, poznaliśmy już wcześniej prostą metodę na zapamiętanie w tablicy innej drzewiastej struktury danych — sterty, podobnie jak metodę tzw. *tablic równoległych* do reprezentacji list z wieloma kryteriami sortowania. Jak widać, inteligentne użycie tablic może nam podsunąć możliwości z trudem uzyskiwane w przypadku optymalnych, listowych struktur danych.

Popatrzmy dla przykładu na implementację tablicową drzew, w których nie są zapamiętywane informacje dotyczące *potomków* danego węzła (tzn. nie interesuje nas zstępowanie w stronę liści), ale informacje o *rodzicach* danego potomka.

Terminologia używająca określać: *ojciec*, *syn*, *potomek lewy*, *potomek prawy*, etc. jest ogólnie spotykana w książkach poświęconych strukturom drzewiastym — również anglojęzycznych. W tym miejscu warto być może przytoczyć anegdotę dotyczącą właśnie tego typu określeń, które mogą osoby nieprzyzwyczajone prowadzić do konfuzji. W 1993 roku uczestniczyłem w kursie języka angielskiego przeznaczonym dla Francuzów i prowadzonym przez przybyłą do Francji Amerykankę o dość ekstrawaganckim sposobie bycia. W trakcie kursu należało przygotować małe exposé na dowolny w zasadzie, ale techniczny, temat. Jeden z francuskich studentów omówił pewien algorytm dotyczący rozproszonych baz danych, w którym dość sporo miejsca zajmowało wyjaśnienie drzewiastej struktury danych, służącej do reprezentacji pewnych istotnych dla algorytmu danych. Terminologia, której używał do opisu drzewa, była identyczna z zaprezentowaną powyżej: ojciec, syn, potomek itp. Anglosasi są ogólnie dość uczuleni na punkcie jawnego rozróżniania form osobowych (on, ona) od bezosobowych, obejmujących w zasadzie wszystko oprócz osób (określane w sposób ogólny zaimkiem *it*). Student, o którym jest mowa, omawiał coś o charakterze bez wątpienia bezosobowym — strukturę danych, ale od czasu do czasu używał określeń zarezerwowanych normalnie dla istot ludzkich — ojciec, syn... Amerykanka słuchała jego przemowy przez dobrych kilka minut, otwierając coraz szerzej oczy, aż w końcu nie wytrzymała, wyskoczyła na środek klasy i przerwała Francuzowi: „What father? What child? Please show me where is the *zizi*²² here!” — pokazując jednocześnie na narysowane na tablicy drzewo...

Ale wróćmy do tematu i pokażmy wreszcie obiecaną implementację drzew za pomocą tablic, tak aby uzyskać informację o węzłach ojcach. Rysunek 5.24 przedstawia drzewo służące do zapamiętywania liter (czyli pole `val` jest typu `char`).

Rysunek 5.24.
Tablicowa
reprezentacja drzewa



Numery znajdujące się przy węzłach mają charakter wyłącznie ilustracyjny — ich wybór jest raczej dowolny i nie podlega żadnym szczególnym regułom (chyba że sobie sami je wymyślimy na użytek konkretnej aplikacji). W ramach kolejnej konwencji umówmy się, że jeśli `ojciec[x]` jest równy `x`, to mamy do czynienia z pierwszym elementem drzewa.

Teraz, gdy już wiemy, jak reprezentować drzewa, wykorzystując dostępne w C++ (oraz w każdym nowoczesnym języku programowania) mechanizmy, popatrzmy na możliwe sposoby przechadzania się po gałęziach drzew.

²² W ten sposób francuskie dzieci określają nieodłączny atrybut każdego mężczyzny.

Drzewa binarne i wyrażenia arytmetyczne

Nasze rozważania o drzewach będziemy prowadzić poprzez prezentację dość rozbudowanego przykładu, na podstawie którego zobrazowane zostaną fenomeny, z którymi programista może się zetknąć, oraz mechanizmy, z których będzie on musiał sprawnie korzystać w celu efektywnego wykorzystania nowo poznanej struktury danych.

Problematyka będzie dotyczyła kwestii zasygnalizowanej już na rysunku 5.23. Zobaczyliśmy tam, że drzewo doskonale się nadaje do reprezentacji informatycznej wyrażeń arytmetycznych, bardzo naturalnie zapamiętując nie tylko informacje zawarte w wyrażeniu (tzn. *operandy* i *operatory*), ale i ich logiczną strukturę, która daje się poglądowo przedstawić właśnie w postaci drzewa.

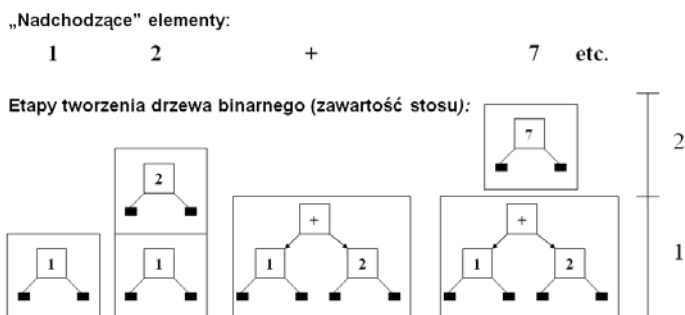
Przypomnijmy jeszcze raz typ komórki, który może służyć — zgodnie z ideą przedstawioną na rysunku 5.23 — do zapamiętywania zarówno operatorów (ograniczmy się tu do: +, -, * i do dzielenia wyrażonego za pomocą znaku dwukropka lub /), jak i operandów (liczb rzeczywistych).

```
struct wyrażenie
{
    double val;
    char op;
    wyrażenie *lewy, *prawy;
};
```

Inicjacja takiej komórki determinuje późniejszą interpretację jej zawartości. Jeśli w polu *op* zapamiętamy wartość 0, to będziemy uważali, że komórka nie jest operatorem i wartość zapamiętana w polu *val* ma sens. W odwrotnym zaś przypadku będziemy zajmowali się wyłącznie polem *op* bez zwracania uwagi na to, co znajduje się w *val*. Popatrzmy na rysunek 5.25, który ukazuje kilka pierwszych etapów tworzenia drzewa binarnego wyrażenia arytmetycznego.

Rysunek 5.25.

Tworzenie drzewa binarnego wyrażenia arytmetycznego



Do tworzenia drzewa użyjemy dobrze nam znanego z poprzednich dyskusji stosu (patrz: strona 119). Tym razem będzie on służył do zapamiętywania wskaźników do rekordów typu `struct wyrażenie`, co implikuje jego deklarację przez `STOS<wyrażenie*> s` (Jak widać, warto było raz się pomeścić i stworzyć stos w postaci klasy szablonowej).

Typowe wyrażenie arytmetyczne, zapisane w powszechnie używanej postaci (zwanej po polsku wrostkową), da się również przedstawić w tzw. *Odwrotnej Notacji Polskiej* (ONP, postfiksowej). Zamiast pisać *a op b*, używamy formy: *a b op*. Mówiąc krótko: operator występuje po swoich argumentach. Operacja arytmetyczna jest łatwa do odtworzenia w postaci klasycznej, jeśli wiemy, ile operandów wymaga dany operator.

Analiza wyrażenia beznawiasowego odbywa się w następujący sposób:

- ♦ Czytamy argumenty znak po znaku, odkładając je na stos.
- ♦ W momencie pojawienia się jakiegoś operatora ze stosu zdejmowana jest odpowiednia dlań liczba argumentów — wynik operacji kładziony jest na stos jako kolejny argument.

Na rysunku 5.25 możemy zaobserwować opisany wyżej proces w bardziej poglądowej formie niż powyższy suchy opis. Pierwsze dwa argumenty, 1 i 2, jako niebędące operatorami, są odkładane na stos (w programie odpowiadać to będzie stworzeniu dwóch komórek pamięci, których pola wskaźnikowe lewy i prawy są zainicjowane wartościami NULL). Trzecim elementem, który przybywa z zewnątrz, jest operator +. Tworzona jest nowa komórka pamięci, jednocześnie sam fakt nadejścia operatora prowokuje zdjęcie ze stosu dwóch argumentów, którymi są komórki zawierające liczby 1 i 2. Te komórki są doczepiane do pól wskaźnikowych komórki zawierającej operator +. Kolejnym nadchodzącym elementem jest znowu liczba 7 — jest ona odkładana na stos i proces może być kontynuowany.

W opisany powyżej sposób pracują kompilatory w momencie obliczania wyrażeń za pośrednictwem stosu. Jedyną różnicą jest to, że odkładane na stos są nie kolejne poddrzewa, ale już obliczone fragmenty dowolnie w zasadzie skomplikowanych wyrażeń arytmetycznych. Czytelnik zgodzi się chyba ze stwierdzeniem, że z punktu widzenia komputera ONP jest istotnie bardzo wygodna w użyciu²³.

Przypatrzmy się konkretnym instrukcjom w C++, które zajmują się inicjacją drzewa binarnego.



wyrazen.cpp

```
typedef struct
{
    double val;
    char op;
} VAL;

int main()
{
    STOS<wyrazenie*> s;
    // Przykład POPRAWNEJ sekwencji danych, w przypadku sekwencji
    // błędnej, gdy np. zabraknie drugiego operanda, otrzymane drzewo
    // będzie również bezsensowne (proszę wykonać odpowiednie próby)
    VAL t[9]={ {2, '0'}, {3, '0'}, {0, '+'}, {7, '0'}, {9, '0'}, {0, '*'}, {0, '+'},
               {12.5, '0'}, {0, '*'} };
    wyrazenie *x;
    for(int i=0; i<9; i++)
    {
        x=new wyrazenie;
        if( (t[i].op=='*') || (t[i].op=='+') || (t[i].op=='-')
            || (t[i].op=='/') || (t[i].op==':') )
            x->op =t[i].op;
        else
        {
            x->val=t[i].val;
            x->op='0'; // Umowna konwencja oznaczająca wartość, a nie operator
        }
        x->lewy =NULL;
        x->prawy=NULL;
        if((t[i].op=='*') || (t[i].op=='+') || (t[i].op=='-')
            || (t[i].op=='/') || (t[i].op==':') )
        {
            wyrazenie *l1, *p1;
            s.pop(l1);
            s.pop(p1);
            x->lewy =l1; // „Podwiązanie” pod węzeł x
```

²³ Wbrew pozorom notacja ONP jest dość wszechstronnie stosowana w pewnych obszarach, patrz np. kalkulatory firmy Hewlett Packard, język Forth, język opisu stron drukarek laserowych Postscript. W pewnych kręgach jest to zatem dość znana notacja.


```

    x->prawy=p1; // „Podwiązanie” pod węzeł x
    }
    s.push(x);
}
pisz_infix(x):cout << "=" << oblicz(x) << endl; // Omówione dalej
pisz_prefix(x):cout << "=" << oblicz(x) << endl; // Omówione dalej
}

```

W powyższym listingu tablica *t* zawiera *poprawną* sekwencję danych, tzn. taką, która istotnie stworzy drzewo binarne mające sens. Warto odrobinę poeksperymentować z zawartością tablicy, aby zobaczyć, jak algorytm zareaguje na błędny ciąg danych. Można się spodziewać, że w przypadku np. braku drugiego operandu lub operatora otrzymane rezultaty będą również błędne — jest to prawda, ale najlepiej przekonać się o tym na własnej skórze.

Jak jednak obejrzeć zawartość drzewa, które tak pieczołowicie stworzyliśmy? Wbrew pozorom zadanie jest trywialne i sprowadza się do wykorzystania własności „topograficznych” drzewa binarnego. Sposób interpretacji formy wyrażenia (czy jest ona *infixowa*, *prefiksowa* czy też *postfixowa*) zależy bowiem tylko i wyłącznie od sposobu przechodzenia przez gałęzie drzewa!

Popatrzmy na realizację funkcji służącej do wypisywania drzewa w postaci klasycznej, tzn. wrostkowej. Jej działanie można wyrazić w postaci prostego algorytmu rekurencyjnego:

```

wypisz(w)
{
    jeśli wyrażenie w jest liczbą. to wypisz ją:
    jeśli wyrażenie w jest operatorem op. to wypisz je w kolejności:
        (wypisz(w -> left) op wypisz(w -> right))
}

```

Realizacja programowa jest oczywiście dosłownym tłumaczeniem powyższego zapisu:

```

void pisz_infix(struct wyrażenie *w)
{ //funkcja wypisuje wyrażenie w postaci wrostkowej
    if(w->op=='0') //wartość liczbową...
        cout << w->val;
    else
    {
        cout << "(";
        pisz_infix(w->lewy);
        cout << w->op;
        pisz_infix(w->prawy);
        cout << ")";
    }
}

```

W analogiczny sposób możemy zrealizować algorytm wypisujący wyrażenie w formie beznawiasowej, czyli ONP:

```

void pisz_prefix(struct wyrażenie *w)
{ //funkcja wypisuje wyrażenie w postaci prefiksowej
    if(w->op=='0') //wartość liczbową...
        cout<<w->val<<" ";
    else
    {
        cout << w->op << " ";
        pisz_prefix(w->lewy);
        pisz_prefix(w->prawy);
    }
}

```

Jak łatwo zauważyć, w zależności od sposobu przechadzania się po drzewie możemy w różny sposób przedstawić jego zawartość bez wykonywania jakiegokolwiek zmiany w strukturze samego drzewa!

Reprezentacja wyrażeń arytmetycznych byłaby z pewnością niekompletna, gdybyśmy nie uzupełnili jej funkcjami wykonującymi operacje na tychże wyrażeniach. Zanim jednak cokolwiek zechcemy obliczać, musimy dysponować funkcją, która sprawdzi, czy wyrażenie znajdujące się w drzewie jest prawidłowo skonstruowane, tzn. czy przykładowo nie zawiera nieznanego nam operatora arytmetycznego.

Zauważmy, że o poprawności drzewa decyduje już sam sposób jego konstruowania z użyciem stosu. Pomimo tego ułatwienia dysponowanie dodatkową funkcją sprawdzającą poprawność drzewa jest jednak mało kosztowne — dosłownie kilka linijek kodu — a użyteczność takiej dodatkowej funkcji jest oczywista.

```
int poprawne(struct wyrazienie *w)
{
    // czy wyrażenie jest poprawne składniowo?
    if (w->op == '0')
        return 1; // OK, wg naszej konwencji jest to liczba
    switch (w->op)
    {
        case '+':
        case '-':
        case '*':
        case '/': // to są znane operatory
        case '(': return (poprawne(w->lewy) * poprawne(w->prawy));
        default: return (0); // błąd, operator nieznan!
    }
}
```

Nie będę nikogo zachęcał do zrealizowania powyższych funkcji w formie iteracyjnej — jest to oczywiście wykonalne, ale rezultat nie należy do specjalnie czytelnych i eleganckich.

Przejdźmy wreszcie do prezentacji funkcji, która zajmie się obliczeniem wartości wyrażenia arytmetycznego. Jej schemat jest bardzo zbliżony do zastosowanego w funkcji `poprawne`:

```
double oblicz(struct wyrazienie *w)
{
    if (poprawne(w)) // wyrażenie poprawne?
        if (w->op == '0')
            return (w->val); // pojedyncza wartość
        else
            switch (w->op)
            {
                case '+': return oblicz(w->lewy) + oblicz(w->prawy);
                case '-': return oblicz(w->lewy) - oblicz(w->prawy);
                case '*': return oblicz(w->lewy) * oblicz(w->prawy);
                case '/': if (oblicz(w->prawy) != 0)
                    return (oblicz(w->lewy) / oblicz(w->prawy));
                    else
                    {
                        cerr << "\nDzielenie przez zero!\n";
                        return -1; // ułomna sygnalizacja błędów
                    }
            }
        else cerr << "Błąd składni....!\n";
    }
}
```

Dla dopełnienia prezentacji tego dość sporego kawałka kodu popatrzmy na rezultaty wykonania funkcji `main`:

```
(12.5*((9*7)+(3+2)))=850
* 12.5 + * 9 7 + 3 2 =850
```

Zachęcam Czytelnika do kontynuowania eksperymentów z drzewiastymi strukturami danych, bowiem temat jest pasjonujący, a rezultaty potrafią zrobić wrażenie.

Uniwersalna struktura słownikowa

Nasze rozważania poświęcone strukturom drzewiastym zakończymy, prezentując szczegółową implementację tzw. *Uniwersalnej Struktury Słownikowej* (określanej dalej jako *USS*). Jest to dość złożony przykład wykorzystania możliwości, jakie oferują drzewa, i nawet jeśli Czytelnik nie będzie miał w praktyce okazji skorzystać z *USS*, to zawarte w tym paragrafie informacje i techniki będą mogły zostać wykorzystane przy rozwiązywaniu innych problemów, w których w grę wchodzi zbliżone kwestie.

Z uwagi na czytelność wyjaśnień wszelkie przykłady dotyczące *USS* będą tymczasowo obywaty się bez poruszania zagadnienia polskich znaków diakrytycznych: *ą, ę, ć*, etc. Temat ten poruszę dopiero pod koniec tego paragrafu, gdzie zaproponuję prosty sposób rozwiązania niniejszego problemu — w istocie będą to niewielkie, wręcz kosmetyczne modyfikacje zaprezentowanych już za moment algorytmów.

Najwyższa już pora wyjaśnić właściwy temat naszych rozważań. Otóż wiele programów z różnych dziedzin, ale operujących tekstem wprowadzanym przez użytkownika, może posiadać funkcję sprawdzania poprawności ortograficznej wprowadzanych pieczołowicie informacji (patrz np. arkusze kalkulacyjne, edytory tekstu). Całkiem prawdopodobne jest, iż wielu Czytelników chciałoby móc zrealizować w swoich programach taki mały weryfikator, jednak z uwagi na znaczne skomplikowanie problemu nawet się do niego nie przysmierzają. W istocie z problemem weryfikacji ortograficznej są ściśle związane następujące pytania, na które odpowiedź wcale nie jest jednoznaczna i prosta:

- ♦ Jakich struktur danych używać do reprezentacji słownika?
- ♦ Jak zapisać słownik na dysku?
- ♦ Jak wczytać słownik „bazowy” do pamięci?
- ♦ Jak uaktualniać zawartość słownika?

Konia z rzędem temu, kto bez wahania ma gotowe odpowiedzi na te pytania! Oczywiście na wszystkie naraz, bowiem nierozwiązanie na przykład problemu zapisu na dysk czyni resztę całkowicie bezużyteczną.

Ze wszelkiego rodzaju słownikami wiąże się również problem ich niebagatelnej objętości. O ile jeszcze możemy się łatwo pogodzić z zajętością miejsca na dysku, to w przypadku pamięci komputera decyzja już nie jest taka prosta — średniej wielkości słownik ortograficzny może z łatwością „zatkać” całą dostępną pamięć i nie pozostawić miejsca na właściwy program. No chyba że ma on wypisywać komunikat: „Out of memory”²⁴... Sprawy komplikują się niepomniemie, jeśli w grę wchodzi tak bogaty język, jakim jest np. nasz ojczysty — z jego mnogimi formami deklinacyjnymi, wyjątkami od wyjątków, etc. Zapamiętanie tego wszystkiego bez odpowiedniej kompresji danych może okazać się po prostu niewykonalne.

Istnieją liczne metody kompresji danych, większość z nich ma jednak charakter archiwizacyjny — służący do przechowywania, a nie do dynamicznego operowania danymi. Marzeniem byłoby posiadanie struktury danych, która przez swoją naturę automatycznie zapewnia kompresję danych już w pamięci komputera, nie ograniczając dostępu do zapamiętanych informacji.

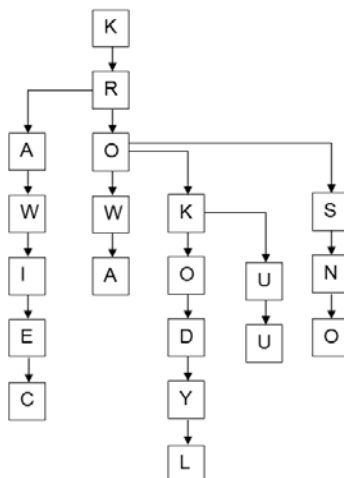
Prawdopodobnie wszyscy Czytelnicy domyślili się natychmiast, że *USS* należy do tego typu struktur danych.

Idea *USS* opiera się na następującej obserwacji: wiele słów posiada te same rdzenie (przedrostki), różniąc się jedynie końcówkami (przyrostkami). Przykładowo weźmy pod uwagę następującą grupę słów: KROKUS, KROSNO, KRAWIEC, KROKODYL, KRAJ. Gdyby można było zapisać je w pamięci w formie drzewa przedstawionego na rysunku 5.26, to problem kompresji mielibyśmy z głowy. Z 31 znaków do zapamiętania zrobiło nam się raptem 21, co może nie

²⁴ Pol. *Brak pamięci*.

Rysunek 5.26.

Kompresja danych
zaletą Uniwersalnej
Struktury Słownikowej



oszałamia, ale pozwala przypuszczać, że w przypadku rozbudowanych słowników zysk byłby jeszcze większy. Zakładamy oczywiście, że w słowniku będą zapamiętywane w dużej części serie słów zaczynających się od tych samych liter — czyli przykładowo pełne odmiany rzeczowników, etc.

Pora już na przedstawienie owej tajemniczej *USS* w szczegółach. Jej realizacja jest nieco przewrotna, bowiem zbędne staje się zapamiętywanie słów i ich fragmentów, a pomimo tego cel i tak zostaje osiągnięty!

Program zaprezentuję w szczegółowo skomentowanych fragmentach. Oto pierwszy z nich zawierający programową realizację *USS*:

**uss.cpp**

```

const int n=29;

typedef struct słownik
{
    struct słownik *t[n];
} USS, *USS_PTR;
```

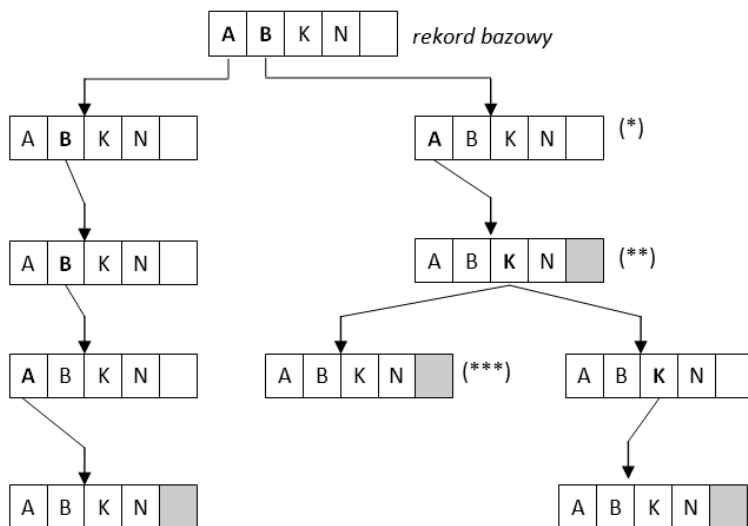
Mamy oto typową dla C++ deklarację typu rekurencyjnego, którego jedynym elementem jest tablica wskaźników do tegoż właśnie typu. (Tak, zdaję sobie sprawę, iż brzmi to okropnie). Litera *a* (lub *A*) odpowiada komórka `t[0]`, analogicznie literom *z* (lub *Z*) komórka `t[25]`. Dodatkowe komórki pamięci będą służyły znakom specjalnym, które nie należą do podstawowych liter alfabetu, ale dość często wchodzą w skład słów (np. myślnik, polskie znaki diakrytyczne itp.).

W celu oszczędności miejsca słowa będą zapamiętywane już w postaci przetransformowanej na duże litery. Słowo *odpowiada* jest tu bardzo charakterystyczne, bowiem słowa nie są w *USS* zapamiętywane bezpośrednio.



Zapętnienie wskaźnika `t[n-1]` do swojej własnej tablicy oznacza znacznik końca słowa.

Dokładną zasadę działania *USS* wyjaśnimy na przykładzie zamieszczonym na rysunku 5.27.

Rysunek 5.27.Reprezentacja
słów w USS

Założeniem przyjętym podczas analizy niech będzie ograniczenie liczby liter alfabetu do 4: A, B, K, N. USS zawiera tablicę *t* o rozmiarze 5: ostatnia komórka służy jako znacznik końca słowa. Jeśli wskaźnik w *t*[4] wskazuje na *t*, oznacza to, że w tym miejscu pewne słowo zawiera swój znacznik końca. Które dokładnie? Spójrzmy jeszcze raz na rysunek 5.26. Komórka nazwana pierwotną umożliwia dostęp do wszystkich słów naszego 4-literowego alfabetu. Wskaźnik znajdujący się w *t*[1] (czyli *t*['B']) zawiera adres komórki oznaczonej jako (*). Znajdujący się w niej wskaźnik *t*[0] (czyli *t*['A']) wskazuje na (**). Tu uwaga! W komórce (**) *t*[4] jest zapętlony, czyli znajduje się tu znacznik końca słowa, na którego litery składały się odwiedzane ostatnio indeksy: najpierw 'B', potem 'A', na koniec znacznik końca słowa — co daje razem słowo BA.

Proces przechadzania się po drzewie nie jest bynajmniej zakończony: od komórki (**) odchodzi strzałka do (***), w której także następuje zapętlenie. Jakie słowo teraz przeczytaliśmy? Oczywiście BAK! Rozumując podobnie, możemy przeczytać jeszcze słowa BANK i ABBA.

Idea USS, dość trudna do wyrażenia bez poparcia rysunkiem, jest zaskakująco prosta w realizacji końcowej, w postaci programu wynikowego. Oczywiście nie stworzymy tutaj kompletnego modułu obsługi słownika, ale ta reszta, której brakuje (obsługa zapisu danych na dysk, „ładne” procedury wyświetlania, etc.), to już tylko zwykła praca wykończeniowa.

Omówmy po kolei procedury tworzące zasadniczy szkielet modułu obsługi USS.

Funkcje *do_indeksu* i *z_indeksu* pełnią role translacyjne. Z indeksów liczbowych tablicy *t* (elementu składowego rekordu USS) możemy odtworzyć odpowiadające poszczególnym pozycjom litery i vice versa. To właśnie zwiększając wartość stałej *n* oraz nieco modyfikując te dwie funkcje, możemy do modułu obsługującego USS dołączyć znajomość polskich znaków!

```
int do_indeksu(char c) //znak ASCII -> indeks
{
    if ( (c<='Z') && (c>='A') || (c<='z') && (c>='a') )
        return toupper(c)-'A'; //toupper = zamiana malej litery na duza
    else
    {
        if (c==' ') return 26;
        if (c=='-') return 27;
    }
}

char z_indeksu(int n) //indeks -> znak ASCII
{

```

```

if ( n>=0 && n<= ('Z'-'A') )
    return toupper((char) n+'A');
else
{
    if (n==26) return ' ';
    if (n==27) return '-';
}
}

```

Funkcja zapisz otrzymuje wskaźnik do pierwszej komórki słownika. Zanim zostanie stworzona nowa komórka pamięci, funkcja ta sprawdzi, czy aby jest to na pewno niezbędne. Przykładowo niech w drzewie USS istnieje już słowo *ALFABET*, a my chcemy doń dopisać imię sympatycznego kosmity ze znanego amerykańskiego serialu: ALF. Otóż wszystkie poziomy odpowiadające literom 'A', 'L' i 'F' już istnieją — w konsekwencji żadne nowe komórki pamięci nie zostaną stworzone. Jedynie na poziomie litery 'F' zostanie utworzona komórka, w której do `t[n-1]` zostanie wpisany wskaźnik „do siebie”. Przypomnijmy, że ten ostatni służy jako znacznik końca słowa.

```

void zapisz(char *slovo, USS_PTR p)
{
    USS_PTR q; // zmienna pomocnicza
    int pos;
    for (int i=1; i<=strlen(slovo); i++)
    {
        pos=do_indeksu(slovo[i-1]);
        if (p->t[pos] != NULL) p=p->t[pos];
        else
        {
            q=new USS;
            p->t[pos]=q;
            for (int k=0; k<n; q->t[k++]=NULL);
            p=q;
        }
    }
    p->t[n-1]=p; //pętla jako koniec słowa
}

```

Funkcja `pisz_sloownik` służy do wypisania zawartości słownika — być może nie w najczytelniejszej formie, ale można się dość łatwo zorientować, jakie słowa zostały zapamiętane w USS.

```

void pisz_sloownik(USS_PTR p)
{
    for (int i=0; i<26; i++)
        if (p->t[i] != NULL)
        {
            if ( (p->t[i])->t[n-1]==p->t[i]) // gdy koniec słowa to
                cout << z_indeksu(i) << endl << " ";//pisz znak końca linii
            else
                cout << z_indeksu(i);
            cout << " ---"; // aby ładniej wyglądało
            pisz_sloownik(p->t[i]); // wypisz rekurencyjnie resztę
        }
}

```

Funkcja `szukaj` realizuje dość oczywisty algorytm szukania pewnego słowa w drzewie: jeśli przejdziemy wszelkie gałęzie (poziomy) odpowiadające literom poszukiwanego słowa i trafimy na znacznik końca tekstu, to wynik jest chyba oczywisty!

```

void szukaj(char *slovo, USS_PTR p)
{
    // szukaj słowa w słowniku
    int test=1, i=0;
    while ((test==1) && (i<strlen(slovo)) )
    {
        if (p->t[do_indeksu(slovo[i])]==NULL)
            test=0; // brak odgałęzienia, słowa nie ma!
    }
}

```

```

    else
        p=p->t[do_indeksu(slowo[i++])]; //szukamy dalej
    }
    if ( (i==strlen(slowo)) && (p->t[n-1]==p) && test)
        cout << "Słowo znalezione!\n";
    else
        cout << "Słowo nie zostało znalezione w słowniku!\n";
}

```

Oto przykładowa funkcja main:

```

int main()
{
    int i;
    char tresc[100];
    USS_PTR p=new USS; //tworzymy nowy słownik
    for(i=0; i<n; p->t[i++]=NULL);
    for(i=1; i<=7; i++) //wczytamy 7 słów
    {
        cout << "Podaj słowo, które mam umieścić w słowniku:";
        cin >> tresc;
        zapisz(tresc,p);
    }
    pisz_słownik(p); //wypisujemy słownik
    for(i=1 ;i<=4;i++) //szukamy 4 słów
    {
        cout << "Podaj słowo, które mam poszukać w słowniku:";
        cin >> tresc;
        szukaj(tresc,p);
    }
}

```

Przypuśćmy, że podczas sesji z programem wpisaliśmy następujące słowa: alf, alfabet, alfabetycznie, anagram, anonim, ASTRonoMIa, Ankara (wielkie i małe litery zostały celowo pomieszczone ze sobą). Po wczytaniu tej serii program powinien wypisać zawartość słownika w dość dziwnej, co prawda, ale w miarę czytelnej formie, która ukazuje rzeczywistą konstrukcję drzewa USS dla tego przykładu:

```

A-L-F
-A-B-E-T
-Y-C-Z-N-I-E
-N-A-G-R-A-M
-K-A-R-A
-O-N-I-M
-S-T-R-O-N-O-M-I-A

```

Zbiory

Implementacja programowa zbiorów matematycznych napotyka na szereg ograniczeń związanych z używanym językiem programowania.

Miłośnicy Pascala znają zapewne definicje zbliżone do:

```

type Litery = 'A'..'Z';
ZbiorLiter = set of Litery;
var Alfabet:ZbiorLiter;
    c: char;
begin
    Alfabet:=['A'..'Z'];
    read(c);
    if c in Alfabet then {itd.}
end.

```

Oczywiście to, co dla programisty pascalowego jest zbiorem, wcale nim nie jest dla matematyka z uwagi na wymóg *jednakowego typu* zapamiętywanych elementów.

Niemniej jednak dla podstawowych zastosowań konwencje istniejące w Pascalu nadają się znakomicie, gdyż możliwe jest np. wykonywanie operacji typu: dodawanie elementu do zbioru, mnożenie (iloczyn) zbiorów, odejmowanie zbiorów, sprawdzanie, czy obiekt należy do zbioru itd.

W tej książce do opisu algorytmów i prezentacji struktur danych używamy języka C++, który na ogół spełnia swoje zadanie dość dobrze. Niestety nie posiada on wbudowanej obsługi zbiorów i w związku z tym należy ją dołożyć w sposób jawny, używając przy okazji różnorodnych technik, zależnych od aktualnie realizowanych zadań.

Weźmy dla przykładu *implementację zbioru znaków*, która nie wymaga użycia struktur listowych i dynamicznego przydzielania pamięci. Założmy, że w komputerze występuje „tylko” 256 znaków (między innymi znaki alfabetu duże i małe, cyfry oraz tzw. znaki kontrolne niedrukowalne).

Do zasymulowania zbioru wystarczy wówczas najzwyklejsza tablica typu unsigned char, tak jak w przykładzie poniżej:



set.cpp

```
class Zbior
{
private:
    unsigned char zbior[256]; // cała tablica ASCII
public:
    Zbior()
    { //zerowanie zbioru w konstruktorze
        for(int i=0; i<256; i++)
            zbior[i]=0;
    }
    Zbior& operator +(unsigned char c)
    { // dodaj znak 'c' do zbioru i zwróć zmieniony obiekt
        zbior[c]=1;
        return *this;
    }
    Zbior& operator -(unsigned char c)
    { // usuń znak 'c' ze zbioru i zwróć zmieniony obiekt
        zbior[c]=0;
        return *this;
    }
    bool nalezy(unsigned char c) // czy 'c' należy do zbioru?
    {
        return zbior[c]==1;
    }

    Zbior& dodaj(Zbior s2) // dodaj zawartość zbioru 's2' do obiektu
    {
        for(int i=0; i<256; i++)
            if(s2.nalezy(i)) // jeśli element obecny w s2
                zbior[i]=1; // dodaj go do zbioru
        return *this; // zwraca zmodyfikowany obiekt
    }

    int ile() // zwraca liczbę elementów w zbiorze
    {
        int n;
        for(int i=0; i<256; i++)
            if(zbior[i]==1) // element obecny
                n++;
        return n;
    }
}
```



```

void pisz()
{ // wypisuje zawartość zbioru
    int i;
    cout << "{";
    for(i=0; i<256; i++)
        if(zbior[i]!=1) // wypisz obecny element
            cout << (char)i << " ";
    if(i==0)
        cout << "Zbiór pusty!";
    cout << "}\n";
}
}; // koniec definicji klasy Zbiór

```

Pomimo dużej prostoty powyższa implementacja umożliwia już manipulacje typowe dla zbiorów:

```

int main()
{
    Zbiór s1, s2;
    s1=s1+'A'; s1=s1+'A'; s1=s1+'B'; s1=s1+'C';
    s2=s2+'B'; s2=s2+'B'; s2=s2+'E'; s2=s2+'F';
    cout << "Zbiór S1 ="; s1.pisz();
    s1=s1-'C';
    cout << "Zbiór S1 - 'C' ="; s1.pisz();
    cout << "Zbiór S2 ="; s2.pisz();
    s1.dodaj(s2);
    cout << "Zbiór S1 + S2 = ";
    s1.pisz();
}

```

Uruchomienie programu powinno spowodować wyświetlenie na ekranie następujących komunikatów:

```

Zbiór S1 = { A B C }
Zbiór S1 - 'C' = { A B }
Zbiór S2 = { B E F }
Zbiór S1 + S2 = { A B E F }

```

Czytelnik z łatwością uzupełni samodzielnie operacje dostępne w powyższej implementacji klasy Zbiór o przecinanie (iloczyn) i odejmowanie zbiorów.

Możliwe jest oczywiście stworzenie bardziej ogólnej implementacji zbiorów, akceptującej zmienną liczbę danych (wymaga dynamicznego przydziału pamięci, np. za pomocą list) i zezwalającej na złożone elementy składowe, np. rekordy danych. Wydaje się jednak, że zaprojektowanie klasy Zbiór z użyciem klas szablonowych i list byłoby nadużyciem siły, w przypadku gdy jedynymi niezbędnymi nam elementami zbiorów miałyby zostać jedynie... znaki alfabetu!

Zadania

Zadanie 1.

Zastanów się, jak można w prosty sposób zmodyfikować model *Uniwersalnej Struktury Słownikowej* (patrz strona 140), aby możliwe było jej użycie jako słownika 2-języcznego, np. polsko-angielskiego. Oszacuj wzrost kosztu słownika (chodzi o ilość zużytej pamięci) dla następujących danych: 6 000 rekordów *USS* w pamięci zawierających 25 000 zapamiętanych słów.

Zadanie 2.

Zestaw dość podobnych zadań. Napisz funkcje, które usuwają:

- ♦ pierwszy element listy;
- ♦ ostatni element listy;

- ◆ pewien element listy, który odpowiada kryteriom poszukiwań podanym jako parametr funkcji (aby uczynić funkcję uniwersalną, wykorzystaj metodę przekazania wskaźnika funkcji jako parametru).

Zadanie 3.

Napisz funkcję, która:

- ◆ Zwraca liczbę elementów listy (a).
- ◆ Zwraca k -ty element listy (b).
- ◆ Usuwa k -ty element listy (c).



Podczas rozwiązywania zadań 2. i 3. proszę dokładnie przemyśleć efektywny sposób informowania o sytuacjach błędnych (np. próba usunięcia k -tego elementu, podczas gdy on nie istnieje, etc.).

Rozwiązania zadań

Zadanie 1.

Modyfikacja struktury *USS*:

```
typedef struct slownik
{
    struct slownik  *t[n];
    char            *tlumaczenie;
} USS,*USS_PTR;
```

Tłumaczenie jest „dopisywane” (alokowane) do funkcji zapisz podczas zaznaczania końca słowa — w ten sposób nie tracimy związku *słowo-tłumaczenie*.

Koszt:

- ◆ bez drugiego języka:
 - ◆ Koszt = $(n = 29) \cdot 4$ bajty („duży” model pamięci) = 696 000 bajtów = ok. 679 kB.
- ◆ z drugim językiem:
 - ◆ Założenie: średnia długość słowa angielskiego wynosi 9 bajtów + ogranicznik, czyli 10 bajtów.
 - ◆ Koszt = przypadek poprzedni plus $25\,000 \cdot 10$ plus pewna liczba nieużytych wskaźników na tłumaczenie — przyjmijmy zaokrąglenie na 1 000. Ostatecznie mamy: $25\,000 \cdot 10 + 1\,000 \cdot 4 = 254\,000$ bajtów, czyli ok. 248 kB.

W danym przypadku koszt wzrósł o ok. 36% pierwotnej zajętości pamięci.

Zadanie 3.

Oto propozycja rozwiązania zadania 3a:

```
int cpt(ELEMENT *q, int res=0)
{
    if (glowa==NULL)
        return res;
    else
        cpt(q->nastepny,res+1);
}
```

Przykładowe wywołanie: `int ilosc=cpt(inf -> glowa).`

Rozdział 6.

Derekursywacja i optymalizacja algorytmów

Podjęcie tematu przekształcania algorytmów rekurencyjnych na ich postać iteracyjną — oczywiście równoważną funkcjonalnie! — jest logiczną konsekwencją omawiania rekurencji. Pomimo iż temat ten był kiedyś podejmowany wyłącznie na użytek języków nieumożliwiających programowania rekurencyjnego (FORTRAN, COBOL), nawet obecnie znajomość tych zagadnień może mieć pewne znaczenie praktyczne.

Sam fakt poruszenia tematu derekursywacji w książce poświęconej algorytmom i technikom programowania jest trochę ryzykowny — nie są to zagadnienia o charakterze czysto algorytmicznym. Mimo to w praktyce warto coś na ten temat wiedzieć, gdyż trudno odmówić derekursywacji znaczenia praktycznego. Skąd jednak wziął się sam pomysł takiego zabiegu? Programy wyrażone w formie rekurencyjnej są z natury rzeczy bardzo czytelne i raczej krótkie w zapisie. Nie trzeba być wybitnym specjalistą od programowania, aby się domyślić, iż wersje iteracyjne będą zarówno mniej czytelne, jak i po prostu dłuższe. Po cóż więc w ogóle podejmować się tego — zdawałoby się bezsensownego — zadania?

Rzeczywiście postawienie sprawy w ten sposób jest zniechęcające. Poznawszy kilka istotnych zalet stosowania technik rekurencyjnych, chcemy się teraz od tego całkowicie odwrócić plecami! Na szczęście nie jest aż tak źle, bowiem nikt tu nie ma zamiaru proponować rezygnacji z rekurencji. Nasze zadanie będzie wchodziło w zakres zwykłej optymalizacji kodu w celu usprawnienia jego wykonywania w rzeczywistym systemie operacyjnym na prawdziwym komputerze.

Piętą Achillesową większości funkcji rekurencyjnych jest intensywne wykorzystywanie stosu, który służy do odtwarzania „zamrożonych” egzemplarzy tej samej funkcji. Z każdym takim nieczynnym chwilowo egzemplarzem trzeba zachować pełny zestaw jego parametrów wywołania, zmiennych lokalnych czy wreszcie adres powrotu. To tyle, jeśli chodzi o samą zajętość pamięci. Nie zapominajmy jednak, że zarządzanie przez kompilator tym całym bałaganem kosztuje cenny czas procesora, który dodaje się do ogólnego czasu wykonania programu!

Pomysł jest zatem następujący: podczas tworzenia oprogramowania wykorzystajmy całą siłę i elegancję algorytmów rekurencyjnych, natomiast w momencie pisania wersji końcowej (tej, która ma być używana w praktyce), dokonajmy transformacji na analogiczną postać iteracyjną¹.

¹ Pod warunkiem że jest to konieczne z uwagi na parametry czasowe naszej aplikacji lub jej ograniczone zasoby pamięci. W każdym innym przypadku podejmowanie się derekursywacji ma sens raczej wątpliwy.

Z uwagi na to, że nie zawsze jest to proces oczywisty, warto poznać kilka standardowych sposobów używanych do tego celu.

Zaletą zabiegu transformacji jest pełna równoważność funkcjonalna. Implikuje to między innymi fakt, iż będąc pewnymi poprawności działania danego programu rekurencyjnego, nie musimy już udowadniać poprawności jego wersji iteracyjnej. Wyrażając to innymi słowy: dobry algorytm rekurencyjny nie ulegnie zepsuciu po swojej poprawnej transformacji.

Jak pracuje kompilator?

Języki strukturalne, pełne konstrukcji o wysokim poziomie abstrakcji, nie mogłyby spełniać w ogóle swojej roli, gdyby nie istniały kompilatory. Kompilatory są to również programy, które przetłumaczą nasze dzieła na postać zrozumiałą dla (mikro) procesora.

Dodajmy jeszcze, że efekt tego tłumaczenia marnie przypomina to, co z takim trudem napisaliśmy i uruchomiliśmy. Wyklęta ongiś instrukcja `goto` (a w każdym razie jej odpowiedniki) występuje w kodzie wynikowym częściej. Popatrzmy dla przykładu na tłumaczenie maszynowe² zwykłej instrukcji warunkowej:

```
if(warunek)
    Instr1;
else
    Instr2;
```

Jest to prosta instrukcja strukturalna, ale jej wykonanie musi być sekwencyjne:

```
if_not warunek goto et1
    ASM(Instr1)
    goto koniec
et1:
    ASM(Instr2)
koniec:
```

`ASM(instr)` oznacza ciąg instrukcji asemblerowych odpowiadających instrukcji `instr`, a `if`, `if_not` i `goto` są elementarnymi instrukcjami procesora (słowami kluczowymi języka asemblera).

Każdą dowolną instrukcję strukturalną można przetłumaczyć na jej postać sekwencyjną (rzeczywiste kompilatory tym właśnie między innymi się zajmują). Także w przypadku wywołań proceduralnych czynność ta, wbrew pozorom, nie jest skomplikowana. Przyjmując pewne uproszczenia, ciąg instrukcji:

```
Instr1:
P(x);
Instr2;
```

odpowiada, już po przetłumaczeniu przez kompilator, następującej sekwencji:

```
ASM(Instr1)
tmp=x
adr_powr=et1
goto et2
et1:
    ASM(Instr2);
et2:
    ASM(P(tmp));
...
goto adr_powr
```

² Przedstawione — oczywiście symbolicznie — za pomocą pseudokodu asemblerowego.

Czy w podany wyżej sposób da się również potraktować wywołania rekurencyjne (w procedurze P wywołujemy jeszcze raz P)? Oczywiście nie powielamy tyle razy fragmentu kodu odpowiadającego tekstowi P , aby obsłużyć wszystkie egzemplarze tej procedury — byłoby to absurdalne i niewykonalne w praktyce. Jedyne, co nam pozostaje, to zasymulować wywołanie rekurencyjne poprzez zwykłe wielokrotne użycie tego bloku instrukcji, który odpowiada procedurze P — z jednym wszakże zastrzeżeniem: wywołanie rekurencyjne nie może zacierać informacji, które są niezbędne do prawidłowego kontynuowania wykonywania programu.

Niestety sposób podany poprzednio nie spełnia tego warunku. Spójrzmy na przykład na następujący program rekurencyjny³:

```
P(int x)
{
  Instr1:
  P(F(x));
  Instr2:
}
```

Jak odróżnić powrót z procedury P , który powoduje definitywne jej zakończenie, od tego, który przekazuje kontrolę do Instr2 ? Okazuje się, że jedyny łatwy do zautomatyzowania sposób polega na użyciu tzw. stosu wywołań rekurencyjnych.

Zarządzanie powrotami z wywołań rekurencyjnych wymaga uprzedniego zapamiętywania dwóch informacji: tzw. otoczenia (np. wartości zmiennych lokalnych) i adresu powrotu, dobrze nam znanego z poprzedniego przykładu. Podczas wywołania rekurencyjnego następuje zapamiętanie na stosie tych informacji i kontrola jest oddawana procedurze. Jeśli wewnątrz niej nastąpi jeszcze raz wywołanie rekurencyjne, to na stos zostaną odłożone kolejne wartości otoczenia i adresu powrotu — różniące się od poprzednich. Podczas powrotu z procedury rekurencyjnej możliwe jest odtworzenie stanu zmiennych otoczenia sprzed wywołania poprzez zwykłe zdjęcie ich ze stosu.

Kompilator „wie”, gdzie ma nastąpić powrót, bowiem adres (argument instrukcji `goto`⁴) także został zapamiętany na stosie. Testując stan stosu, możemy określić moment zakończenia procedury: jeśli stos jest pusty, to wszystkie wywołania rekurencyjne już się wykonały.

Oto jak możliwe byłoby zrealizowanie w formie sekwencyjnej poprzedniego przykładu:

```
start:
  ASM(Instr1)
  push(Otoczenie, et1)
  x=F(x)                //procedura + wywołania
  goto start            //rekurencyjne
et1:
  ASM(Instr2)
  if_not(StosPusty)
  {
    pop(Otoczenie, Addr)
    Odtwórz(Otoczenie)
    goto Addr           //powroty z wywołań
                        //rekurencyjnych
  }
```

To tyle tytułem wstępu. W dalszej części rozdziału przystąpimy już do kilku prób tłumaczenia algorytmów rekurencyjnych na iteracyjne.

³ Funkcja F oznacza grupę przekształceń dokonywanych na parametrach funkcji.

⁴ Warto przypomnieć, że instrukcja `goto` istnieje również w C++.

Odrobina formalizmu nie zaszkodzi!

Mimo iż podręcznik ten bazuje na przykładach, od czasu do czasu warto przywdziać „garnitur naukowy” i zachowywać się dostojnie — a nic tak nie przekonuje o wadze tematu jak *Definicje* i *Twierdzenia*. Oto i one:



Definicja

Definicja 1.

Procedura iteracyjna I jest równoważna procedurze rekurencyjnej P , jeśli wykonuje dokładnie to samo zadanie, co P , dając identyczne rezultaty.

Przykładowo: dwie poniższe procedury `symetria1` i `symetria2` mogą być uważane za równoważne. Obie zajmują się dość blahym zadaniem rysowania szlaczka typu <<<<->>>> — o regulowanej przez parametr x szerokości.

```
void symetria1(int x)
{
    if (x==0)
        cout <<"-";
    else
    {
        cout <<"<";
        symetria1(x-1);
        cout <<">";
    }
}

void symetria2(int x)
{
    for(int i=1; i<=x;i++)
        cout<<"<";
    cout <<"-";
    for(i=1; i<=x;i++)
        cout <<">";
}
```



Definicja

Definicja 2.

Wywołanie rekurencyjne procedury P jest zwane *terminalnym* (ang. *end-recursion*), jeśli nie następuje po nim żadna instrukcja tej procedury.

Przykład:

```
void RecTerm(int n)
{
    if (x==0)
        cout <<".";
    else
    {
        cout <<"A";
        RecTerm(n-1);
    }
}
```



Uwaga

Wywołanie rekurencyjne procedury P zawarte w jakiegokolwiek pętli, np.:

```
void P(int n)
{
    ...
    while(V)
    {
        Instr1;
        P(n-1);
    }
}
```

nie jest uważane za terminalne, bowiem w zależności od warunku V , wywołanie $P(n-1)$ może, ale nie musi być wykonywane jako ostatnie.



Definicja

Twierdzenie 1.

Następujące procedury P1 i P2 są sobie wzajemnie równoważne pod warunkiem, że P1 zawiera tylko jedno rekurencyjne wywołanie terminalne.

```
void P1(x)
{
  if (Cond(x))
    Instr1(x);
  else
  {
    Instr2(x);
    P1(F(x))
  }
}
```

```
void P2(x)
{
  while(!Cond(x))
  {
    Instr2(x);
    x=F(x);
  }
  Instr1(x);
}
```

Kilka przykładów derekursywacji algorytmów

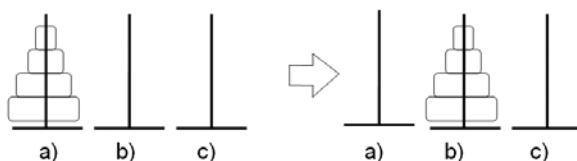
Wypróbujmy teraz świeżo nabytą wiedzę na „nieśmiertelnym” przykładzie tzw. wież *Hanoi*. Jest to łamigłówka o dość legendarnym rodowodzie — w co wnikać nie będziemy podczas naszych wywodów, koncentrując się raczej na problemie logicznym i sposobie rozwiązania go.

Zadanie jest następujące: mamy do dyspozycji n krążków o malejących średnicach, każdy z nich posiada wydrążoną dziurkę, która umożliwia nadzianie go na jeden z 3 wbitych w ziemię drążków. Na rysunku 6.1 jest przedstawiona sytuacja początkowa (z lewej strony) i końcowa (z prawej) dla 4 krążków.

Rysunek 6.1.

Wieże Hanoi

— prezentacja problemu



Musimy przenieść krążki z drążka oznaczonego a na drążek b, posługując się drążkiem pomocniczym c — tak jednak postępując, aby w żadnym przypadku krążek o mniejszej średnicy nie został przykryty przez inny krążek o większej średnicy. Przyjmuje się, że krążek o numerze 1 ma najmniejszą średnicę, a ten o numerze n — największą. Ponadto dla potrzeb programu wynikowego oznaczmy krążki a, b i c jako 0, 1 i 2.

Analiza rekurencyjna zadania prowadzi nas do następujących spostrzeżeń:

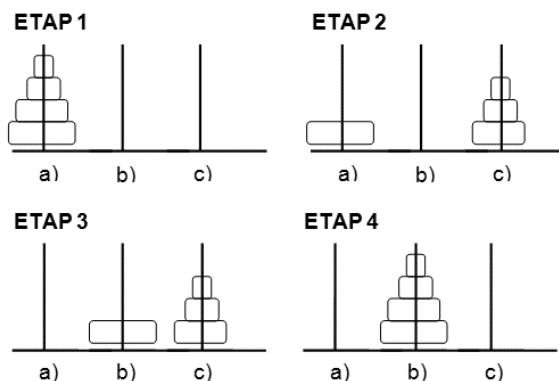
- ♦ Jeśli mamy do czynienia z jednym krążkiem, to zadanie sprowadza się do przemieszczenia go z a na b (przypadek elementarny).
- ♦ Jeśli mamy do czynienia z $n \geq 2$ krążkami, to przy założeniu, że umiemy przenieść $n-1$ krążków z jednego drążka na drugi, zadanie sprowadza się do wykonania przemieszczeń symbolicznie przedstawionych na rysunku 6.2.

Etap pierwszy przedstawia sytuację wyjściową. Załóżmy teraz, że przenieśliśmy jakimś „tajemniczym” sposobem $n-1$ krążków z drążka a na drążek c. Na drążku a pozostał nam największy krążek, ten o numerze n . W tym momencie dotarliśmy do sympatycznie prostego przypadku elementarnego i już bez żadnej dodatkowej magii możemy krążek o numerze n przenieść z drążka a na drążek b. Znajdziemy się w ten sposób w sytuacji oznaczonej na rysunku jako *etap 3*. Jak doprowadzić do rozwiązania łamigłówki, dysponując taką konfiguracją danych?

Rysunek 6.2.

Wieże Hanoi

— sposób rozwiązania



Pouczeni doświadczeniem etapu pierwszego postąpimy dokładnie w taki sam sposób: weźmiemy $n-1$ krążków z drążka c i przemieścimy je tajemniczym sposobem na drążek b.

Wzmiankowany powyżej „tajemniczy sposób” nie powinien stanowić niespodzianki dla osób, które mają za sobą lekturę rozdziału 2. Chodzi oczywiście o spowodowanie serii wywołań rekurencyjnych, które będą pamiętały o naszych intencjach i postępując wg założonych reguł, rozwiążą łamigłówkę.

Zauważmy, że przy przyjętych oznaczeniach mamy $a+b+c=0+1+2=3$, czyli $c=3-a-b$. Procedura, która rozwiązuje problem wież *Hanoi*, jest teraz niesłychanie prosta:

**hanoi.cpp**

```
void hanoi(int n, int a, int b)
{
    if (n==1)
        cout << "Przesuń dysk nr "<< n << " z " << a << " na "<< b << endl;
    else
    {
        hanoi(n-1,a,3-a-b);
        cout << "Przesuń dysk nr "<< n << " z " << a << " na "<< b << endl;
        hanoi(n-1,3-a-b,b);
    }
}
```

Niestety algorytm ten jest dość kosztowny, bowiem czas jego wykonania wynosi aż $(2^n-1) \cdot t_e$, gdzie t_e jest czasem pojedynczego przemieszczenia krążka z jednego drążka na inny⁵.

O ile jednak nie możemy specjalnie w ten czas ingerować (sam problem jest z natury dość czasochłonny), to możemy nieco ułatwić generację kodu kompilatorowi, eliminując drugie wywołanie rekurencyjne, które spełnia warunek narzucony przez *Twierdzenie 1* (patrz strona 151). Przekształcenie procedury *hanoi* wg podanej tam reguły jest natychmiastowe:

```
void hanoi2(int n, int a, int b)
{
    while (n!=1)
    {
        hanoi2(n-1,a,3-a-b);
        cout << "Przesuń dysk nr."<< n << " z " << a << " na "<< b << endl;
        n=n-1;
        a=3-a-b;
    }
}
```

⁵ Wynik ten nie jest trudny do uzyskania, ale dla czytelności wykładu zostanie pominięty.


```

    }
    cout << "Przesuń dysk nr.1 z " << a << " na " << b << endl;
}

```

Pokażna grupa procedur rekurencyjnych dość łatwo poddaje się transformacji opisanej w *Twierdzeniu 1*. Ponadto wiele procedur daje się sprowadzić — poprzez niewielkie modyfikacje kodu — do „transformowalnej” postaci. Taki właśnie przykład będziemy teraz analizowali.

Podczas omawiania rekurencji mieliśmy okazję poznać programową realizację funkcji obliczającej silnię:

```

int silnia(int x)
{
    cout << "x" << x << endl;
    if (x==0)
        return 1;
    else
        return x*silnia(x-1);
}

```

Czy uda nam się zamienić ją na wersję iteracyjną? Pierwszy problem skupia się na tym, że mamy do czynienia ze skrótem, polegającym na wprowadzeniu wywołania rekurencyjnego do równania zwracającego wynik funkcji. Nic jednak nie stoi na przeszkodzie, aby ową sporną linię rozpisać, co da nam następującą wersję (oczywiście — całkowicie równoważną):

```

int silnia(int x)
{
    if (x==0)
        return 1;
    else
    {
        int tmp=silnia(x-1);
        return x*tmp;
    }
}

```

Niestety niewiele nam to pomogło, gdyż wywołanie rekurencyjne nie jest terminalne, a zatem nie jest możliwe zastosowanie *Twierdzenia 1*. Ta przeszkoda może być jednak łatwo pokonana, jeśli dokonamy kolejnej transformacji:

```

int silnia(int x, int res1)
{
    if (x==0)
        return res;
    else
        silnia(x-1,x*res);
}

```

Nie sposób tu ukryć, że powróciliśmy do tak zachwalanego, podczas omawiania rekurencji, typu rekurencji „z parametrem dodatkowym” (taką wówczas przyjęliśmy nazwę). Czyżby zatem rekurencja „terminalna” i rekurencja „z parametrem dodatkowym” były dokładnie tymi samymi fenomenami?! Jeśli tak, to dlaczego nie wspomnieliśmy o tym wcześniej, wprowadzając na dodatek nowe nazewnictwo?

Odpowiedź zabrzmi dość przewrotnie: te dwa typy rekurencji są i nie są zarazem takie same. Wprowadzając nowy termin, ową rekurencję z parametrem dodatkowym, miałem na uwadze pewną klasę zagadnień natury numerycznej lub quasi-numerycznej. Wyrażając to jeszcze dokładniej: grupę programów, które zwracają namacalny wynik, np. liczbę, tablicę, ciąg znaków, etc. Ten wynik jest dostarczany poprzez parametr dodatkowy i stąd pochodzi nazwa. Natomiast programem terminalnym może być procedura *hanoi*, która nic „dotykającego” — oprócz przepisu na rozwiązanie łamigłówek — nie dostarcza! Poprzestając na tym wyjaśnieniu, przekształćmy wreszcie funkcję *silnia* na jej postać rekurencyjną. Niespodzianek nie powinno być żadnych — tłumaczenie jest niemal automatyczne:

```
int silnia_it(int x, int res=1)
{
    while (x!=0)
    {
        res=x*res;
        x--;
    }
    return res;
}
```

Derekursywacja z wykorzystaniem stosu

W tym paragrafie zapoznamy się z nową metodą *derekursywacji*, która jest dość kontrowersyjna. Zmuszeni bowiem będziemy do swoistego zaprzeczenia regułom programowania strukturalnego i na dodatek proponowane rozwiązania nie będą miały nic wspólnego z estetycznymi wymogami programowania. Powodem tego jest operowanie pojęciami o bardzo niskim poziomie abstrakcji, bardzo zbliżonymi do zwykłego języka assemblera. Zasada jest prosta: wiedząc, jak kompilator traktuje wywołania rekurencyjne, będziemy usiłowali robić to samo, lecz po drodze będziemy próbować nieco upraszczać jego zadanie. Mamy bowiem do dyspozycji coś, czego brakuje współczesnym kompilatorom: naszą inteligencję. Kompilator jest zwykłym programem postępującym automatycznie: plik tekstowy zawierający program w języku wysokiego poziomu jest zamieniany na maszynową reprezentację, która jest możliwa do wykonania przez procesor komputera. Rozpatruje on programy pod kątem poprawności składniowej i nie jest w stanie analizować ich sensu i celu. My natomiast całą tę wiedzę posiadamy i stąd właśnie wziął się pomysł metody derekursywacji z wykorzystaniem stosu.

Metoda ta jest podzielona na dwa etapy:

1. zamianę zmiennych lokalnych na globalne;
2. transformację programu rekurencyjnego pozbawionego zmiennych lokalnych na postać iteracyjną.

W kolejnych paragrafach szczegółowo omówimy te dwa posunięcia.

Eliminacja zmiennych lokalnych

Zanim w ogóle zaczniemy coś eliminować, warto upewnić się, czy zdajemy sobie sprawę, co będzie przedmiotem naszych zabiegów. Zmienne lokalne pełnią w języku strukturalnym rolę szczególną: umożliwiają czytelne formułowanie algorytmów i pozbawiają tak dobrze znanego programującym w dawnym BASIC-u strachu przed modyfikacją jakiegś ważnej w innym miejscu zmiennej. Mając to na uwadze, dziwną wydawać by się mogła propozycja powrotu do tych prehistorycznych czasów, w których nie było procedur, zmiennych lokalnych, przesłaniania nazw, etc. Na szczęście nikt czegoś takiego nie ma zamiaru proponować! Omawiana metoda nie jest bowiem w żadnym razie metodą programowania, lecz zwykłą techniką optymalizacyjną — a jest to istotna różnica. Wróćmy zatem do zmiennych lokalnych i zdefiniujmy sobie, co to takiego.

- ◆ *Zmienną lokalną* pewnej procedury *P* będziemy zwali taką zmienną, która może być modyfikowana tylko przez tę procedurę.
- ◆ *Zmienną globalną* z punktu widzenia procedury *P* będzie taka zmienna, która może być zmodyfikowana na zewnątrz tej procedury.

W C++ każda zmienna zadeklarowana wewnątrz bloku ograniczonego nawiasami klamrowymi { } jest uważana za lokalną dla tego bloku! Tak więc w poniższej procedurze mamy do czynienia z dwiema różnymi zmiennymi lokalnymi `var_loc` i jedną zmienną globalną `var_glob`:

```

int var_glob;
void P()
{
    int var_loc;
    ...
    while(jakiś_warunek)
    {
        int var_loc;
        ...
    }
}

```

Wiedząc już dokładnie, z czym mamy do czynienia, możemy zobaczyć, w jaki sposób przekształcić rekurencyjną procedurę zawierającą zmienne lokalne `zm_loc` i pewne parametry wywołania `param_wywoł`⁶ w analogicznie działającą procedurę, ale używającą tylko zmiennych globalnych (tym samym procedura `P` nie będzie już miała w ogóle parametrów wywołania).

Rozważmy dość ogólną formę wywołania procedury rekurencyjnej `P`:

```

void P(param_wywoł)
{
    ...
    F(param_wywoł);
    ...
}

```

Pierwszy etap transformacji polega na usunięciu funkcji `F` z wywołania `P`:

```

void P(param_wywoł)
{
    ...
    param_wywoł=F(param_wywoł);
    P(param_wywoł);
    ...
}

```

Jest to najzwyczajniejsze przepisanie kodu w nieco innej postaci. Chcemy uczynić `zm_loc` i `param_wywoł` zmiennymi globalnymi, tymczasem ulegają one podczas wywołania rekurencyjnego modyfikacji poprzez kolejny egzemplarz procedury `P`! Jak sobie z tym poradzimy? Musimy bowiem w jakiś sposób zachować wartości `zm_loc` i `param_wywoł`, aby — pomimo ewentualnych zmian ich zawartości podczas wykonania procedury `P` — sytuacja przed i po była taka sama. Pomoże nam w tym oczywiście stos:

```

void P()
{
    ...
    push(param_wywoł)
    push(zm_loc)
    param_wywoł=F(param_wywoł);
    P(param_wywoł);
    pop(zm_loc);
    pop(param_wywoł);
    ...
}

```

Dokonałiśmy zatem tego, co było naszym celem: pozbawiliśmy procedurę `P` wszelkich parametrów lokalnych, a pomimo to jej funkcjonowanie — jak również funkcjonowanie całego programu — nie uległo zmianie. Musimy jednak pamiętać o tym, by prawidłowo zainicjować globalne już zmienne `zm_loc` i `param_wywoł` właściwymi wartościami⁷, tak aby zachować pełną równoważność funkcjonalną naszego programu — przed i po przeróbce. Analizując jeszcze naszą

⁶ Zarówno `zm_loc`, jak i `param_wywoł` reprezentują listy zmiennych — to dla skrócenia zapisu.

⁷ Przed pierwszym wywołaniem procedury `P`.

metodę, warto wspomnieć o nasuwającej się od razu optymalizacji. Na stosie musimy zachowywać tylko te wartości zmiennych lokalnych, które są potrzebne. W szczególności absolutnie nie ma potrzeby chować na stos tych zmiennych lokalnych, które nie są już używane po wywołaniu rekurencyjnym.

Dla ilustracji opisanego powyżej procesu przeanalizujmy raz jeszcze nasz klasyczny przykład wież *Hanoi* (patrz strona 151). Proste przekształcenia algorytmu prowadzą do następującej wersji:

```
void hanoi3()
{
    while (n!=1)
    {
        push(n); push(a); push(b);
        n=n-1;
        b=3-a-b;
        hanoi3();
        pop(b); pop(a); pop(n);
        cout << "Przesuń dysk nr "<< n << " z " << a << " na " << b <<endl;
        n=n-1;
        a=3-a-b;
    }
    cout << "Przesuń dysk nr 1 z " << a << " na " << b <<endl;
}
```

Metoda funkcji przeciwnych

Użycie stosu — wywołań typu `push` i `pop` — jest kosztowne zarówno ze względu na czas potrzebny na obsługę tej struktury danych, jak i na pamięć niezbędną do rezerwacji dostatecznie dużego stosu. Jak dużego? Problemem jest to, że nie wiemy tego *a priori*, co zmusza nas do założenia najgorszego przypadku. Z tego też powodu wszelkie ewentualne metody pozwalające nie korzystać ze stosu powinny być przez nas powitane jak najprzychylniej. Taka metoda zostanie przedstawiona już za moment.

Dużą wadą nowej techniki będzie niemożność łatwego jej sformalizowania. Z praktycznego punktu widzenia sprawa polega na tym, iż nie jest możliwe podanie prostego przepisu, który mógłby być w miarę automatycznie⁸ zastosowany. Będziemy musieli zatrudnić naszą wyobraźnię i intuicję — a czasami nawet pogodzić się z niemożnością znalezienia rozwiązania. Przejdźmy jednak do szczegółów.

Przypomnijmy raz jeszcze ogólną postać procedury rekurencyjnej:

```
void P1(param_wywoł)
{
    ...
    param_wywoł=F(param_wywoł);
    P1(param_wywoł);
    ...
}
```

Wiemy, że wywołanie `P(param_wywoł)` modyfikuje (lub może zmodyfikować) `zm_loc` i `param_wywoł`. Poprzednio, aby się od tego uchronić, wykorzystaliśmy zachowawcze własności stosu.

Pomysł polega na tym, aby uzupełnić procedurę `P1` o pewne instrukcje, które „wiedząc”, jak wywołanie `P1(param_wywoł)` modyfikuje `zm_loc` i `param_wywoł`, wykonałyby czynność odwrotną, tak aby przywrócić wartości tych funkcji sprzed wywołania! Inaczej mówiąc, chodzi nam o doprowadzenie programu do postaci:

⁸ Co nie znaczy, że bezmyślnie!

```

void P2()
{
    ...
    param_wywoł=F(param_wywoł);
    P2;
    FUNKCJA_ODWROTNA(zm_loc,param_wywoł);
    ...
}

```

(1)

(2)

Działanie owej tajemniczej *funkcji odwrotnej* musi być takie, aby wartości `zm_loc` i `param_wywoł` były dokładnie takie same w punktach programu oznaczonych (1) i (2). Jak to zrobić? Ba! Oto jest dopiero pytanie! Odpowiedź będzie inna w przypadku każdego programu i nie pozostaje nam nic innego, jak tylko pokazać jakiś konkretny przykład.

Poniższa procedura `P1` liczy elementy wymyślnego ad hoc ciągu matematycznego:



odwrotna.cpp

```

void P1(int a,int& b)
{
    if(a==0)
        b=1;
    else
    {
        P1(a-1,b);
        // tu funkcja odwrotna?
        b=b+a;
    }
}

```

Sens matematyczny tej procedury jest dla nas nieistotny. Jedyne, co nas w tym momencie interesuje, to takie jej przekształcenie, aby uzyskać procedurę `void P2`, która — korzystając tylko ze zmiennych globalnych `a` i `b` — będzie działała w sposób identyczny.

Pierwszym etapem naszej analizy jest odpowiedź na pytanie: „Które zmienne są modyfikowane przez rekurencyjne wywołanie `P1`?”. Zmienna `b` ma charakter globalny, gdyż nie jest przez `P1` modyfikowana. Służy ona wyłącznie do przekazywania wyniku z wywoływanej procedury. Tak więc *funkcja odwrotna* — jaka nie byłaby jej postać — nie będzie się musiała zajmować zachowaniem wartości `b`. Jedyną zmienną, która jest modyfikowana, jest `a`. Dekrementowana wartość zmiennej `a` jest przekazywana procedurze `P1`, natomiast po ukończeniu pracy tejże chcemy korzystać z niezmienionej wartości `a`.

W poznanej poprzednio metodzie *eliminacji zmiennych lokalnych* należałoby po prostu zachować oryginalną wartość `a` na stosie. W naszym przypadku wystarczy (wiedząc, że jedyną modyfikacją, jakiej może na zmiennej `a` dokonać procedura `P1`, jest dekrementacja) po prostu przywrócić oryginalną wartość `a`, inkrementując ją! I to jest tą naszą tajemniczą „funkcją odwrotną”!

Popatrzmy na zmodyfikowaną treść procedury:

```

int a,b; //zmienne globalne!
void P2()
{
    if(a==0)
        b=1;
    else
    {
        a=a-1;
        P2();
        a=a+1;
        b=b+a;
    }
}

```

Sprawdźmy teraz w programie głównym, czy istotnie nasz program działa prawidłowo⁹:

```
int main()
{
    for (int i=0; i<17;i++)
    {
        P1(i,b);
        cout << b << " ";
    }
    cout << endl;
    for (int i=0; i<17;i++)
    {
        a=i;
        P2():
        cout << b << " ";
    }
    cout << endl;
}
```

Oto co ukaże się na ekranie:

```
2 4 7 11 16 22 29 37 46 56 67 79 92 106 121 137
2 4 7 11 16 22 29 37 46 56 67 79 92 106 121 137
```

Wszelkie znaki na ekranie i papierze wskazują, iż procedury P1 i P2 są równoważne.

Klasyczne schematy derekursywacji

Poznane wcześniej metody eliminacji zmiennych lokalnych z procedur, jak również pozbawienie ich parametrów służyły jednemu istotnemu celowi: jak największemu zbliżeniu sposobu wykonywania procedur rekurencyjnych do typowego programu iteracyjnego. W istocie czym jest program określany jako „iteracyjny”? Termin ten dotyczy zasadniczo systematycznego powtarzania pewnych fragmentów kodu, np. za pomocą instrukcji `for`, `while`, `do-while`. Wywołanie rekurencyjne ma wiele wspólnego z iteracyjnym sposobem wykonywania programów pod względem ideowym (chodzi o systematyczne powtarzanie pewnych czynności), bardzo niewiele jednak ma z nim wspólnego praktycznie. Iteracje są zwykłymi instrukcjami `goto`¹⁰ przeplatanymi badaniem warunków. Wywołania rekurencyjne natomiast znajdują się co najmniej o poziom¹¹ wyżej. Poprzez usunięcie zmiennych lokalnych i parametrów funkcji przybliżyliśmy je bardzo do schematu iteracyjnego.

Procedury rekurencyjne posiadają obowiązkowo pewne testy służące do sprowadzania procesu wywołań rekurencyjnych do tzw. przypadków elementarnych¹².

Przykładowo, obliczając rekurencyjnie silnię z n , ciągle badamy, czy n jest równe zero. Jeśli odpowiedź brzmi: tak, procedura zwraca wartość 1 — zaś w przeciwnym wypadku następuje kolejne wywołanie rekurencyjne. Są to dwie różne rzeczy — dwa różne fragmenty kodu wykonywane w zależności od spełnienia lub nie pewnych warunków. Iteracje natomiast, generalnie rzecz ujmując, wykonują systematycznie pewne stałe fragmenty kodu i to je odróżnia od procedur rekurencyjnych.

Narzucającym się natychmiast rozwiązaniem jest włożenie do części wykonawczej instrukcji iteracyjnej instrukcji warunkowych sprawiających, iż kod wykonywany w iteracji numer i będzie — być może — odmienny od kodu iteracji $i+1$.

⁹ Nie zastąpi to oczywiście formalnego dowodu równoważności P1 i P2, ale zadanie jest na tyle proste, iż dowód ten wydaje się w tym miejscu zbędny.

¹⁰ W rozmaitych wariacjach zależnych od zestawu instrukcji procesora.

¹¹ Abstrakcji, skomplikowania itd.

¹² Jest to wymuszone naturalną chęcią zakończenia kiedyś szeregu wywołań rekurencyjnych!

Jest to droga, którą pójdziemy w celu odnalezienia sposobu derekursywacji pewnych schematów, często spotykanych podczas programowania z wykorzystaniem technik rekurencyjnych.



Wszystkie rozpatrywane dalej schematy dotyczą procedur już bezparametrowych i pozbawionych zmiennych lokalnych.

Schemat typu while

Kolejnym schematem, z którym będziemy mieli do czynienia, jest:

```
void P()
{
  while(warunek(x))
  {
    A(x);
    P();
    B(x);
  }
  C(x);
}
```

W celu wynalezienia równoważnej formy iteracyjnej zapiszmy procedurę P w nieco innej postaci z użyciem instrukcji `goto`. Posunięcie to doprowadzi do wyeliminowania instrukcji `while` (w dość sztuczny sposób, to trzeba przyznać). Wprowadźmy ponadto kolejną globalną zmienną N — używaną już zresztą wcześniej:

```
void P()
{
  N=0;
  start:
  if(warunek(x))
  {
    A(x);
    N++; P; N--;
    B(x);
    goto start;
  }
  else
    C(x);
}
```

Jest to forma niewątpliwie równoważna, choć pozornie niewiele z niej na razie wynika. Przeanalizujmy jednak dokładniej działanie tego programu, starając się odtworzyć sekwencyjny sposób wywoływania grup instrukcji oznaczonych symbolicznie jako $A(x)$, $B(x)$ i $C(x)$.

Widać od razu, iż każdorazowe spełnienie warunku instrukcji `if... else` spowoduje na pewno wykonanie $A(x)$ i $N++$. Niespełnienie zaś warunku spowoduje jednokrotne wykonanie $C(x)$. Tyle możemy zaobserwować odnośnie do kodu obsługiwanego przed wywołaniem rekurencyjnym P .

Co się jednak dzieje podczas wywołań i powrotów rekurencyjnych? Otóż wykonywana jest instrukcja $B(x)$, oczywiście wraz z $N--$. Jeśli teraz zdecydujemy się na zasymulowanie operacji wywoływania i powrotu rekurencyjnego poprzez — odpowiednio — $N++$ i $N--$, możliwe jest zaproponowanie następującej równoważnej formy procedury P :

```
int N=0;
void P()
{
  do
  {
    while(warunek(x))
    {
```

```

    A(x);
    N++;
  }
  C(x);
  if(N==0)
    goto koniec;
  N--;
  B(x);
} while(N!=0);
koniec:
}

```

Czytelnik, którego nie przekonał ten wywód, może znaleźć bardziej ścisły matematycznie dowód prawidłowości powyższej transformacji w [Kro89]. Na użytek tego podręcznika zdecydowałem się jednak na zamieszczenie mniej formalnego wyjaśnienia — tym bardziej że zagłębianie się w dywagacje na temat derekursywacji ma bardzo mało wspólnego z algorytmiką, a bardzo wiele ze sztuczkami łatwo prowadzącymi do przyjęcia złego stylu programowania.

Schemat typu if-else

Weźmy pod uwagę schemat rekurencyjny przedstawiony na listingu poniżej:

```

void P()
{
  if(warunek(x))
  {
    A(x);
    P();
    B(x);
  }
  else
    C(X);
}

```

Zakładając N -krotne wywołanie procedury P , jej działanie można poglądowo przedstawić jako sekwencję instrukcji:

$$\overbrace{A(x); \dots A(x)}^N; C(x); \overbrace{B(x); \dots B(x)}^N;$$

Jest to taka forma zapisu algorytmu, która od razu sugeruje możliwe zapisanie w formie iteracyjnej... pod warunkiem wszakże znajomości N . Niestety liczba wywołań procedury P nie jest nigdy znana a priori — wszystko zależy od globalnego parametru, z którym zostanie ona wywołana!

Nie popadajmy jednak w przedwczesną depresję i spójrzmy na następującą wersję procedury P :

```

int N=0;
void P()
{
  if warunek(x)
  {
    A(x);
    N++;P();N--;
    B(x);
  }
  else
    C(X);
}

```

Załóżmy, że wykonanie tego programu zostało przerwane w pewnym losowo wybranym momencie i za pomocą debuggera odczytaliśmy wartość N . Biorąc pod uwagę, iż — jak to wynika z treści programu — zmienna globalna N jest inkrementowana podczas każdego wywołania

rekurencyjnego P i dekrementowana po powrocie z niego, możemy wykorzystać tę zmienną do odczytywania aktualnego poziomu rekurencji procedury P ¹³.

Idea kolejnej transformacji procedury P jest teraz następująca: wywołanie rekurencyjne procedury P będziemy symulować za pomocą skoku do jej początku. Podczas kolejnego wykonania procedury P możemy w bardzo łatwy sposób przetestować, czy wszystkie zaległe jej wywołania zostały już ukończone — powie nam o tym wartość N , do której zawsze mamy dostęp wewnątrz P ¹⁴.

Powyższe uwagi prowadzą natychmiast do kolejnej wersji programu:

```
int N=0;

void P()
{
  start:
  if warunek(x)
  {
    A(x);
    N++;
    goto start;
  powrot:
    N--;
    B(x);
  }
  else
    C(X);
  if (N!=0)
    goto powrot;
}
```

Zapis z użyciem instrukcji `goto` jest dopuszczalny tylko wtedy, gdy przemawiają za tym szczególne względy. Nasz prosty przykład ich nie dostarcza; program ten bowiem może być z łatwością zamieniony na postać strukturalną.

Poniżej zamieszczone są obie wersje procedury P : oryginalna i tak długo poszukiwana jej iteracyjna wersja:

<pre>void P() { if(warunek(x)) { A(x); P(); B(x); } else C(x); }</pre>	<pre>int N=0; void P() { while(warunek(x)) { A(x); N++; } C(X); while(N--!=0) B(x); }</pre>
--	---

Sprawdźmy teraz, czy w istocie podane wyżej przekształcenie działa. W tym celu powróćmy do programu przykładowego ze strony 157 (zapisanego teraz w nieco zwięźlejszej postaci).



odwrot2.cpp

```
void P2()
{
  if(a!=0)
  {
    a--;
    P2();
  }
```

¹³ Patrz rozdział 2.

¹⁴ Jeśli N wynosi 0, to wszystkie zaległe wywołania zostały już ukończone.

```

        b=b+(++a);
    }
    else
        b=1;
}

```

Stosując omawiane przekształcenie, otrzymujemy natychmiast:

```

void P2_ITERAT()
{
    int k=0;
    while (a!=0)
    {
        a--;
        k++;
    }
    b=1;
    while (k--!=0)
        b=b+(++a);
}

```

Wykonanie programu potwierdza równoważność obu procedur.

Schemat z podwójnym wywołaniem rekurencyjnym

Ostatni omawiany schemat rekurencyjny należy do rzadko spotykanych w praktyce. Ponadto dowód na poprawność transformacji jest dość złożony, dlatego poniżej przeanalizujemy jedynie gotowy rezultat i omówimy przykład zastosowania transformacji.

Oto dwie równoważne formy algorytmów:

```

void P()
{
    if(warunek(x))
    {
        A(x);
        P();
        B(x);
        P();
        C(x);
    }
    else
        D(x);
}

```

```

int N=1;
void P()
{
    do
    {
        while(warunek(x))
        {
            A(x);
            N*=2;
        }
        D(x);
        while((N!=1)&&(N%2))
        {
            N=N/2;
            C(x);
        }
    }
    if(N==1)
        goto koniec;
    N=N+1;
    B(x);
}while(p!=1);
}

```

Dla zilustrowania metody rozważmy jeszcze raz problem wież *Hanoi*, przedstawiony na stronie 151. Mając do dyspozycji metodę *funkcji* przeciwnych (strona 156), łatwo dojdziemy do następującej wersji zaproponowanej tam procedury:



hanoi_it.cpp

```

int a,b,n;
void hanoi()
{

```

```

if (n!=1)
{
    n--; b=3-a-b;
    hanoi();
    n++; b=3-a-b;
    cout << "Przesuń dysk nr."<< n << " z " << a << " na "<< b <<endl;
    n--; a=3-a-b;
    hanoi();
    n++; a=3-a-b;
}
else
    cout << "Przesuń dysk nr."<< n << " z " << a << " na "<< b <<endl;
}

```

Zauważmy, że instrukcje `n++` i `n--` znoszą się wzajemnie, można je więc usunąć.

Jeśli poddamy procedurę `hanoi` przeróbce na wersję iteracyjną, powinniśmy otrzymać:

```

void hanoi_iter()
{
    int M=1;
    do
    {
        while (n!=1)
        {
            n--;
            b=3-a-b;
            M*=2;
        }
        cout << "Przesuń dysk nr."<< n << " z " << a << " na "<< b <<endl;
        while ((M!=1) && (M%2))
        {
            M=M/2;
            n=n+1;
            a=3-a-b;
        }
        if(M==1)
            goto KONIEC;
        M++;
        n++;
        b=3-a-b;
        cout << "Przesuń dysk nr."<< n << " z " << a << " na "<< b <<endl;
        n--;
        a=3-a-b;
    } while (M!=1);
    KONIEC:
    ;
}

```

Uruchomienie programu przekonuje, iż obie procedury wykonują dokładnie to samo zadanie i dają identyczne rezultaty.

Podsumowanie

Omówione w tym rozdziale techniki derekursywacji algorytmów nie wyczerpują zestawu dostępnych metod służących do tego celu, jednak prezentują wachlarz dostatecznie szeroki, aby móc obsłużyć większość najczęściej spotykanych procedur rekurencyjnych. Prezentowane metody mogą ponadto posłużyć jako wzorzec przy rozwiązywaniu zadań podobnych, ale nie całkowicie zgodnych z omówionymi schematami.

Rozdział 7.

Algorytmy przeszukiwania

Pojęcie „przeszukiwania” pojawiało się w tej książce już kilka razy w charakterze przykładów i zadań. Niemniej jest ono na tyle ważne, iż wymaga ujęcia w klamry osobnego rozdziału. Aby unikać powtórzeń, tematy już omówione będą zawierały raczej odnośniki do innych części książki niż pełne omówienia. Szczegółowej dyskusji zostanie poddana metoda *transformacji kluczowej*. Z uwagi na znaczenie i pewną odrębność tematu przeszukiwanie tekstów zostało zgrupowane w rozdziale kolejnym.

Przeszukiwanie liniowe

Temat przeszukiwania liniowego pojawił się już jako ilustracja pojęcia rekurencji. Iteracyjna wersja zaproponowanego tam programu jest oczywista — do jej wymyślenia nie jest nawet potrzebna znajomość treści rozdziału 6. Poniżej przedstawiony jest przykład przeszukiwania tablicy liczb całkowitych. Oczywiście metoda ta działa również w nieco bardziej złożonych przypadkach — modyfikacji wymaga jedynie funkcja porównująca x z aktualnie analizowanym elementem. Jeśli elementami tablicy są rekordy o dość skomplikowanej strukturze, to warto użyć jednej funkcji `szukaj`, która otrzymuje jako parametr wskaźnik do funkcji porównawczej¹.



linear.cpp

```
int szukaj(int tab[n], int x)
{
    int i;
    for( i=0; (i<n) && (tab[i]!=x ); i++);
    return i;
}
```

Odnalezienie liczby x w tablicy `tab` jest sygnalizowane poprzez wartość funkcji; jeśli jest to liczba z przedziału $0\dots, n-1$, wówczas jest po prostu indeksem komórki, w której znajduje się x . W przypadku zwrotu liczby n jesteśmy informowani, iż element x nie został znaleziony. Zasada obliczania wyrażeń logicznych w C++ gwarantuje nam, że podczas analizy wyrażenia $(i < n) \ \&\& \ (tab[i] != x)$ w momencie stwierdzenia fałszu pierwszego czynnika iloczynu logicznego reszta wyrażenia — jako niemająca znaczenia — nie będzie już sprawdzana. W konsekwencji nie będzie badana wartość spoza zakresu dozwolonych indeksów tablicy, co jest tym cenniejsze, iż kompilator C++ w żaden sposób o tego typu przeoczeniu by nas nie poinformował.

¹ Użycie wskaźników do funkcji zostało omówione na przykładzie w rozdziale 5.

Typ przeszukiwania, polegający na zwykłym sprawdzaniu elementu po elemencie, jest metodą bardzo wolno działającą, mamy tu do czynienia z klasą $O(n)$. Przeszukiwanie liniowe może być stosowane wówczas, gdy nie posiadamy żadnej informacji na temat struktury przeszukiwanych danych, ewentualnie sposobu ich składowania w pamięci.

Przeszukiwanie binarne

Jak już zostało zauważone w paragrafie poprzednim, ewentualna informacja na temat sposobu składowania danych może być niesłychanie użyteczna podczas przeszukiwania. W istocie często mamy do czynienia z uporządkowanymi już w pamięci komputera danymi: np. rekordami posortowanymi alfabetycznie, według niemalejących wartości pewnego pola rekordu, etc. Zakładamy zatem, że tablica jest posortowana, ale jest to dość częsty przypadek w praktyce, bowiem człowiek lubi mieć do czynienia z informacją uporządkowaną. W takim przypadku można skorzystać z naszej „meta-wiedzy” w celu usprawnienia przeszukiwania danych. Możemy bowiem łatwo wyeliminować z przeszukiwania te obszary tablicy, gdzie element x na pewno nie może się znaleźć. Dokładnie omówiony przykład *przeszukiwania binarnego* znalazł się już w rozdziale 2. — patrz zadanie 2. i jego rozwiązanie. W tym miejscu możemy dla odmiany podać iteracyjną wersję algorytmu:



binary-i.cpp

```
int szukaj(int tab[], int x)
{ //zwraca indeks poszukiwanej wartości lub -1
enum {TAK,NIE} Znalazlem=NIE;
int left=0, right=n-1, mid;
while( (left<=right) && (Znalazlem!=TAK) )
{
mid=(left+right)/2;
if(tab[mid]==x)
    Znalazlem=TAK;
else
    if(tab[mid]<x)
        left=mid+1;
    else
        right=mid-1;
}
if(Znalazlem==TAK)
    return mid;
else
    return -1;
}
```

Nazwy i znaczenie zmiennych są dokładnie takie same, jak we wspomnianym zadaniu, dlatego warto tam zerknąć choć raz dla porównania. Pewnej dyskusji wymaga problem wyboru elementu środkowego (mid). W naszych przykładach jest to dosłownie środek aktualnie rozpatrywanego obszaru poszukiwań. W rzeczywistości jednak może nim być oczywiście dowolny indeks pomiędzy $left$ i $right$! Nietrudno jednak zauważyć, że dzielenie tablicy na pół zapewnia nam eliminację największego możliwego obszaru poszukiwań. Ich niepowodzenie jest sygnalizowane przez zwrot wartości -1 . W przypadku sukcesu zwracany jest tradycyjnie indeks elementu w tablicy.

Przeszukiwanie binarne jest algorytmem klasy $O(\log_2 N)$ (patrz podpunkt „Rozkład logarytmiczny”, rozdział 3.). Dla dokładnego uświadomienia sobie jego zalet weźmy pod uwagę konkretny przykład numeryczny:

Przeszukiwanie liniowe pozwala w czasie proporcjonalnym do rozmiaru tablicy (listy) odpowiedzieć na pytanie, czy element x się w niej znajduje. Zatem dla tablicy o rozmiarze 20 000 należałoby w najgorszym przypadku wykonać 20 000 porównań, aby odpowiedzieć na postawione pytanie. Analogiczny wynik dla przeszukiwania binarnego wynosi $\log_2 20\,000$ (ok. 14 porównań).

Nic tak dobrze nie przemawia do wyobraźni, jak dobrze dobrany przykład liczbowy, a powyższy na pewno do takich należy.

Transformacja kluczowa (hashing)

Zanim powiemy choćby słowo na temat transformacji kluczowej², musimy sprecyzować dokładnie dziedzinę zastosowań tej metody. Otóż jest ona używana, gdy maksymalna liczba elementów należących do pewnej dziedziny³ \mathfrak{R} jest z góry znana (E_{\max}), natomiast wszystkich możliwych (różnych) elementów tej dziedziny mogłoby być potencjalnie bardzo dużo (C). Tak dużo, że o ile przydział pamięci na tablicę o rozmiarze E_{\max} jest w praktyce możliwy, o tyle przydział tablicy dla wszystkich potencjalnych C elementów dziedziny \mathfrak{R} byłby fizycznie niewykonalny.

Dlaczego tak ważna jest uwaga o fizycznej niemożności przydziału takiej tablicy? Otóż gdyby taki przydział był realny, stworzylibyśmy bardzo dużą tablicę, w której elementy odnajdywane byłyby poprzez zwykłe adresowanie bezpośrednie i wszystkie funkcje szukania działałyby w czasie $O(1)$. Jak pamiętamy z rozdziału 3., notacja $O(1)$ umownie oznacza, że liczba operacji wykonywanych przez algorytm jest niezależna od rozmiarów problemu. Teoretycznie wielkość zbioru do przeszukania nie będzie miała znaczenia, jeśli będziemy dysponowali pewną funkcją H , która pozwoli nam dotrzeć do poszukiwanego rekordu w mniej więcej tym samym, skończonym czasie.

Oto przykład ilustrujący sytuację:

- ♦ Chcemy zapamiętać $R_{\max} = 250$ słów o rozmiarze 10 (tablica o rozmiarze $250 \cdot 10 = 2\,500$ bajtów jest w pełni akceptowalna⁴).
- ♦ Wszystkich możliwych słów jest $C = 26^{10}$ (nie licząc w ogóle polskich znaków!). Praktycznie niemożliwe jest przydzielenie pamięci na tablicę, która mogłaby je wszystkie pomieścić.

Idea transformacji kluczowej polega na próbie odnalezienia takiej funkcji H^5 , która — otrzymując w parametrze pewien zbiór danych — podałaby nam indeks w tablicy T , gdzie owe dane znajdowałyby się... gdyby je tam wcześniej zapamiętano!

Inaczej rzecz ujmując: transformacja kluczowa polega na odwzorowaniu:

$$\text{dane} \mapsto \text{adres komórki w pamięci}.$$

Zakładając taką organizację danych, położenie nowego rekordu w pamięci *teoretycznie* nie powinno zależeć od położenia rekordów już wcześniej zapamiętanych. Jak zapewne pamiętamy z rozdziału 5., nie był to przypadek list posortowanych, drzew binarnych, sterty itp. Naturalną konsekwencją nowego sposobu zapamiętywania danych jest maksymalne uproszczenie procesu po-

² Inne spotykane nazwy: mieszanie, rozpraszanie.

³ „Dziedziny” w sensie matematycznym.

⁴ Jeden dodatkowy bajt na znak ‘\0’ kończący ciąg tekstowy w C++.

⁵ Inna spotykana nazwa to „funkcja mieszająca”.

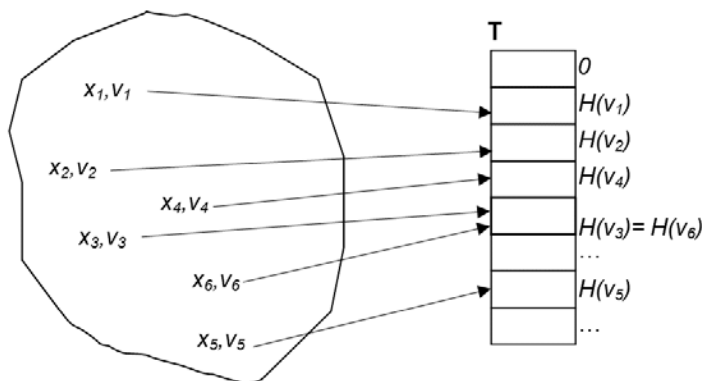
szukiwania. Poszukując bowiem elementu x charakteryzowanego przez pewien klucz⁶ v , możemy się posłużyć pewną znaną nam funkcją H . Obliczenie $H(v)$ powinno zwrócić nam pewien adres pamięci, pod którym należy sprawdzić istnienie x , i do tego w sumie sprowadza się cały proces poszukiwania!

Idea transformacji kluczowej jest przedstawiona na rysunku 7.1.

Rysunek 7.1.

*Idea transformacji
kluczowej*

C – dziedzina wszystkich kluczy



Załóżmy, że z całej dziedziny kluczy C interesuje nas wyłącznie sześć wartości kluczy: v_1, v_2, \dots, v_6 (odpowiadają one elementom x_1, x_2, \dots, x_6) i dysponujemy funkcją odwzorowującą wartość elementu na indeks tablicy T .

Element o kluczu v zostaje zapisany w tablicy T pod indeksem, który wyliczamy, aplikując funkcję $H(v)$. Wartość NULL w tej tablicy oznacza brak elementu.

Ponieważ odwzorowujemy bardzo dużą dziedzinę C na mały zbiór elementów (wielkość tablicy T jest ograniczona), to w efekcie nieuniknione są kolizje odwzorowania wartości $H(v)$ na indeks tablicy T . Na rysunku widać to na przykładzie elementów x_3 i x_6 , o kluczach v_3 i v_6 , dla których funkcja H wyliczyła taki sam indeks w tablicy! (Na szczęście nie jest to wada eliminująca metodę, co postaram się udowodnić w dalszej części rozdziału).

W metodzie transformacji kluczowej mamy do czynienia z zupełnym porzuceniem jakiegokolwiek procesu przeszukiwania zbioru danych: znając funkcję H , automatycznie możemy określić położenie dowolnego elementu w pamięci — wiemy również od razu, gdzie prowadzić ewentualne poszukiwania.

Czytelnik ma prawo w tym miejscu zadać sobie pytanie: to po co w takim razie męczyć się z listami, drzewami czy też innymi strukturami danych, jeśli można używać transformacji kluczowej? Jest to bardzo dobre pytanie, niestety odpowiedź na nie jest możliwa w jednym zdaniu. Wstępnie możemy tu podać dwa istotne powody ograniczające użycie tej metody:

- ♦ ograniczenia pamięci (trzeba z góry zarezerwować tablicę T na E_{max} elementów);
- ♦ trudności w odnalezieniu dobrej funkcji H .

O ile pierwszy powód jest oczywisty i może mniej istotny, to odnalezienie dobrej funkcji nie jest trywialne. A czym charakteryzuje się dobra funkcja H , dowiemy się w kolejnym podrozdziale.

⁶ Pojęcie „klucza” pochodzi z teorii baz danych i jest dość powszechnie używane w informatyce; kluczem określa się zbiór atrybutów, które jednoznacznie identyfikują rekord (nie ma dwóch *różnych* rekordów posiadających *taką samą* wartość atrybutów kluczowych).

W poszukiwaniu funkcji H

Funkcja H na podstawie klucza v dostarcza indeksu w tablicy T , służącej do przechowywania danych. Potencjalnych funkcji, które na podstawie wartości danego klucza v zwrócą pewien adres adr , jest — jak się zapewne domyślamy — mnóstwo. Parametry, które mają główny wpływ na stopień skomplikowania funkcji H , to: długość tablicy, w której zamierzamy składać rekordy danych, oraz — bez wątpienia — wartość klucza v . Przed zamierzonym przystąpieniem do klawiatury, aby wklepać naprędce wymyśloną funkcję H , warto się dobrze zastanowić, które atrybuty rekordu danych zostaną wybrane do reprezentowania klucza. Logicznym wymogiem jest posiadanie przez tę funkcję następujących własności:

- ♦ Powinna być łatwo obliczalna, tak aby ciężaru przeszukiwania zbioru danych nie przenosić na czasochłonne wyliczanie $H(v)$.
- ♦ Różnym wartościom klucza v powinny odpowiadać odmienne indeksy w tablicy T , tak aby nie powodować kolizji dostępu (np. elementy powinny być rozkładane w tablicy T równomiernie).
- ♦ Zastosowanie funkcji H powinno w efekcie spowodować równomierne i losowe rozmieszczenie elementów w tablicy T .

Pierwszy punkt nie wymaga komentarza, do kolejnych jeszcze powrócimy, gdyż stanowią one istotę metody transformacji kluczowej. W następnym paragrafie poznamy typowe metody konstruowania funkcji H . W rzeczywistych aplikacjach stosuje się przeróżne kombinacje cytowanych tam funkcji i w zasadzie nie można tu podać reguł postępowania. Często także wymagane jest eksperymentowanie i przeprowadzanie symulacji mających na celu odnalezienie dla zbioru danych o podanej charakterystyce funkcji H spełniającej podane wyżej warunki.

Najbardziej znane funkcje H

Najwyższa już pora zaprezentować kilka typowych funkcji matematycznych używanych do konstruowania funkcji stosowanych w transformacji kluczowej. Są to metody w miarę proste, jednak samodzielnie niewystarczające — w praktyce stosuje się raczej ich kombinacje niż każdą z nich osobno. Czytelnik, który z pojęciem transformacji kluczowej spotyka się po raz pierwszy, ma prawo być nieco zbulwersowany poniższymi propozycjami (modulo, mnożenie, etc.). Brakuje tu bowiem pewnej naukowej metody: nic nie jest do końca zdeterminowane, programista może w zasadzie wybrać, co mu się żywnie podoba, a algorytmy poszukiwania lub wstawiania danych będą i tak działały. W dalszych przykładach będziemy zakładać, że klucze są ciągami znaków, które można łączyć ze sobą i dość dowolnie interpretować jako liczby całkowite. Każdy znak alfabetu będziemy dla uproszczenia obliczeń w naszych przykładach kodować za pomocą 5 bitów (patrz tabela 7.1) — wybór kodu nie jest niczym zdeterminowany.

Tabela 7.1. Przykład kodowania liter za pomocą 5 bitów

A = 00001	B = 00010	C = 00011	D = 00100	E = 00101	F = 00110	G = 00111
H = 01000	I = 01001	J = 01010	K = 01011	L = 01100	M = 01101	N = 01111
O = 01110	P = 10000	Q = 10001	R = 10010	S = 10011	T = 10100	U = 10101
V = 10110	W = 10111	X = 11000	Y = 11001	Z = 11010		

Wspomniany wyżej brak metody jest na szczęście pozorny. Wiele podręczników algorytmiki błędnie prezentuje transformację kluczową, koncentrując się na tym JAK, a nie omawiając szczegółowo, PO CO chcemy w ogóle wykonywać operacje arytmetyczne na zakodowanych kluczach. Tymczasem sprawa jest względnie prosta:

- ♦ *Kodowanie* jest wykonywane w celu zamiany wartości klucza (niekoniecznie numerycznej!) na liczbę; sam kod jest nieistotny, ważne jest tylko, aby jako wynik otrzymać pewną liczbę, którą można później stosować w obliczeniach.

- ◆ Naszym celem jest możliwie jak najbardziej losowe rozmieszczenie rekordów w tablicy wielkości M : funkcja H ma nam dostarczyć w zależności od argumentu v adresy od 0 do $M-1$. Cały problem polega na tym, że nie jest możliwe uzyskanie losowego rozrzutu elementów, dysponując danymi wejściowymi, które z założenia nie są losowe. Musimy zatem uczynić coś, aby ową „losowość” w jakiś sposób *dobrze* zasymulować.

Badanie praktyczne dokonywane na dużych zestawach danych wejściowych wykazało, że istnieje grupa prostych funkcji arytmetycznych (modulo, mnożenie, dzielenie), które dość dobrze się do tego celu nadają⁷. Omówimy je kolejno w kilku paragrafach.

Suma modulo 2

Formuła: $H(v_1v_2...v_n) = v_1 \oplus v_2 \oplus ... \oplus v_n$

Przykład:

Dla

$$R_{\max} = 37, H(„KOT”) = (010110111010100)_2$$

daje

$$(01011)_2 \oplus (01110)_2 \oplus (10000)_2 = (10001) = (17)_{10}$$

Zalety:

- ◆ Funkcja H łatwa do obliczenia; suma modulo 2, w przeciwieństwie do iloczynu i sumy logicznej, nie powiększa (jak to czyni suma logiczna) ani nie pomniejsza (jak iloczyn) swoich argumentów.
- ◆ Używanie operatorów $\&$ i $|$ powoduje akumulację danych odpowiednio na *początku* i na *końcu* tablicy T , czyli jej potencjalna pojemność nie jest efektywnie wykorzystywana.

Wady:

- ◆ Permutacje tych samych liter dają w efekcie identyczny wynik — można jednak temu zaradzić poprzez systematyczne przesuwanie cykliczne reprezentacji bitowej: pierwszy znak o jeden bit w prawo, drugi znak o dwa bity w prawo, etc.

Przykład:

- ◆ bez przesuwania $H(„KTO”) = (01011)_2 \oplus (10100)_2 \oplus (01110)_2 = (17)_{10}$, jednocześnie $H(„TOK”) = (17)_{10}$;
- ◆ z przesuwaniem $H(„KTO”) = (10101)_2 \oplus (00101)_2 \oplus (11001)_2 = (9)_{10}$, natomiast $H(„TOK”) = (01010)_2 \oplus (10011)_2 \oplus (01101)_2 = (20)_{10}$.

Suma modulo R_{\max}

Formuła: $H(v) = v \% R_{\max}$

Przykład:

Dla

$$R_{\max} = 37: H(„KOT”) = (010110111010100)_2 \% (37)_{10} = (11732)_{10} = 3.$$

Zalety:

- ◆ funkcja H łatwa do obliczenia.

⁷ Jeśli zapomniałeś operacji arytmetycznych na wartościach logicznych, to polecam lekturę dodatku B.

Wady:

- ♦ Otrzymana wartość zależy — dość paradoksalnie — bardziej od R_{max} niż od klucza!
Przykładowo: gdy R_{max} jest parzyste, *na pewno* wszystkie otrzymane indeksy danych o kluczach parzystych będą również parzyste, ponadto dla pewnych dzielników wiele danych otrzyma ten sam indeks. Można temu częściowo zaradzić poprzez wybór R_{max} jako liczby pierwszej, ale tu znowu będziemy mieli do czynienia z akumulacją elementów w pewnym obszarze tablicy
— a wcześniej wyraźnie zażyczyliśmy sobie, aby funkcja H rozdzielała indeksy sprawiedliwie po całej tablicy!
- ♦ W przypadku dużych liczb binarnych niemieszczących się w reprezentacji wewnętrznej komputera obliczenie modulo już nie jest możliwe za pomocą zwykłego dzielenia arytmetycznego.

Co się tyczy ostatniej wady, to prostym rozwiązaniem dla ciągów tekstowych w C++ (wewnętrznie są to przecież zwykłe ciągi bajtów!) jest następująca funkcja, bazująca na interpretacji tekstu jako szeregu cyfr 8-bitowych:

```
int H(char *s, int Rmax)
{
    for(int tmp=0; *s != '\0'; s++)
        tmp = (64*tmp+(*s)) % Rmax;
    return tmp;
}
```

Mnożenie

Formuła: $H(v) = [((v \cdot \theta) \% 1) \cdot E_{max}]$, gdzie $0 < \theta < 1$

Powyższą formułę należy odczytywać następująco: klucz v jest mnożony przez pewną liczbę θ z przedziału otwartego $(0, 1)$. Z wyniku bierzemy część ułamkową, mnożymy przez E_{max} i ze wszystkiego liczymy część całkowitą.

Istnieją dwie wartości parametru θ , które rozrzucają klucze w miarę równomiernie po tablicy:

$$\theta_1 = \frac{\sqrt{5}-1}{2} = 0,6180339887 \text{ oraz } \theta_2 = 1 - \theta_1 = 0,3819660113.$$

Powyższa informacja jest prezentem od matematyków, a ponieważ *darowanemu koniowi nie patrzy się w zęby*, to nie będziemy zbytnio wnikać w kwestię, JAK oni to wynaleźli!

Przykład:

Dla

$$\theta = 0,6180339887, E_{max} = 30 \text{ i klucza } v = „KOT” = 11\,732 \text{ otrzymamy}^8 H(„KOT”) = 23.$$

Obsługa konfliktów dostępu

Kilka prostych eksperymentów przeprowadzonych z funkcjami zaprezentowanymi w poprzednim paragrafie prowadzi do szybkiego rozczarowania. Spostrzegamy, iż nie spełniają one założonych własności, co może łatwo skłonić do zwątpienia w sens całej prezentacji. Cóż, właściwie rzecz jest nieco bardziej złożona. Z jednej strony widzimy już, że idealne funkcje H nie istnieją⁹, z drugiej

⁸ Programowo można otrzymać tę wartość za pomocą instrukcji `int (fmod (11732 * 0.61803398887, 1) * 30)`; ponadto należy na początku programu dopisać `#include <math.h>`.

⁹ Daje się to nawet uzasadnić teoretycznie (patrz np. dobrze znany w statystyce tzw. *paradoks urodzin*).

zaś strony dziwnym byłoby zaczynać dyskusję o transformacji kluczowej i doprowadzić ją do stwierdzenia, że... jej realizacja nie jest możliwa praktycznie! Oczywiście nie jest aż tak źle. Istnieje kilka metod, które pozwalają poradzić sobie w zadowalający sposób z zauważonymi niedoskonałościami, i one właśnie będą stanowić przedmiot naszych dalszych rozważań.

Powrót do źródeł

Co robić w przypadku stwierdzenia kolizji dwóch odmiennych rekordów, którym funkcja H przydzieliła ten sam indeks w tablicy T ? Okazuje się, że można sobie poradzić poprzez pewną zmianę w samej filozofii transformacji kluczowej. Otóż jeśli umówimy się, że w tablicy T zamiast rekordów będziemy zapamiętywać *głowy* list do elementów charakteryzujących się tym samym kluczem, wówczas problem mamy... z głowy! Istotnie, jeśli wstawiając element x do tablicy pod indeks m , stwierdzimy, że już wcześniej ktoś się tam „zameldował”, wystarczy doczepić x na koniec listy, której głowa jest zapamiętana w $T[m]$.

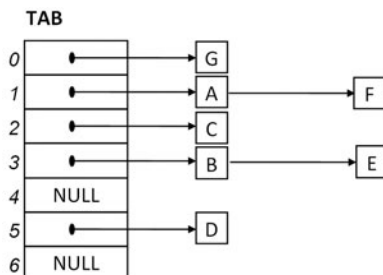
Analogicznie działa poszukiwanie: szukamy elementu x i $H(x)$ zwraca nam pewien indeks m . W przypadku gdy $T[m]$ zawiera NULL, możemy być pewni, że szukanego elementu nie odnaleźliśmy — w odwrotnej sytuacji, aby się ostatecznie upewnić, wystarczy przeszukać listę $T[m]$. (Warto przy okazji zauważyć, że listy będą na ogół bardzo krótkie).

Opisany powyżej sposób jest zilustrowany na rysunku 7.2.

Obrazuje on sytuację powstałą po sukcesywnym wstawianiu do tablicy T rekordów A, B, C, D, E, F i G, którym funkcja H przydzieliła adresy (indeksy): 1, 3, 2, 5, 3, 1 i 0. Indeksy tablicy, pod którymi nie ukrywają się żadne rekordy danych, są zainicjowane wartością NULL — patrz np. komórki 4 i 6. Na pozycji 1 mamy do czynienia z konfliktem dostępu: rekordy A i F otrzymały ten sam adres! Odpowiednia funkcja *wstaw* (którą musimy przewidująco napisać!) wykrywa tę sytuację i wstawia element F na koniec listy $T[1]$.

Rysunek 7.2.

Użycie list do obsługi konfliktów dostępu



Podobna sytuacja dotyczy rekordów B i E. Proces poszukiwania elementów jest zbliżony do ich wstawiania — Czytnik nie powinien mieć trudności z dokładnym odtworzeniem sposobu przeszukiwania tablicy T w celu odpowiedzi na pytanie, czy został w niej zapamiętany dany rekord, np. E.

Co jest niepokojące w zaproponowanej powyżej metodzie? Zaprezentowana wcześniej idea transformacji kluczowej zawiera zachęcającą obietnicę porzucenia wszelkich list, drzew i innych skomplikowanych w obsłudze struktur danych na rzecz zwykłego odwzorowania:

$$\text{dane} \mapsto \text{adres komórki w pamięci}$$

Podczas dokładniejszej analizy napotkaliśmy jednak mały problem i... powróciliśmy do starych, dobrych list. Z tych właśnie przyczyn rozwiązanie to można ze spokojnym sumieniem uznać za nieco sztuczne¹⁰ — równie dobrze można było trzymać się list i innych dynamicznych

¹⁰ Choć parametry czasowe tej metody są bardzo korzystne.

struktur danych, bez wprowadzania do nich dodatkowo elementów transformacji kluczowej! Czy możemy w tej sytuacji mieć nadzieję na rozwiązanie problemów dotyczących kolizji dostępu? Zainteresowanych odpowiedzią na to pytanie zachęcam do lektury następnych paragrafów.

Jeszcze raz tablice!

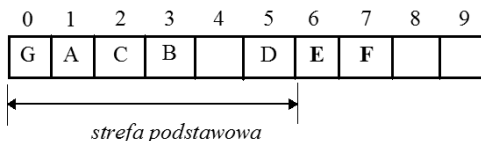
Metoda transformacji kluczowej została z założenia przypisana aplikacjom, które — pozwalając przewidzieć maksymalną liczbę rekordów do zapamiętania — umożliwiają zarezerwowanie pamięci na statyczną tablicę stwarzającą łatwy, indeksowany dostęp do nich. Jeśli możemy zarezerwować tablicę na wszystkie elementy, które chcemy zapamiętać, może by jej część przeznaczyć na obsługę konfliktów dostępu?

Idea polegałaby na podziale tablicy T na dwie części: strefę *podstawową* i strefę *przepełnienia*. Do tej drugiej elementy trafiałyby w momencie stwierdzenia braku miejsca w części podstawowej. Strefa przepełnienia byłaby wypełniana liniowo wraz z napływem nowych elementów „kolizyjnych”. W celu zilustrowania nowego pomysłu spróbujmy wykorzystać dane z rysunku 7.1, zakładając rozmiar stref: *podstawowej* i *przepełnienia* na odpowiednio: 6 i 4.

Efekt wypełnienia tablicy jest przedstawiony na rysunku 7.3.

Rysunek 7.3.

Podział tablicy
do obsługi
konfliktów dostępu



Rekordy E i F zostały zapamiętane w momencie stwierdzenia przepełnienia na kolejnych pozycjach 6 i 7. Sugeruje to, że gdzieś „w tle” musi istnieć zmienna zapamiętująca ostatnią wolną pozycję strefy przepełnienia.

Również w jakiś sposób należy się umówić co do oznaczania pozycji wolnych w strefie podstawowej — to już leży w gestii programisty i zależy w dużym stopniu od struktury rekordów, które będą zapamiętywane.

Rozwiązanie uwzględniające podział tablic nie należy do skomplikowanych, co jest jego niewątpliwą zaletą. Stworzenie funkcji *wstaw* i *szukaj* jest kwestią kilku minut i zostaje powierzone Czytelnikowi jako proste ćwiczenie.

Dla ścisłości należy jednak wskazać pewien słaby punkt. Otóż nie jest zbyt oczywiste, co należy zrobić w przypadku zapelnienia strefy... przepełnienia! (Wypisanie „ładnego” komunikatu o błędzie nie likwiduje problemu). Użycie tej metody powinno być poprzedzone szczególnie starannym obliczeniem rozmiarów tablic, tak aby nie załamać aplikacji w najbardziej niekorzystnym momencie — na przykład przed zapisem danych na dysk.

Próbkowanie liniowe

W opisaną poprzednio metodzie w sposób nieco sztuczny rozwiązaliśmy problem konfliktów dostępu w tablicy T . Podzieliśmy ją mianowicie na dwie części służące do zapamiętywania rekordów, ale w różnych sytuacjach. O ile jednak dobór ogólnego rozmiaru tablicy R_{max} jest w wielu aplikacjach łatwy do przewidzenia, to dobranie właściwego rozmiaru strefy przepełnienia jest w praktyce bardzo trudne. Ważną rolę grają tu bowiem zarówno dane, jak i funkcja H i w zasadzie należałoby je analizować jednocześnie, aby w przybliżony sposób oszacować właściwe rozmiary obu części tablic. Problem oczywiście znika samoczynnie, gdy dysponujemy bardzo dużą ilością wolnej pamięci, jednak przewidywanie a priori takiego przypadku mogłoby być dość niebezpieczne.

Jak zauważyliśmy wcześniej, konflikty dostępu są w metodzie transformacji kluczowej nieuchronne. Powód jest prosty: nie istnieje idealna funkcja H , która rozmieściłaby równomiernie wszystkie R_{max} elementów po całej tablicy T . Jeśli taka jest rzeczywistość, to może zamiast walczyć z nią — jak to usiłowaliśmy czynić poprzednie metody — spróbować się do niej dopasować?

Idea jest następująca: w momencie zapisu nowego rekordu¹¹ do tablicy, w przypadku stwierdzenia konfliktu, możemy spróbować zapisać element w pierwszym wolnym miejscu. Algorytm funkcji wstaw byłby wówczas następujący (zakładamy próbę zapisu do tablicy T rekordu x charakteryzowanego kluczem v):

```
int pos=H(x.v);
while (T[pos] != WOLNE)
    pos = (pos+1) % Rmax;
T[pos]=x;
```

Załóżmy teraz, że poszukujemy elementu charakteryzującego się kluczem k . W takim przypadku funkcja szukaj mogłaby wyglądać następująco:

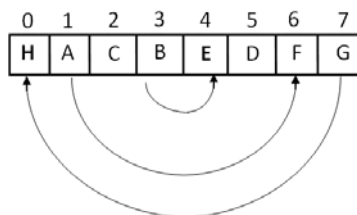
```
int pos=H(k);
while ((T[pos] != WOLNE) && (T[pos].v != k))
    pos = (pos+1) % Rmax;
return T[pos]; //zwraca znaleziony element
```

Różnica pomiędzy poszukiwaniem i wstawianiem jest w przypadku transformacji kluczowej doprawdy nieznaczna. Algorytmy są celowo zapisane w pseudokodzie, bowiem sensowny przykład korzystający z tej metody musiałby zawierać dokładne deklaracje typu danych, tablicy, funkcji H , wartości specjalnej $WOLNE$ — analiza tego byłaby bardzo nużąca. Instrukcja $pos = (pos+1) \% Rmax$ zapewnia nam powrót do początku tablicy w momencie dotarcia do jej końca podczas (kolejnych) iteracji pętli `while`.

Dla ilustracji spójrzmy, jak poradzi sobie nowa metoda przy próbie sukcesywnego wstawiania do tablicy T rekordów A, B, C, D, E, F, G i H, którym funkcja H przydzieliła adresy (indeksy): 1, 3, 2, 5, 3, 1, 7 i 7. Ustalmy ponadto rozmiar tablicy T na 8 — wyłącznie w ramach przykładu, bowiem w praktyce taka wartość nie miałaby zbytniego sensu. Efekt jest przedstawiony na rysunku 7.4:

Rysunek 7.4.

Obsługa konfliktów
dostępu przez
próbkiowanie liniowe



Dość ciekawymi jawią się teoretyczne wyliczenia *średniej liczby prób* potrzebnej do odnalezienia danej x . W przypadku poszukiwania zakończonym sukcesem średnia liczba prób wynosi około:

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right),$$

gdzie α jest współczynnikiem zapelnienia tablicy T . Analogiczny wynik dla poszukiwania zakończonym niepowodzeniem wynosi około:

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right).$$

¹¹ Oczywiście może to być również dowolna zmienna prosta!

Przykładowo: dla tablicy wypełnionej w dwóch trzecich swojej pojemności ($\alpha = \frac{2}{3}$) liczby te wyniosą odpowiednio: 2 i 5.

W praktyce należy unikać szczelnego wypełniania tablicy T, gdyż zacytowane powyżej liczby stają się bardzo duże (α nie powinno przybierać wartości bliskich 1). Powyższe wzory zostały wyprowadzone przy założeniu funkcji H, która rozsiewa równomiernie elementy po dużej tablicy T. Te zastrzeżenia są tu bardzo istotne, bowiem podane wyżej rezultaty mają charakter statystyczny.

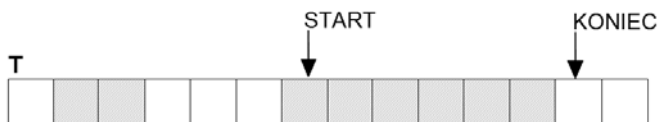
Podwójne kluczowanie

Stosowanie próbkowania liniowego prowadzi do niekorzystnego liniowego wypełniania tablicy T, co kłóci się z wymogiem narzuconym wcześniej funkcji H (patrz strona 169). Intuicyjnie rozwiązanie tego problemu nie wydaje się trudne: trzeba coś zrobić, aby nieco bardziej losowo porozrzucać elementy. Próbkowanie liniowe nie było z tego względu dobrym pomysłem, gdyż — napotkawszy pewien zapelniony obszar tablicy T — proponowało wstawienie nowego elementu tuż za nim — jeszcze go powiększając! Czytelnik mógłby zadać pytanie: a dlaczego jest to aż takie groźne? Oczywiście względy estetyczne nie grają tu żadnej roli: zauważmy, że liniowo zapelniony obszar przeszkadza w szybkim znalezieniu wolnego miejsca na wstawienie nowego elementu! Fakt ten utrudnia również sprawne poszukiwanie danych.

Rozpatrzmy prosty przykład przedstawiony na rysunku 7.5.

Rysunek 7.5.

Utrudnione poszukiwanie danych przy próbkowaniu liniowym



Na rysunku tym zacieniowane komórki tablicy oznaczają miejsca już zajęte. Funkcja $H(k)$ dostarczyła pewien indeks, od którego zaczyna się przeszukiwana strefa tablicy (poszukujemy pewnego elementu charakteryzującego się kluczem k) — powiedzmy, że zaczynamy poszukiwanie od indeksu oznaczonego symbolicznie jako START. Proces poszukiwania zakończy się sukcesem w przypadku trafienia w poszukiwany rekord — aby to stwierdzić, czynimy dość kosztowne¹² porównanie $T[pos].v \neq k$ (patrz algorytm procedury szukaj omówiony w poprzednim punkcie).

Co więcej, wykonujemy je za każdym razem podczas przesuwania się po liniowo wypełnionej strefie! Informację o ewentualnej porażce poszukiwań dostajemy dopiero po jej całkowitym sprawdzeniu i natrafieniu na pierwsze wolne miejsce. W naszym rysunkowym przykładzie dopiero po siedmiu porównaniach algorytm natrafi na pustą komórkę (oznaczoną etykietą KONIEC), która poinformuje go o daremności podjętego uprzednio wysiłku. Gdyby zaś tablica była zapelniona w mniej liniowy sposób, statystycznie o wiele szybciej natrafilibyśmy na WOLNE miejsce, co automatycznie zakończyłoby proces poszukiwania zakończonego porażką.

Na szczęście istnieje łatwy sposób uniknięcia liniowego grupowania elementów: tzw. *podwójne kluczowanie*. W chwili napotkania kolizji następuje próba rozrzucenia elementów za pomocą drugiej, pomocniczej funkcji H.

Procedura wstaw pozostaje niemal niezmienniona:

```
int pos = H1(x.v);
int krok = H2(x.v);
while (T[pos] != WOLNE)
```

¹² Koszt operacji porównania zależy od stopnia złożoności klucza, tzn. od liczby i typów pól rekordu, które go tworzą.

```
pos = (pos+krok) % Rmax;
T[pos]=x;
```

Procedura poszukiwania jest bardzo podobna i Czytelnik z pewnością będzie w stanie napisać ją samodzielnie, wzorując się na przykładzie poprzednim.

Przedyskutujmy teraz problem doboru funkcji H_2 . Nie trudno się domyślić, iż ma ona duży wpływ na jakość procesu wstawiania (i oczywiście poszukiwania!). Przede wszystkim funkcja H_2 powinna być różna od H_1 ! W przeciwnym przypadku doprowadzilibyśmy tylko do bardziej skomplikowanego tworzenia stref „ciąglych” — a właśnie od tego chcemy uciec. Kolejny wymóg jest oczywisty: musi być to funkcja prosta, która nie spowolni nam procesu poszukiwania i wstawiania. Przykładem takiej prostej i jednocześnie skutecznej w praktyce funkcji może być $H_2(k) = 8 - (k \% 8)$: zakres skoku jest dość szeroki, a prostota niezaprzeczalna!

Metoda *podwójnego kluczowania* jest interesująca z uwagi na widoczny zysk w szybkości poszukiwania danych. Popatrzmy na teoretyczne rezultaty wyliczeń średniej liczby prób przy poszukiwaniu zakończonym sukcesem i porażką. W przypadku poszukiwania zakończonym sukcesem średnia liczba prób wynosi około:

$$\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$$

(gdzie α jest, tak jak poprzednio, współczynnikiem zapęnlienia tablicy T).

Analogiczny wynik dla poszukiwania zakończonym niepowodzeniem wynosi około:

$$\frac{1}{1-\alpha}$$

Zastosowania transformacji kluczowej

Dotychczas obracaliśmy się wyłącznie w kręgu elementarnych przykładów: tablice o małych rozmiarach, proste klucze znakowe lub liczbowe itp. Rzeczywiste aplikacje mogą być oczywiście znacznie bardziej skomplikowane i dopiero wówczas Czytelnik będzie mógł w pełni docenić wartość posiadanej wiedzy. Zastosowania transformacji kluczowej mogą być dość nieoczekiwane: dane wcale nie muszą znajdować się w pamięci głównej; w przypadku programu bazy danych można w dość łatwy sposób użyć H -kodu do sprawnego odszukiwania danych. Konstruując duży kompilator lub konsolidator, możemy wykorzystać metody transformacji kluczowej do odszukiwania skompilowanych modułów w dużych plikach bibliotecznych.

Podsumowanie metod transformacji kluczowej

Transformacja kluczowa poddaje się dobrze badaniom porównawczym — otrzymywane wyniki są wiarygodne i intuicyjnie zgodne z rzeczywistością. Niestety sposób ich wyprowadzenia jest skomplikowany i ze względów czysto humanitarnych zostanie tu opuszczony. Mimo to ogólne wnioski o charakterze praktycznym są warte zacytowania:

- ◆ Przy słabym wypełnieniu¹³ tablicy T wszystkie metody są w przybliżeniu tak samo efektywne.
- ◆ Metoda *próbkiowania liniowego* doskonale sprawdza się przy dużych, słabo wykorzystanych tablicach T (czyli wówczas, gdy dysponujemy dużą ilością wolnej pamięci). Za jej stosowaniem przemawia również niewątpliwa prostota.

¹³ Tzn. do ok. 30 – 40% całkowitej objętości tablicy.

Na koniec warto podkreślić coś, o czym w ferworze prezentacji rozmaitych metod i dyskusji mogliśmy łatwo zapomnieć: transformacja kluczowa jest narzędziem wprost idealnym... ale tylko w przypadku obsługi danych, których liczba jest z dużym prawdopodobieństwem przewidywalna. Nie możemy sobie bowiem pozwolić na załamanie się aplikacji z powodu naszych zbyt nieostrożnych oszacowań rozmiarów tablic!

Przykładowo: wiedząc, że będziemy mieli do czynienia ze zbiorem rekordów w liczbie ustalonej na przykład na 700, deklarujemy tablicę `T` o rozmiarze 1000, co zagwarantuje nam szybkie poszukiwanie i wstawianie danych nawet przy zapisie wszystkich 700 rekordów. Wypełnienie tablicy w 70 – 80% okazuje się tą magiczną granicą, po przekroczeniu której sens stosowania transformacji kluczowej staje się coraz mniej widoczny — dlatego po prostu nie warto zbliżać się do niej zbliżać. Niemniej metoda jest ciekawa i warta stosowania — oczywiście gdy uwzględnimy kontekst praktyczny aplikacji końcowej.



Uwaga

Kilka przykładowych funkcji `H` zostało zebranych w pliku `hash.cpp` — zachęcam do eksperymentów i symulowania ich funkcjonowania dla różnych zbiorów danych.

Rozdział 8.

Przeszukiwanie tekstów

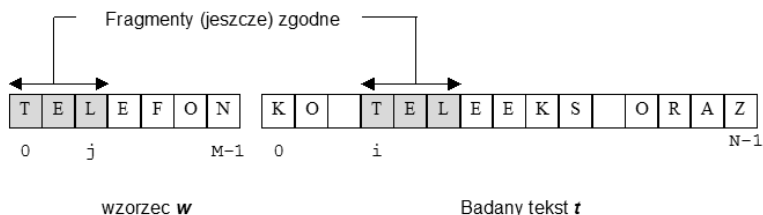
Zanim na dobre zanurzymy się w lekturę nowego rozdziału, należy wyjaśnić pewne nieporozumienie, które może towarzyszyć jego tytułowi. Otóż za *tekst* będziemy uważali ciąg znaków w sensie informatycznym. Nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną”! Tekstem będzie na przykład również ciąg bitów¹, który tylko przez umowność może być podzielony na równej wielkości porcje, którym przyporządkowano pewien kod liczbowy².

Okazuje się wszelako, że przyjęcie konwencji dotyczących interpretacji informacji ułatwia wiele operacji na niej. Dlatego też pozostaniemy przy ogólnikowym stwierdzeniu „tekst”, wiedząc, że za tym określeniem może się kryć dość sporo znaczeń.

Algorytm typu brute-force

Zadaniem, które będziemy usiłowali wspólnie rozwiązać, jest poszukiwanie wzorca³ w o długości M znaków w tekście t o długości N . Z łatwością możemy zaproponować dość oczywisty algorytm rozwiązujący to zadanie, bazując na pomysłach symbolicznie przedstawionych na rysunku 8.1.

Rysunek 8.1.
*Algorytm typu
brute-force
przeszukiwania tekstu*



Zarezerwujmy indeksy j i i do poruszania się odpowiednio we wzorcu i tekście podczas operacji porównywania znak po znaku zgodności wzorca z tekstem. Załóżmy, że w trakcie poszukiwań obszary objęte szarym kolorem na rysunku okazały się zgodne. Po stwierdzeniu tego faktu przesuwamy się zarówno we wzorcu, jak i w tekście o jedną pozycję do przodu ($i++$; $j++$).

Cóż się jednak powinno stać z indeksami i oraz j podczas stwierdzenia niezgodności znaków? W takiej sytuacji całe poszukiwanie kończy się porażką, co zmusza nas do anulowania „szarej strefy” zgodności. Czynimy to poprzez cofnięcie się w tekście o to, co było zgodne, czyli o $j-1$.

¹ Reprezentujący np. pamięć ekranu.

² Np. ASCII lub dowolny inny.

³ Ang. *pattern matching*.

znaków, wyzerowując przy okazji j . Omówmy jeszcze moment stwierdzenia całkowitej zgodności wzorca z tekstem. Kiedy to nastąpi? Otóż nietrudno zauważyć, że podczas stwierdzenia zgodności ostatniego znaku j powinno zwrócić się z M . Możemy wówczas łatwo odtworzyć pozycję, od której wzorec startuje w badanym tekście: będzie to oczywiście $i-M$.

Tłumacząc powyższe sytuacje na C++, możemy łatwo dojść do następującej procedury:



txt-1.cpp

```
int szukaj(char *w, char *t)
{
    int i=0, j=0, M=strlen(w), N=strlen(t);
    while( (j<M) && (i<N) )
    {
        if(t[i]!=w[j])
        {
            i-=j-1;
            j=-1;
        }
        i++;
        j++;
    }
    if(j==M)
        return i-M;
    else
        return -1;
}
```

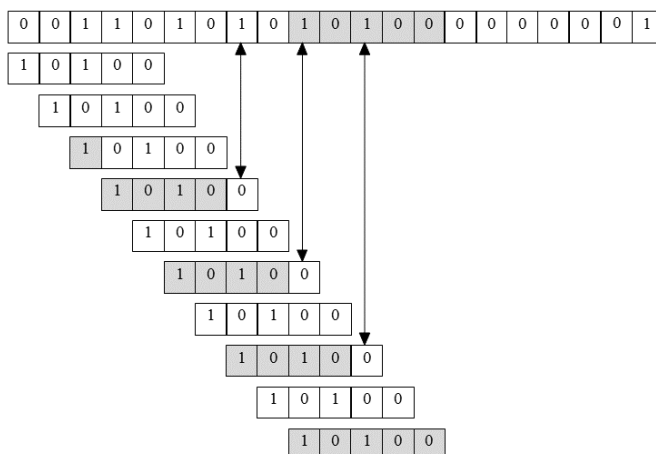
Sposób korzystania z funkcji `szukaj` jest przedstawiony na przykładzie następującej funkcji `main`:

```
int main()
{
    char *b="abrakadabra", *a="rak";
    cout << szukaj(a,b) << endl;
}
```

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorec, lub -1 w przypadku, gdy poszukiwany tekst nie został odnaleziony — jest to znana nam już doskonale konwencja. Przypatrzmy się dokładniej przykładowi poszukiwania wzorca `10100` w pewnym tekście binarnym (patrz rysunek 8.2).

Rysunek 8.2.

„Falszywe starty”
podczas poszukiwania



Rysunek jest nieco uproszczony: w istocie poziome przesuwanie się wzorca oznacza instrukcje zaznaczone na listingu jako (*), natomiast cała szara strefa o długości k oznacza k -krotne wykonanie (**).

Na podstawie zobrazowanego przykładu możemy spróbować wymyślić taki najgorszy tekst i wzorec, dla których proces poszukiwania będzie trwał możliwie najdłużej. Chodzi oczywiście zarówno o tekst, jak i wzorec złożone z samych zer i zakończone jedynką⁴.

Spróbujmy obliczyć klasę tego algorytmu dla opisanego przed chwilą ekstremalnego najgorszego przypadku. Obliczenie nie należy do skomplikowanych czynności: zakładając, że restart algorytmu będzie konieczny $(N-1) \cdot (M-2) = N \cdot M - 1$ razy, i wiedząc, że podczas każdego cyklu jest konieczne wykonanie M porównań, otrzymujemy natychmiast $M(N-M+1)$, czyli około⁵ $M \cdot N$.

Zaprezentowany w tym paragrafie algorytm wykorzystuje komputer jako bezmyślne, ale sprawne liczydło⁶. Jego złożoność obliczeniowa eliminuje go w praktyce z przeszukiwania tekstów binarnych, w których może wystąpić wiele niekorzystnych konfiguracji danych. Jedyną zaletą algorytmu jest jego prostota, co i tak nie czyni go na tyle atrakcyjnym, by dać się zamęczyć jego powolnym działaniem.

Nowe algorytmy poszukiwań

Algorytm, o którym będzie mowa w tym rozdziale, posiada ciekawą historię, którą w formie anegdoty warto przytoczyć. Otóż w 1970 roku S. A. Cook udowodnił teoretyczny rezultat dotyczący pewnej abstrakcyjnej maszyny. Wynikało z niego, że istniał algorytm poszukiwania wzorca w tekście, który działał w czasie proporcjonalnym do $M+N$ w najgorszym przypadku. Rezultat pracy Cooka wcale nie był przewidziany do praktycznych celów, niemniej D. E. Knuth i V. R. Pratt otrzymali na jego podstawie algorytm, który można już było zaimplementować w komputerze — ukazując przy okazji, iż pomiędzy praktycznymi realizacjami a rozważaniami teoretycznymi nie istnieje wcale aż tak ogromna przepaść, jakby się to mogło wydawać. W tym samym czasie J. H. Morris odkrył dokładnie ten sam algorytm jako rozwiązanie problemu, który napotkał podczas praktycznej implementacji edytora tekstu. Algorytm $K-M-P$ — bo tak będziemy go dalej zwali — jest jednym z przykładów dość częstych w nauce odkryć równoległych: z jakichś niewiadomych powodów nagle kilku pracujących osobno ludzi dochodzi do tego samego dobrego rezultatu. Prawda, że jest w tym coś niesamowitego i aż się prosi o jakieś metafizyczne hipotezy?

Knuth, Morris i Pratt opublikowali swój algorytm dopiero w 1976 roku. W międzyczasie pojawił się kolejny „cudowny” algorytm, tym razem autorstwa R. S. Boyera i J. S. Moore’a, który okazał się w pewnych zastosowaniach znacznie szybszy od algorytmu $K-M-P$. Został on również równolegle wynaleziony (odkryty?) przez R. W. Gospera. Oba te algorytmy są jednak dość trudne do zrozumienia bez pogłębionej analizy, co utrudniło ich rozprowadzenie.

W roku 1980 R. M. Karp i M. O. Rabin doszli do wniosku, że przeszukiwanie tekstów nie jest aż tak dalekie od standardowych metod przeszukiwania, i wynaleźli algorytm, który — działając ciągle w czasie proporcjonalnym do $M+N$ — jest ideowo zbliżony do poznanego już przez nas algorytmu typu *brute-force*. Na dodatek jest to algorytm, który można względnie łatwo uogólnić na przypadek poszukiwania w tablicach 2-wymiarowych, co czyni go potencjalnie użytecznym w obróbce obrazów.

W następnych trzech sekcjach szczegółowo omówimy sobie wspomniane w tym przeglądzie historycznym algorytmy.

⁴ Zera i jedynki symbolizują tu dwa różne od siebie znaki.

⁵ Zwykle M będzie znacznie mniejsze niż N .

⁶ Termin *brute-force* jeden z moich znajomych ślicznie przetłumaczył jako „metodę mastodonta”.

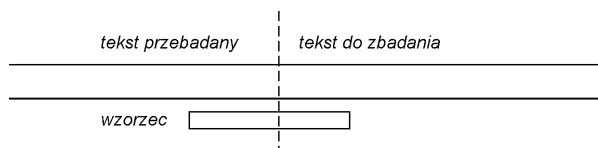
Algorytm K-M-P

Wadą algorytmu *brute-force* jest jego czułość na konfigurację danych: fałszywe restarty są tu bardzo kosztowne; w analizie tekstu cofamy się o całą długość wzorca, zapominając po drodze wszystko, co przetestowaliśmy do tej pory. Narzuca się tu niejako chęć skorzystania z informacji, które już w pewien sposób posiadamy — przecież w następnym etapie będą wykonywane częściowo te same porównania, co poprzednio!

W pewnych szczególnych przypadkach przy znajomości struktury analizowanego tekstu możliwe jest ulepszenie algorytmu. Przykładowo: jeśli wiemy na pewno, iż w poszukiwanym wzorcu jego pierwszy znak nie pojawia się już w nim w ogóle⁷, to w razie restartu nie musimy cofać wskaźnika i o $j-1$ pozycji, jak to było poprzednio (patrz listing *txt-1.cpp*). W tym przypadku możemy po prostu zinkrementować i , wiedząc, że ewentualne powtórzenie poszukiwań na pewno nic by już nie dało. Owszem, można się łatwo zgodzić z twierdzeniem, iż tak wyspecjalizowane teksty zdarzają się relatywnie rzadko, jednak powyższy przykład ukazuje, iż ewentualne manipulacje algorytmami poszukiwań są ciągle możliwe — wystarczy się tylko rozejrzeć. Idea algorytmu *K-M-P* polega na wstępnym zbadaniu wzorca w celu obliczenia liczby pozycji, o które należy cofnąć wskaźnik i w przypadku stwierdzenia niezgodności badanego tekstu ze wzorcem. Oczywiście można również rozumować w kategoriach przesuwania wzorca do przodu — rezultat będzie ten sam. To właśnie tę drugą konwencję będziemy stosować dalej. Wiemy już, że powinniśmy przesuwać się po badanym tekście nieco inteligentniej niż w poprzednim algorytmie. W przypadku zauważenia niezgodności na pewnej pozycji j wzorca⁸ należy zmodyfikować ten indeks, wykorzystując informację zawartą w już zbadanej „szarej strefie” zgodności.

Brzmi to wszystko (zapewne) niesłychanie tajemniczo, pora więc jak najszybciej wyjaśnić tę sprawę, aby uniknąć możliwych nieporozumień. Popatrzmy w tym celu na rysunek 8.3.

Rysunek 8.3.
Wyszukiwanie
optymalnego
przesunięcia
w algorytmie *K-M-P*



Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorec bardzo wolno w prawo, patrząc jednocześnie na już zbadany tekst — tak aby obserwować ewentualne pokrycie się tej części wzorca, która znajduje się po lewej stronie przerywanej kreski, z tekstem, który umieszczony jest powyżej wzorca. W pewnym momencie może okazać się, że następuje pokrycie obu tych części. Zatrzymujemy wówczas przesuwanie i kontynuujemy testowanie (znak po znaku) zgodności obu części znajdujących się za kreską pionową.

Od czego zależy ewentualne pokrycie się oglądanych fragmentów tekstu i wzorca? Otóż dość paradoksalnie badany tekst nie ma tu nic do powiedzenia — jeśli można to tak określić. Informacja o tym, jaki on był, jest ukryta w stwierdzeniu „ $j-1$ znaków było zgodnych” — w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie sam wzorec, odkryć poszukiwane optymalne przesunięcie. Na tym właśnie spostrzeżeniu opiera się idea algorytmu *K-M-P*. Okazuje się, że badając samą strukturę wzorca, można obliczyć, jak powinniśmy zmodyfikować indeks j w razie stwierdzenia niezgodności tekstu ze wzorcem na j -tej pozycji.

Zanim zagłębimy się w wyjaśnienia na temat obliczania owych przesunięć, popatrzmy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 8.4 możemy dostrzec, iż na siódmej pozycji wzorca⁹ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

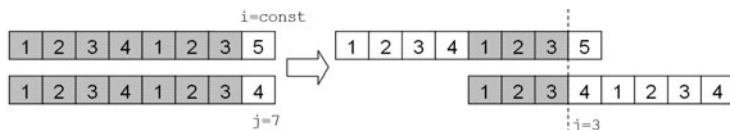
⁷ Przykład: „ABBBBBBB” — znak ‘A’ wystąpił tylko jeden raz.

⁸ Lub i w przypadku badanego tekstu.

⁹ Licząc indeksy tablicy tradycyjnie od zera.

Rysunek 8.4.

Przesuwanie się wzorca
w algorytmie K-M-P (1)

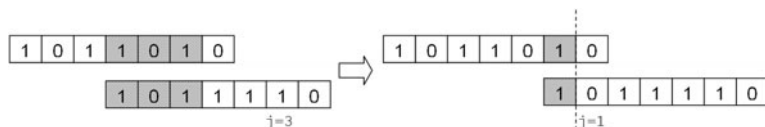


Jeśli zostawimy indeks i w spokoju, to — modyfikując wyłącznie j — możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? Ślizgając go wolno w prawo (patrz rysunek 8.4), doprowadzamy w pewnym momencie do nałożenia się ciągów 123 przed kreską — cała strefa niezgodności została wyprowadzona na prawo i ewentualne dalsze testowanie może być kontynuowane!

Analogiczny przykład znajduje się na rysunku 8.5.

Rysunek 8.5.

Przesuwanie się wzorca
w algorytmie K-M-P (2)



Tym razem niezgodność wystąpiła na pozycji $j=3$. Dokonując — podobnie jak poprzednio — przesuwania wzorca w prawo, zauważamy, iż jedyne możliwe nałożenie się znaków wystąpi po przesunięciu o dwie pozycje w prawo — czyli dla $j=1$. Dodatkowo okazuje się, że znaki za kreską też się pokryły, ale o tym algorytm dowie się dopiero podczas kolejnego testu zgodności na pozycji i .

Dla potrzeb algorytmu K-M-P konieczne okazuje się wprowadzenie tablicy przesunień `int shift[M]`. Sposób jej zastosowania będzie następujący: jeśli na pozycji j wystąpiła niezgodność znaków, to kolejną wartością j będzie `shift[j]`. Nie wnikając chwilowo w sposób inicjalizacji tej tablicy (odmiennej oczywiście dla każdego wzorca), możemy natychmiast podać algorytm K-M-P, który w konstrukcji jest niemal dokładną kopią algorytmu typu *brute-force*:

**kmp.cpp**

```
int kmp(char *w, char *t)
{
    int i, j, N=strlen(t);
    for(i=0, j=0; (i<N) && (j<M); i++, j++)
        while( (j>=0) && (t[i]!=w[j]) )
            j=shift[j];
    if (j==M)
        return i-M;
    else
        return -1;
}
```

Szczególnym przypadkiem jest wystąpienie niezgodności na pozycji zerowej: z założenia nie-możliwe jest tu przesuwanie wzorca w celu uzyskania nałożenia się znaków. Z tego powodu chcemy, aby indeks j pozostał niezmienny przy jednoczesnej progresji indeksu i . Jest to możliwe do uzyskania, jeśli umówimy się, że `shift[0]` zostanie zainicjowany wartością `-1`. Wówczas podczas kolejnej iteracji pętli `for` nastąpi inkrementacja i oraz j , co wyzeruje nam j .

Pozostaje do omówienia sposób konstrukcji tablicy `shift[M]`. Jej obliczenie powinno nastąpić przed wywołaniem funkcji `kmp`, co sugeruje, iż w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona praktycznie identyczna z `kmp` z tą tylko różnicą, iż algorytm sprawdza zgodność wzorca... z nim samym!

```

int shift[M];
void init_shifts(char *w)
{
    int i,j;
    shift[0]=-1;
    for(i=0,j=-1;i<M-1;i++,j++,shift[i]=j)
        while((j>=0)&&(w[i]!=w[j]))
            j=shift[j];
}

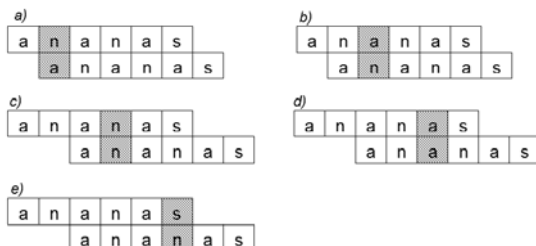
```

Sens tego algorytmu jest następujący: tuż po inkrementacji i i j wiemy, że pierwsze j znaków wzorca jest zgodne ze znakami na pozycjach: $p[i-j-1] \dots p[i-1]$ (ostatnie j pozycji w pierwszych i znakach wzorca). Ponieważ jest to największe j spełniające powyższy warunek, zatem aby nie ominąć *potencjalnego* miejsca wykrycia wzorca w tekście, należy ustawić $shift[i]$ na j .

Popatrzmy, jaki będzie efekt zadziałania funkcji `init_shifts` na słowie *ananas* (patrz rysunek 8.6). Zacięniowane litery oznaczają miejsca, w których wystąpiła niezgodność wzorca z tekstem. W każdym przypadku graficznie przedstawiono efekt przesunięcia wzorca — widać wyraźnie, które strefy pokrywają się przed strefą zacięniowaną (porównaj rysunek 8.5).

Rysunek 8.6.

Optymalne przesunięcia
wzorca „ananas”
w algorytmie K-M-P



Przypomnijmy jeszcze, że tablica `shift` zawiera nową wartość dla indeksu j , który przemieszcza się po wzorcu.

Algorytm K-M-P można zoptymalizować, jeśli znamy z góry wzorce, których będziemy poszukiwać. Przykładowo: jeśli bardzo często zdarza nam się szukać w tekstach słowa *ananas*, to w funkcję `kmp` można wbudować tablicę przesunięć:



ananas.cpp

```

int kmp_ananas(char *t)
{
    int i=-1;
    start: i++;
    et0: if (t[i]!='a')
        goto start;
        i++;
    et1: if (t[i]!='n')
        goto et0;
        i++;
    et2: if (t[i]!='a')
        goto et0;
        i++;
    et3: if (t[i]!='n')
        goto et1; i++;
        if (t[i]!='a')
            goto et2; i++;
        if (t[i]!='s')
            goto et3; i++;
    return i-6;
}

```


W celu właściwego odtworzenia etykiet należy oczywiście co najmniej raz wykonać funkcję `init_shifts` lub obliczyć samemu odpowiednie wartości. W każdym razie gra jest warta świeczki: powyższa funkcja charakteryzuje się bardzo zwinłym kodem wynikowym asemblerowym, jest zatem bardzo szybka. Posiadacze kompilatorów, które umożliwiają generację kodu wynikowego jako tzw. „assembly output”¹⁰, mogą z łatwością sprawdzić różnice pomiędzy wersjami `kmp` i `kmp_ananas`! Dla przykładu mogę podać, że w przypadku wspomnianego kompilatora GNU klasyczna wersja procedury `kmp` (wraz z `init_shifts`) miała objętość około 170 linii kodu asemblerowego, natomiast `kmp_ananas` zmieściła się w ok. 100 liniach. (Patrz: pliki z rozszerzeniem `s` na dyskiecie dla kompilatora GNU lub `asm` dla kompilatora Borland C++ 5.5).

Algorytm *K-M-P* działa w czasie proporcjonalnym do $M+N$ w najgorszym przypadku. Największy zauważalny zysk związany z jego użyciem dotyczy przypadku tekstów o wysokim stopniu samopowtarzalności — dość rzadko występujących w praktyce. Dla typowych tekstów zysk związany z wyborem metody *K-M-P* będzie zatem słabo zauważalny.

Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik `i` w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez cofania się w nim. W niektórych systemach może to mieć istotne znaczenie praktyczne — przykładowo: mamy zamiar analizować bardzo długi plik tekstowy i charakter wykonywanych operacji nie pozwala na cofnięcie się w tej czynności (`i` w odczytywanym na bieżąco pliku).

Algorytm Boyera i Moore’a

Kolejny algorytm, który będziemy omawiali, jest ideowo znacznie prostszy do zrozumienia niż algorytm *K-M-P*. W przeciwieństwie do metody *K-M-P* porównywaniu ulega ostatni znak wzorca. To niekonwencjonalne podejście niesie ze sobą kilka istotnych zalet:

- ♦ Jeśli podczas porównywania okaże się, że rozpatrywany aktualnie znak nie wchodzi w ogóle w skład wzorca, wówczas możemy „skoczyć” w analizie tekstu o całą długość wzorca do przodu! Ciężar algorytmu przesunął się więc z analizy ewentualnych zgodności na badanie niezgodności — a te ostatnie są statystycznie znacznie częściej spotykane.
- ♦ Skoki wzorca są zazwyczaj znacznie większe od 1 — porównaj z metodą *K-M-P*!

Zanim przejdziemy do szczegółowej prezentacji kodu, omówimy sobie na przykładzie jego działanie. Spójrzmy w tym celu na rysunek 8.7, gdzie przedstawione jest poszukiwanie ciągu znaków „*lek*” w tekście „*Z pamiętnika młodej lekarki*”¹¹.

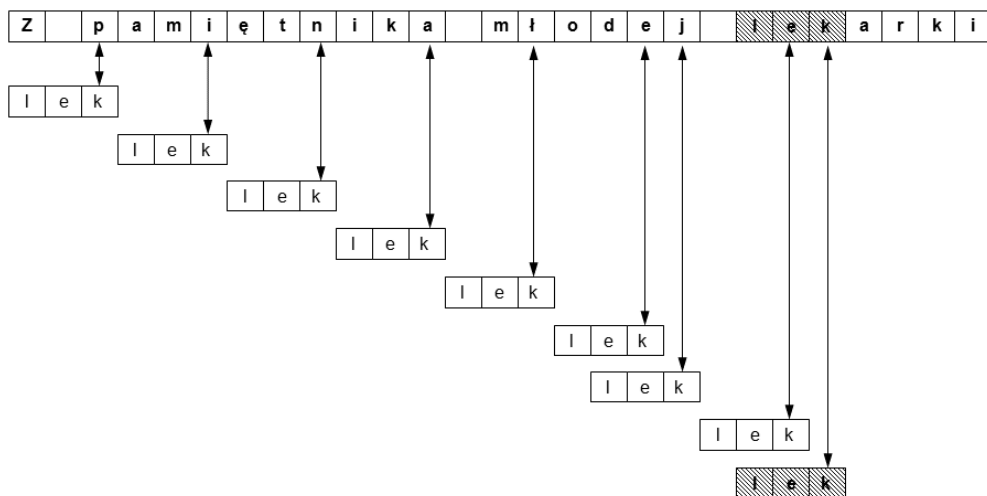
Pierwsze pięć porównań trafia na litery: `p`, `i`, `n`, `a` i `ł`, które we wzorcu nie występują! Za każdym razem możemy zatem przeskoczyć w tekście o trzy znaki do przodu (długość wzorca). Porównanie szóste trafia jednak na literę `e`, która w słowie „*lek*” występuje. Algorytm wówczas przesunął wzorec o tyle pozycji do przodu, aby litery `e` nałożyły się na siebie, i porównywanie jest kontynuowane.

Następnie okazuje się, że litera `j` nie występuje we wzorcu — mamy zatem prawo przesunąć się o kolejne 3 znaki do przodu. W tym momencie trafiamy już na poszukiwane słowo, co następuje po jednokrotnym przesunięciu wzorca, tak aby pokryły się litery `k`.

Algorytm jest, jak widać, klarowny, prosty i szybki. Jego realizacja także nie jest zbyt skomplikowana. Podobnie jak w przypadku metody poprzedniej, także i tu musimy wykonać pewną prekompilację w celu stworzenia tablicy przesunięć. Tym razem jednak tablica ta będzie miała tyle pozycji, ile jest znaków w alfabecie — wszystkie znaki, które mogą wystąpić w tekście plus spacja.

¹⁰ W przypadku kompilatorów popularnej serii Borland C++ należy skompilować program ręcznie poprzez polecenie `bcc32 -S -lxxx plik.cpp`, gdzie `xxx` oznacza katalog z plikami typu `H`; identyczna opcja istnieje w kompilatorze GNU C++, należy wystukać: `c++ -S plik.cpp`.

¹¹ Tytuł znakomitego cyklu autorstwa Ewy Szumańskiej.



Rysunek 8.7. Przeszukiwanie tekstu metodą Boyera i Moore'a

Będziemy również potrzebowali prostej funkcji indeks, która zwraca w przypadku spacji liczbę zero — w pozostałych przypadkach numer litery w alfabecie. Poniższy przykład uwzględnia jedynie kilka polskich liter — Czytelnik uzupełni go z łatwością o brakujące znaki. Numer litery jest oczywiście zupełnie arbitralny i zależy od programisty. Ważne jest tylko, aby nie pominąć w tablicy żadnej litery, która może wystąpić w tekście. Jedną z możliwych wersji funkcji indeks jest przedstawiona poniżej:



bm.cpp

```
const int K=26*2+2*2+1;// znaki ASCII + polskie litery + odstep
int shift[K];
int indeks(char c)
{
    switch(c)
    {
        case ' ':return 0;           // odstep = 0
        case 'e':return 53;
        case 'E':return 54;         // polskie litery
        case 'ł':return 55;
        case 'Ł':return 56;         // itd. dla pozostałych polskich liter
        default:
            if(islower(c))
                return c-'a'+1;
            else
                return c-'A'+27;
    }
}
```

Funkcja indeks ma jedynie charakter usługowy. Służy ona m.in. do właściwej inicjalizacji tablicy przesunięć. Mając za sobą analizę przykładu z rysunku 8.7, Czytelnik nie powinien być zbyt zadowolony sposobem inicjalizacji:

```
void init_shifts(char *w)
{
    int M=strlen(w);
    for(int i=0; i<K; i++)
        shift[i]=M;
```

```

for(int i=0; i<M; i++)
    shift[indeks(w[i])] = M-i-1;
}

```

Przejdźmy wreszcie do prezentacji samego listingu algorytmu, z przykładem wywołania:

```

int bm(char *w, char *t)
{
    init_shifts(w);
    int i, j, N=strlen(t), M=strlen(w);
    for(i=M-1, j=M-1; j>0; i--, j--)
        while(t[i]!=w[j])
        {
            int x=shift[indeks(t[i])];
            if(M-j>x)
                i+=M-j;
            else
                i+=x;
            if (i>=N)
                return -1;
            j=M-1;
        }
    return i;
}

int main()
{
    char *t="Z pamiętnika młodej lekarki";
    cout << "Wynik poszukiwań=" <<bm("lek",t)<<endl;
}

```

Algorytm *Boyer'a* i *Moore'a*, podobnie jak i *K-M-P*, jest klasy $M+N$ — jednak jest on o tyle od niego lepszy, iż w przypadku krótkich wzorców i długiego alfabetu kończy się po około M/N porównaniach. W celu obliczenia optymalnych przesunięć¹² autorzy algorytmu proponują skombinowanie powyższego algorytmu z tym zaproponowanym przez Knutha, Morrisa i Pratta. Celowość tego zabiegu wydaje się jednak wątpliwa, gdyż optymalizując sam algorytm, można w bardzo łatwy sposób uczynić zbyt czasochłonnym sam proces prekompilacji wzorca.

Algorytm Rabina i Karpa

Ostatni algorytm do przeszukiwania tekstów, który będziemy analizowali, wymaga znajomości rozdziału 7. i terminologii, która została w nim przedstawiona.

Algorytm *Rabina i Karpa* polega bowiem na dość przewrotnej idei:

- ♦ Wzorec w (do odszukania) jest *kluczem* (patrz terminologia transformacji kluczowej w rozdziale 7.) o długości M znaków, charakteryzującym się pewną wartością wybranej przez nas funkcji H . Możemy zatem obliczyć jednokrotnie $H_w=H(w)$ i korzystać z tego wyliczenia w sposób ciągły.
- ♦ Tekst wejściowy t (do przeszukania) może być w taki sposób odczytywany, aby na bieżąco znać M ostatnich znaków¹³. Z tych M znaków wyliczamy na bieżąco $H_t=H(t)$.

Gdy założymy jednoznaczność wybranej funkcji H , sprawdzenie zgodności wzorca z aktualnie badanym fragmentem tekstu sprowadza się do odpowiedzi na pytanie: czy H_w jest równe H_t ? Spostrzegawczy Czytelnik ma jednak prawo pokręcić w tym miejscu z powątpiewaniem głową: przecież to nie ma prawa działać szybko! Istotnie pomysł wyliczenia dodatkowo funkcji H dla

¹² Rozważ np. wielokrotne występowanie takich samych liter we wzorcu.

¹³ Na samym początku będzie to oczywiście M pierwszych znaków tekstu.

każdego słowa wejściowego o długości M wydaje się tak samo kosztowny — jeśli nie bardziej! — jak zwykle sprawdzanie tekstu znak po znaku (np. stosując algorytm typu *brute-force*). Tym bardziej że jak do tej pory nie powiedzieliśmy ani słowa na temat funkcji H ! Z poprzedniego rozdziału pamiętamy zapewne, iż jej wybór wcale nie był taki oczywisty.

Omawiany algorytm jednak istnieje i na dodatek działa szybko! Zatem, aby to wszystko, co poprzednio zostało napisane, logicznie się ze sobą łączyło, potrzebny nam będzie zapewne jakiś trik. Sztuka polega na właściwym wyborze funkcji H . Robin i Karp wybrali taką funkcję, która dzięki swym szczególnym właściwościom umożliwia dynamiczne wykorzystywanie wyników obliczeń dokonanych krok wcześniej, co znacząco potrafi uprościć obliczenia wykonywane w kroku bieżącym.

Założmy, że ciąg M znaków będziemy interpretować jako pewną liczbę całkowitą. Przyjmując za b — jako podstawę systemu — liczbę wszystkich możliwych znaków, otrzymamy:

$$x = t[i]b^{M-1} + t[i+1]b^{M-2} + \dots + t[i+M-1]$$

Przesuńmy się teraz w tekście o jedną pozycję do przodu i zobaczmy, jak zmieni się wartość x :

$$x' = t[i+1]b^{M-1} + t[i+2]b^{M-2} + \dots + t[i+M]$$

Jeśli dobrze przyjrzymy się x i x' , to okaże się, że wartość x' jest w dużej części zbudowana z elementów tworzących x — pomnożonych przez b z uwagi na przesunięcie. Nietrudno jest wówczas wywnioskować, że:

$$x' = (x - t[i]b^{M-1}) + t[i+M]$$

Jako funkcji H użyjemy dobrze nam znanej z poprzedniego rozdziału $H(x) = x \% p$, gdzie p jest dużą liczbą pierwszą. Założmy, że dla danej pozycji i wartość $H(x)$ jest nam znana. Po przesunięciu się w tekście o jedną pozycję w prawo pojawia się konieczność wyliczenia wartości funkcji $H(x')$ dla tego „nowego” słowa. Czy istotnie zachodzi potrzeba powtarzania całego wyliczenia? Być może istnieje pewne ułatwienie bazujące na zależności, jaka istnieje pomiędzy x i x' ?

Na pomoc przychodzi nam tu własność funkcji `modulo` użytej w wyrażeniu arytmetycznym. Można oczywiście obliczyć `modulo` z wyniku końcowego, lecz to bywa czasami niewygodne na przykład z uwagi na wielkość liczby, z którą mamy do czynienia — a poza tym gdzie tu byłby zysk szybkości?! Jednak identyczny wynik otrzymuje się, aplikując funkcję `modulo` po każdej operacji cząstkowej i przenosząc otrzymaną wartość do następnego wyrażenia cząstkowego! Dla przykładu weźmy obliczenie:

$$(5 \cdot 100 + 6 \cdot 10 + 8) \% 7 = 568 \% 7 = 1.$$

Wynik ten jest oczywiście prawdziwy, co można łatwo sprawdzić z kalkulatorem. Identyczny rezultat da nam jednak następująca sekwencja obliczeń:

$$(5 \cdot 100) \% 7 = 3 \quad (3 + 6 \cdot 10) \% 7 = 0 \quad (0 + 8) \% 7 = 1.$$

co jest też łatwe do weryfikacji.

Implementacja algorytmu jest prosta, lecz zawiera kilka instrukcji wartych omówienia. Spójrzmy na listing:



rk.cpp

```
const long p=33554393; // duża liczba pierwsza
const int b=64;        // duże + małe znaki + „coś jeszcze”
int rk(char w[], char t[])
{
```

```

unsigned long i, bM_1=1, Hw=0, Ht=0, M=strlen(w), N=strlen(t);
for(i=0; i<M; i++)
{
    Hw=(Hw*b+indeks(w[i]))%p; // inicjacja funkcji H dla wzorca
    Ht=(Ht*b+indeks(t[i]))%p; // inicjacja funkcji H dla tekstu
}
for(i=1; i<M; i++)
    bM_1=(b*bM_1)%p;
for(i=0; Hw!=Ht; i++) // przesuwanie się w tekście
{
    Ht=(Ht+b*p-indeks(t[i])*bM_1)%p;
    Ht=(Ht*b+indeks(t[i+M]))%p;
    if (i>N-M)
        return -1; // porażka poszukiwań
}
return i;
}

```

Na pierwszym etapie następuje wyliczenie początkowych wartości H_t i H_w . Ponieważ ciągi znaków trzeba interpretować jako liczby, konieczne będzie zastosowanie znanej nam już doskonałej funkcji `indeks` (patrz strona 186). Wartość H_w jest niezmienna i nie wymaga uaktualniania. Nie dotyczy to jednak aktualnie badanego fragmentu tekstu — tutaj wartość H_t ulega zmianie podczas każdej inkrementacji zmiennej i . Do obliczenia $H(x')$ możemy wykorzystać omówioną wcześniej własność funkcji `modulo` — co jest dokonywane w trzeciej pętli `for`. Dodatkowego wyjaśnienia wymaga być może linia oznaczona (*). Otóż dodawanie wartości $b \cdot p$ do H_t pozwala nam uniknąć przypadkowego wskoczenia w liczby ujemne. Gdyby istotnie tak się stało, przeniesiona do następnego wyrażenia arytmetycznego wartość `modulo` byłaby nieprawidłowa i sfałszowałaby końcowy wynik!

Kolejne uwagi należą się parametrom p i b . Zaleca się, aby p było dużą liczbą pierwszą¹⁴, jednakże nie można tu przesadzać z uwagi na możliwe przekroczenie zakresu pojemności użytych zmiennych. W przypadku wyboru dużego p zmniejszamy prawdopodobieństwo wystąpienia kolizji spowodowanej niejednoznacznością funkcji H . Ta możliwość — mimo iż mało prawdopodobna — ciągle istnieje i ostrożny programista powinien wykonać dodatkowy test zgodności $w[i \dots t[i]] \dots t[i+M-1]$ po zwróceniu przez funkcję `rk` pewnego indeksu i .

Co zaś się tyczy wyboru podstawy systemu (oznaczonej w programie jako b), to warto jest wybrać liczbę nawet nieco za dużą, zawsze jednak będącą potęgą liczby 2. Możliwe jest wówczas zaimplementowanie operacji mnożenia przez b jako przesunięcia bitowego — wykonywanego przez komputer znacznie szybciej niż zwykłe mnożenie. Przykładowo: dla $b = 64$ możemy zapisać mnożenie $b \cdot p$ jako $p << 6$.

Gwoli formalności można jeszcze dodać, że gdy nie występuje kolizja (typowy przypadek!), algorytm *Robina* i *Karpa* wykonuje się w czasie proporcjonalnym do $M+N$.

¹⁴ W naszym przypadku jest to liczba 33 554 393.

Rozdział 9.

Zaawansowane techniki programowania

Rozdziały poprzednie¹ dostarczyły nam interesujących narzędzi programistycznych. Zapoznaliśmy się z wieloma ciekawymi strukturami danych i przede wszystkim nauczyliśmy się posługiwać technikami rekurencyjnymi, stanowiącymi bazę nowoczesnego programowania. Zasadnicza rola rekurencji w procesie *koncepcji* programów nie była specjalnie eksponowana, koncentrowaliśmy się bowiem na próbach dokładnego zapoznania się z tym mechanizmem od strony technicznej.

W rozdziale niniejszym akcent położony na stosowanie rekurencji będzie o wiele silniejszy, gdyż większość prezentowanych w nim metod swoje istnienie zawdzięcza właśnie tej technice programowania.

Tematyka tego rozdziału jest nieco przewrotna i łatwo może nieuwważnego odbiorcę sprowadzić na manowce. Będziemy się bowiem zajmowali tzw. *technikami* (lub też inaczej: *metodami*) programowania, mającymi charakter niesłychanie ogólny i sugerującymi możliwość programowego rozwiązania niemal wszystkiego, co nam może tylko przyjść do głowy. Podawane algorytmy (a raczej ich wzorce) zostaną bowiem zilustrowane bardzo różnorodnymi zadaniami i, generalnie rzecz biorąc, będą dostarczać urzekająco efektywnych rezultatów. Co więcej, będzie się wręcz wydawać, że dostajemy do ręki uniwersalne recepty, które *automatycznie* spowodują zniknięcie wszelkich nierozwiązywalnych wcześniej zadań. Czytelnik domyśla się już zapewne, że bynajmniej nie będzie to prawdą. Złudzenie, któremu uleglibyśmy (gdyby nie niniejsze ostrzeżenie), wyniknie z dobrze dobranych przykładów, które wręcz wzorcowo będą pasować do aktualnie omawianej metody. W ogólnym jednak przypadku rzeczywistość będzie o wiele bardziej skomplikowana i próby stosowania tych technik programowania jako uniwersalnych „przepisów kucharskich” nie powiodą się. Czy ma to oznaczać, że owe metody są błędne? Oczywiście nie, tylko wszelkie usiłowania bezmyślnego ich zastosowania na pewno spalą na panewce, o ile nie dokonamy *adaptacji metody* do napotkanego problemu algorytmicznego.

Należy zdawać sobie bowiem sprawę z tego, iż każde nowe zadanie powinno być dla nas nowym wyzwaniem! Programista, dysponując pewną bazą wiedzy (nabyta teoria i praktyka), będzie z niej czynił odpowiedni pożytek, wiedząc jednak, że uniwersalne przepisy (w zasadzie) nie istnieją. Po algorytmice bowiem, jak i innych gałęziach wiedzy, nie należy spodziewać się cudów (chciałoby się dodać: niestety...).

¹ Szczególnie rozdział 2. poświęcony rekurencji i rozdział 5. poświęcony strukturom danych.

Programowanie typu „dziel i zwyciężaj”

Programowanie typu „dziel i zwyciężaj”² polega na wykorzystaniu podstawowej cechy rekurencji: dekompozycji problemu na pewną skończoną liczbę podproblemów tego samego typu, a następnie połączeniu w pewien sposób otrzymanych częściowych rozwiązań w celu odnalezienia rozwiązania globalnego. Jeśli oznaczymy problem do rozwiązania przez Pb , a rozmiar danych przez N , to zabieg wyżej opisany da się przedstawić za pomocą zapisu:

$$Pb(N) \rightarrow Pb(N_1) + Pb(N_2) + \dots + Pb(N_k) + KOMB(N).$$

Problem rzędu N został podzielony na k podproblemów.



Funkcja $KOMB(N)$ nie jest rekurencyjna.

Zasadniczo znak $+$ nie jest użyty powyżej w charakterze arytmetycznym, ale jeśli będziemy rozumować za pomocą czasów wykonania programu (patrz oznaczenia z rozdziału 3.), to wówczas \rightarrow możemy zamienić na znak równości i otrzymana *równość* będzie spełniona.

Powyższa uwaga ma fundamentalne znaczenie dla omawianej techniki programowania, bowiem podział problemu nie jest na ogół wykonywany w celach estetycznych (choć nie jest to oczywiście zabronione), ale ma za zadanie zwiększenie efektywności programu. Inaczej rzecz ujmując: chodzi nam o *przyspieszenie* algorytmu.

Technika „dziel i zwyciężaj” pozwala w wielu przypadkach na zmianę klasy algorytmu (np. z $O(n)$ do $O(\log N)$, etc.). Z drugiej jednak strony istnieje grupa zadań, dla których zastosowanie metody „dziel i zwyciężaj” nie spowoduje pożądanego przyspieszenia — z rozdziału 3. „Analiza złożoności algorytmów” wiemy, jak porównywać ze sobą algorytmy i przed zastosowaniem omawianej metody warto wziąć do ręki kartkę i ołówek, aby przekonać się, czy w ogóle opłaca się zasiadać do klawiatury!

Oto formalny zapis metody zaprezentowany za pomocą pseudojęzyka programowania:

```

dziel_i_rządź(N)
{
  jeśli N wystarczająco mały
    zwróć Przypadek_Elementarny(N);
  w przeciwnym przypadku
  {
    "Podziel" Pb(N) na k mniejszych egzemplarzy:
      Pb(N1), Pb(N2)... Pb(Nk);
    dla i=1... k
      Oblicz wynik cząstkowy wi=dziel_i_rządź(Ni);
    zwróć KOMB(w1, w2, ... wk);
  }
}

```

Określenie właściwego znaczenia sformułowań: „wystarczająco mały” „przypadek elementarny” będzie ściśle związane z rozważanym problemem i trudno tu podać dalej posuniętą generalizację. Ewentualne niejasności powinny się wyjaśnić podczas analizy przykładów znajdujących się w następnych podpunktach.

² Termin ten, rozpropagowany w literaturze angielskiej, niezbyt odpowiada idei Machiavelliego wyrażonej przez jego zdanie „Divide ut Regnes” (które ma niewątpliwą konotację destruktywną), ale wydaje się, że mało kto już na to zwraca uwagę.

Odszukiwanie minimum i maksimum w tablicy liczb

Z metodą „dziel i zwyciężaj” mieliśmy już w tej książce do czynienia w sposób niejawni i odnalezienie algorytmów, które mogą się do niej zakwalifikować, zostaje pozostawione Czytelnikowi jako test na spostrzegawczość i orientację (kilka odnośników zostanie jednak pod koniec podanych).

Jako pierwszy przykład przestudiujemy dość prosty problem odnajdywania elementów: *największego i najmniejszego* w tablicy. Problem raczej banalny, ale o pewnym znaczeniu praktycznym. Przykładowo: wyobraźmy sobie, że chcemy wykreślić na ekranie przebieg funkcji $y = f(x)$. W tym celu w pewnej tablicy zapamiętujemy obliczane N wartości tej funkcji dla przedziału, powiedzmy, $x_d \dots x_g$. Mając zestaw punktów, musimy przeprowadzić jego rzut na ekran komputera, tak aby zmieścić na nim tyle punktów, by otrzymany wykres nie przesunął się na niewidoczne na ekranie obszary. Z osią OX nie ma problemów: możemy się umówić, że x_d odpowiada współrzędnej 0, a x_g — odpowiednio maksymalnej rozdzielczości poziomej. Aby jednak przeprowadzić skalowanie na osi OY , konieczna jest znajomość ekstremalnych wartości funkcji $f(x)$. Dopiero wówczas możemy być pewni, że wykres istotnie zmieści się w strefie widocznej ekranu!

Ćwiczenie 1.

Proszę wyprowadzić wzory tłumaczące wartości (x_i, y_i) na współrzędne ekranowe (x_{ekr}, y_{ekr}) , jeśli znamy jednocześnie rozdzielczość ekranu graficznego X_{max} i Y_{max} oraz maksymalne odchylenia wartości funkcji $f(x)$, które oznaczmy jako f_{min} i f_{max} .

Powróćmy teraz do właściwego zadania. Pierwszy pomysł na zrealizowanie algorytmu poszukiwania minimum i maksimum pewnej tablicy³ polega na jej liniowym przeszukaniu:



min-max.cpp

```
const int n=12;
int tab[n]={23,12,1,-5,34,-7,45,2,88,-3,-9,1};
void min_max1(int t[], int& min, int& max)
{ // użyj tylko, gdy n >= 1!
  min=max=t[0];
  for(int i=1; i<n; i++)
  {
    if(max<t[i])           // (*)
      max=t[i];
    if(min>t[i])           // (**)
      min=t[i];
  }
}
```

Załóżmy, że tablica ma rozmiar n , tzn. obejmuje elementy od $t[0]$ do $t[n-1]$. Obliczmy złożoność obliczeniową praktyczną tego algorytmu, przyjmując za element czasochłonny instrukcje porównań. Wynik jest natychmiastowy: $T(n)=2(n-1)$, a zatem program jest klasy $O(n)$. Algorytm jest nieskomplikowany i... nieefektywny. Po bliższym przyjrzeniu się procedurze można bowiem zauważyć, że porównanie **(**)** jest zbędne, w przypadku gdy **(*)** jako pierwsze zakończyło się sukcesem. Dołożenie instrukcji **else** tuż po **(*)** spowoduje, że w najgorszym razie wykonamy $2(n-1)$ porównań, a w najlepszym — tylko $n-1$. Nie zmienia to oczywiście klasy algorytmu, ale ewentualnie go przyspieszy — w zależności, oczywiście, od konfiguracji danych wejściowych.

Zrealizujmy teraz analogiczną procedurę, wykorzystującą rekurencyjne uproszczenie algorytmu zgodnie z zasadą „dziel i zwyciężaj”. Idea jest następująca:

³ W przykładzie będzie to tablica liczb całkowitych, co nie zmniejsza stopnia ogólności algorytmu.

Przypadek ogólny:

- ◆ Jeśli tablica ma rozmiar równy 1, to zarówno `min`, jak i `max` są równe jednemu elementowi, który się w niej znajduje.
- ◆ Jeśli tablica ma dwa elementy, poprzez ich porównanie możemy z łatwością określić wartości `min` i `max`.

Przypadek ogólny:

- ◆ jeśli tablica ma rozmiar > 2 , to:
 - ◆ Podziel ją na dwie części.
 - ◆ Wylicz wartości (`min1`, `max1`) i (`min2`, `max2`) dla obu tych części.
 - ◆ Zwróć jako wynik `min=min(min1, min2)` i `max=max(max1, max2)`.

Odpowiadająca temu opisowi procedura rekurencyjna ma następującą postać:

```
void min_max2(int left, int right, int t[], int& min, int& max)
{
    if (left==right)
        min=max=t[left]; //jeden element
    else
        if (left==right-1) //dwa elementy
            if (t[left]<t[right])
            {
                min=t[left];
                max=t[right];
            }
            else
            {
                min=t[right];
                max=t[left];
            }
        else
        {
            int temp_min1, temp_max1, temp_min2, temp_max2, mid;
            mid=(left+right)/2;
            min_max2(left, mid, t, temp_min1, temp_max1);
            min_max2(mid+1, right, t, temp_min2, temp_max2);
            if (temp_min1<temp_min2) //(1)
                min=temp_min1;
            else
                min=temp_min2;
            if (temp_max1>temp_max2) //(2)
                max=temp_max1;
            else
                max=temp_max2;
        }
}
```

Porównując procedurę ze schematem ogólnym metody, można zauważyć, że:

- ◆ „Wystarczająco mały” oznacza rozmiar tablicy 1 lub 2.
- ◆ Mamy dwa przypadki elementarne.
- ◆ Dzielimy tablicę na 2 równe egzemplarze `N1` i `N2`.
- ◆ Wynikami cząstkowymi są pary: (`temp_min1`, `temp_max1`) oraz (`temp_min2`, `temp_max2`).
- ◆ Funkcja `KOMB` polega na najzwyczajszym porównywaniu wyników cząstkowych — jej rolę pełnią instrukcje oznaczone w komentarzach przez (1) i (2).

W dalszych przykładach nie będziemy już tak dokładnie „rozbierać” procedur, ufamy bowiem, że Czytelnik w miarę potrzeby bez trudu uczyni to samodzielnie.

Obliczenie złożoności obliczeniowej procedury `min_max2` także nie nastręcza trudności. Dekompozycja problemu jest następująca:

$$T(n) \rightarrow 2 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = 2 + 2T\left(\frac{n}{2}\right).$$

Jest to znany nam już rozkład logarytmiczny, po rozwiązaniu otrzymamy wynik $T(n) \in O(n)$ (patrz rozdział 3.).

Obliczenie złożoności praktycznej $T(n)$ tego programu nie należy do trudnych. Jeśli odpowiednio rozpiszemy równanie:

$$T(n) = 2 + 2T\left(\frac{n}{2}\right) = 2 + 2\left\{2 + T\left(\frac{n}{4}\right)\right\} = \text{etc.}$$

i założymy, że istnieje pewne k takie, że $n = 2^k$, to otrzymamy takie samo równanie, jak w przypadku procedury `min_max1`. Nie powinno to budzić zdziwienia, jeśli weźmiemy pod uwagę, że w drugiej wersji wykonujemy dokładnie taką samą pracę, jak w poprzedniej — postępując jednak w odmienny sposób.

Czy powyższy wynik nie jest czasem nieco niepokojący? Wygląda bowiem na to, że nowa metoda nie gwarantuje poprawy efektywności algorytmu!

Trafiliśmy już na zapowiedziany we wstępie przypadek problemu, dla którego zastosowanie metody „dziel i zwyciężaj” nie zmienia w istotny sposób parametrów czasowych programu. Cóż, można się przynajmniej łudzić, że ich nie pogarsza! Niestety w naszym przypadku nie jest to prawdą. Jeśli sięgniemy pamięcią do rozdziału poświęconego rekurencji i jej „ciemnym stronom”, powinno stać się dla nas jasne, że z uwagi na wprowadzenie dodatkowego obciążenia pamięci (stos wywołań rekurencyjnych) i niepomijalnej liczby dodatkowych wywołań rekurencyjnych zakładana równoważność czasowa obu procedur nie jest prawdą.

Przedstawiony powyżej przykład nie jest prawdopodobnie najlepszą reklamą omawianej metody — miał on jednak na celu ukazanie potencjalnych zagrożeń związanych z naiwną wiarą w cudowne metody. Są oczywiście przypadki, w których „dziel i zwyciężaj” czyni wręcz cuda (już zaraz kilka z nich zresztą zaprezentuję), ale o ich zaistnieniu można się jedynie przekonać, wyliczając złożoność obliczeniową obu metod. Jeśli w istocie otrzymamy znaczący zysk szybkości — na przykład zmianę klasy programu na lepszą — to jest mało prawdopodobne, aby pewne niekorzystne cechy rekurencji grały istotną rolę. W przypadku jednak otrzymania wyniku dowodzącego równoważność czasową metod, trzeba również wziąć pod uwagę wskaźniki, które nie mając nic wspólnego z teorią, odgrywają niebagatelną rolę w praktyce (w tzw. rzeczywistym świecie). Pewne uwagi trzeba bowiem wypowiedzieć co najmniej raz, aby później nie denerwować się, że komputer nie chce robić tego, co my mu każemy (lub robi to gorzej, niż chcielibyśmy).

Mnożenie macierzy o rozmiarze $N \times N$

W wielu zagadnieniach natury numerycznej często zachodzi potrzeba mnożenia ze sobą macierzy, co z definicji jest dość czasochłonną operacją. Sposób wyliczania iloczynu dwóch macierzy może być symbolicznie przedstawiony w sposób zaprezentowany na rysunku 9.1.

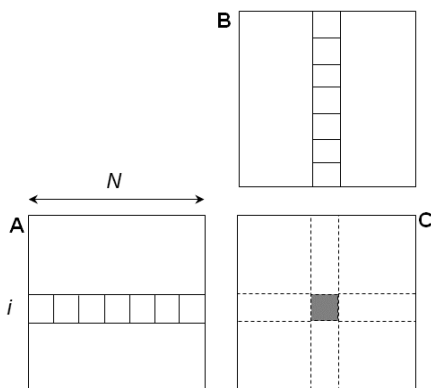
Jeśli macierz C (przypomnijmy, że z punktu widzenia programisty macierz jest tablicą dwuwymiarową) będziemy uważać za wynik mnożenia $A \times B$, to dowolny element $C[i,j]$ można otrzymać, stosując wzór:

$$C_{i,j} = \sum_{k=1}^N A_{i,k} B_{k,j}.$$

(Mnożymy odpowiadające sobie elementy linii i i kolumny j , kumulując jednocześnie sumy cząstkowe).

Rysunek 9.1.

Mnożenie macierzy



Koszt wyliczenia jednego elementu macierzy C , a mieć musimy na uwadze liczbę wykonywanych operacji mnożenia (przyjmijmy rozsądnie, że to one są tu najkosztowniejsze), jest oczywiście równy N . Ponieważ wszystkich elementów jest N^2 , to koszt całkowity wyniesie N^3 , czyli algorytm należy do $O(N)^3$.

Algorytm jest bardzo kosztowny, ale wydawało się to przez długi czas tak nieuniknione, że praktycznie wszyscy się z tym pogodzili. W roku 1968 Volker Strassen sprawił jednakże wszystkim sporą niespodziankę, wynajdując algorytm bazujący na idei „dziel i zwyciężaj”, który był lepszy niż, wydawałoby się „nienaruszalne”, $O(N)^3$.

Oznaczmy elementy macierzy A , B i C w sposób następujący:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Nie jest trudno wykazać, że prawdziwe są następujące równości:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21},$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Podejście polegające na podziale każdej z matryc A i B na 4 równe części (zakładając oczywiście, że N jest potęgą liczby 2) i wykonanie mnożenia matryc mniejszego rzędu wydaje się bezpośrednim zastosowaniem techniki „dziel i zwyciężaj”. Ponieważ jednak podział nie oszczędza nam pracy (w dalszym ciągu jesteśmy zmuszeni do zrobienia dokładnie tego samego, co algorytm iteracyjny), to na pewno nie otrzymamy tu efektywniejszego algorytmu. Stwierdzenie to nie jest poparte obliczeniami, ale zapewniam, że jest prawdziwe.

Spójrzmy teraz jak V. Strassen zoptymalizował mnożenie macierzy. Zasadnicza idea jego metody polega na wprowadzeniu dodatkowych zmiennych, będących macierzami rzędu $\frac{N}{2}$: P, Q, R, S, T, U, V , służących do zapamiętania wyników następujących obliczeń⁴:

$$P = (A_{11} + A_{22})(B_{11} + B_{22}) \quad *, +, +$$

$$Q = (A_{21} + A_{22})B_{11} \quad +, *$$

$$R = A_{11}(B_{12} - B_{22}) \quad *, -$$

⁴ Obok równań są zaznaczone operacje arytmetyczne wymagane do wyliczenia danej równości.

$$\begin{aligned}
S &= A_{22}(B_{21}-B_{11}) & *, - \\
T &= B_{22}(A_{11}+A_{12}) & *, + \\
U &= (A_{21}-A_{11})(B_{11}+B_{12}) & *, +, - \\
V &= (A_{12}-A_{22})(B_{21}+B_{22}) & *, +, -
\end{aligned}$$

Gdy mamy te cząstkowe wyniki, otrzymanie macierzy C może być dokonane poprzez następujące podstawienia:

$$\begin{aligned}
C_{11} &= P + S - T + V & +, -, + \\
C_{12} &= R + T & + \\
C_{21} &= Q + S & + \\
C_{22} &= P + R - Q + U & +, -, +
\end{aligned}$$

Algorytm tej postaci wymaga 7 operacji mnożenia i dodatkowych 18 operacji dodawania lub odejmowania.

Zauważmy, że algorytm *Strassena* przenosi w inteligentny sposób ciężar obliczeń z zawsze kosztownej operacji mnożenia⁵ na znacznie szybsze dodawanie lub odejmowanie.

Rozkład rekurencyjny w algorytmie *V. Strassena* jest następujący:

$$T(n) = 7T\left(\frac{N}{2}\right) + cN^2 \in O(N^{2.81})$$

(c jest pewną stałą).

Blizsze badania praktyczne tego algorytmu wykazały, że realny zysk powyższej metody daje się zauważyć w przypadku mnożenia macierzy dla N rzędu kilkudziesięciu, ale w przypadku naprawdę dużych N (np. powyżej 100) efektywność algorytmu ponownie zbliża się do swojego iteracyjnego konkurenta. Fenomen ten zależy m.in. od sposobu zarządzania pamięcią w danym środowisku sprzętowym. Jeśli mamy do czynienia z komputerem osobistym o dużym „prywatnym” zasobie pamięci, to działanie algorytmu będzie zbliżone do przewidywań teoretycznych. Jednak w przypadku systemów rozproszonych, w których program — widząc pozornie całą żadaną pamięć — faktycznie operuje jej „stronami” dosyłanymi mu w miarę potrzeb przez system operacyjny, sprawa może wyglądać już trochę gorzej. Drugim istotnym powodem spadku efektywności algorytmu dla dużych (praktycznie występujących) wartości N jest kumulacja wielokrotnych wywołań rekurencyjnych i wzrastającej roli operacji dodawania i odejmowania. Z wymienionych wyżej względów algorytm *V. Strassena* należy raczej traktować jako ciekawy wynik teoretyczny o niepodważalnych walorach edukacyjnych!

Mnożenie liczb całkowitych

Kolejny przykład jest również natury obliczeniowej: zajmiemy się mnożeniem liczb całkowitych.

Mnożenie dwóch liczb całkowitych X i Y , których reprezentacja wewnętrzna ma rozmiar N -bitów, jest operacją klasy $O(N^2)$. Zakładamy, że mnożenie jest wykonywane klasycznie, tak jak nas tego nauczono w szkole podstawowej (sumujemy w słupku N wyników iloczynów cząstkowych, każdy z nich jest klasy $O(N)$).

⁵ Zwłaszcza jeśli elementami macierzy są liczby rzeczywiste (typ *float* i *double* w C++).

Metoda „dziel i zwyciężaj” w przypadku mnożenia liczb całkowitych może być zastosowana po dokonaniu następującej obserwacji:

$$X = [A \ B] = A \cdot 2^{\frac{N}{2}} + B,$$

$$Y = [C \ D] = C \cdot 2^{\frac{N}{2}} + D.$$

A i B oraz C i D oznaczają odpowiednio połówki reprezentacji binarnych liczb X i Y . Iloczyn $X \cdot Y$ może być zapisany jako:

$$X \cdot Y = AC \cdot 2^{\frac{N}{2}} + (AD + BC) \cdot 2^{\frac{N}{2}} + BD.$$

Jeśli założymy, że N jest potęgą liczby 2 (co jest generalnie prawdą we współczesnych komputerach), to możemy wyrazić złożoność obliczeniową programu przez:

$$T(1) = 1,$$

$$T(N) = 4T\left(\frac{N}{2}\right) + cN.$$

W równaniach tych zaznaczamy wpływ czterech kosztownych operacji mnożenia plus pewien proporcjonalny do N koszt związany z dodawaniami i przesunięciami bitowymi⁶. Aby wyliczyć klasę tego algorytmu, można sięgnąć do wzorów podanych w rozdziale 3. albo nie wysilać się zbytnio i dojść do wniosku, że... mamy do czynienia z $O(N^2)$.

Skąd ta pewność? Pochodzi ona z obserwacji wynikłej podczas analizy procedury `min_max`: dokonaliśmy podziału problemu, ale w niczym nie zmniejszyliśmy ilości wykonywanej pracy. Cudów zatem nie będzie⁷! Zanim jednak rozczarujemy się na dobrą metodą „dziel i zwyciężaj”, popatrzymy na następujące przepisanie operacji $X \cdot Y$ w nieco inny sposób niż poprzednio:

$$X \cdot Y = AC \cdot 2^{\frac{N}{2}} + [(A - B)(D - C) + AC + BD]2^{\frac{N}{2}} + BD.$$

Mimo nieco bardziej skomplikowanej postaci (patrz algorytm *Strassen*!) zmniejszyliśmy liczbę operacji mnożenia z 4 na 3 (AC i BD występują podwójnie, zatem za drugim użyciem można skorzystać z poprzedniego wyniku). Formuła rekurencyjna towarzysząca temu rozkładowi jest identyczna jak w przypadku poprzednim, wystarczy tylko zamienić 4 na 3. Wiedząc to, otrzymujemy natychmiast, że algorytm jest klasy $O(N^{\log_2 3}) = O(N^{1.59})$.

Zachęca się Czytelnika do zbadania na różnych przykładach i przy użyciu różnorodnych założeń dotyczących kosztów operacji elementarnych (+, −, przesunięcie bitowe), kiedy istotnie ten algorytm może dać zauważalne rezultaty w porównaniu z metodą klasyczną.

Inne znane algorytmy „dziel i zwyciężaj”

Mimo iż nie było to specjalnie podkreślane, już w rozdziałach poprzednich mogliśmy zapoznać się z kilkoma ciekawymi algorytmami, które można zaklasyfikować do metody „dziel i zwyciężaj”.

⁶ Przypomnijmy, że mnożenie liczby przez potęgę podstawy systemu ($2^1, 2^2, 2^3 \dots$) jest równoważne przesunięciu jej reprezentacji wewnętrznej o wykładnik potęgi miejsc w lewo (1, 2, 3...).

⁷ Dla niedowiarków prezentuję dowód matematyczny: $T(n) \in O(N^{\log_2 4}) = O(N^2)$.

Programem, który zdecydowanie króluje wśród nich, jest niewątpliwie słynny *Quicksort* (patrz rozdział 4.). Oferuje on znaczący wzrost szybkości sortowania i, co najważniejsze, jest przy tym niesłychanie prosty zarówno w zapisie, jak i ideowo.

Omówiony przy okazji rozpatrywania technik derekusuacji problem wież *Hanoi* (patrz rozdział 6.) jest również dobrym przykładem inteligentnej dekompozycji problemu. Mimo iż wersje iteracyjne i rekurencyjne są tej samej klasy, to prostota zapisu rekurencyjnego jest najlepszym argumentem za jego zastosowaniem.

Procedura *przeszukiwania binarnego* również może być zaklasyfikowana do metody „dziel i zwyciężaj”, choć różni się nieco filozofią od podanego wcześniej w tym rozdziale schematu. Jest ona dobrym przykładem na to, jak dobry algorytm może przyspieszyć rozwiązanie postawionego problemu, dla którego znana jest prosta, ale nieefektywna metoda (w tym przypadku jest nią przeszukiwanie liniowe).

Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas...

Nazwa nowej metody jest bardzo intrygująca, ale w literaturze przedmiotu przyjęło się nazywać pewną klasę metod jako „żarłoczne” lub „zachłanne” (ang. *greedy*). Algorytmy te służą do odnajdywania rozwiązań, które mają zastosowanie w odszukiwaniu *przepisu* na rozwiązanie danego problemu. Przepis ten jest obarczony pewnymi założeniami (ograniczeniami), które mogą na przykład żądać podania rozwiązania optymalnego wg pewnych kryteriów. Chcąc skonstruować ów przepis, mamy do czynienia z szeregiem opcji tworzących zbiór danych wejściowych. Cechą szczególną algorytmu „żarłocznego” jest to, że na każdym etapie poszukiwania rozwiązania wybiera on opcję *lokalnie* optymalną. W zależności od tego doboru rozwiązanie globalne może być również optymalne, ale nie jest to gwarantowane. Omawiana metoda najlepiej odpowiada pewnej klasie zadań natury optymalizacyjnej: podać najkrótszą drogę w grafie, określić optymalną kolejność wykonywania pewnych zadań przez komputer, etc.

Metoda algorytmów „żarłocznych” odpowiada ludzkiej naturze, gdyż bardzo często, otrzymując jakieś zadanie, zadowalamy się rozwiązaniem niekoniecznie optymalnym, ale za to szybko otrzymanym.

Schemat generalny algorytmu jest następujący:

```

żarłok(W)
{
    ROZW=∅; //zbiór pusty
    dopóki(nie Znalezione(ROZW) i W≠∅)
        wykonuj:
        {
            X=Wybór(W);
            jeśli Odpowiada(X) to ROZW = ROZW ∪ {X};
        }
    jeśli Znalezione(ROZW)
        zwróć ROZW;
    w przeciwnym przypadku:
        zwróć "nie ma rozwiązania"
}

```

W opisie metody zostały użyte następujące oznaczenia:

W — zbiór danych wejściowych.

ROZW — zbiór, na podstawie którego będzie konstruowane rozwiązanie.

X — element zbioru.

Wybór(A) — funkcja dokonująca „optymalnego” wyboru elementu ze zbioru A (usuwając go z niego).

Odpowiada(X) — czy wybierając X, można tak skompletować rozwiązanie cząstkowe, aby odnaleźć co najmniej jedno rozwiązanie globalne?

Znaleziono(R) — czy R jest rozwiązaniem zadania?

Powyższy zapis wyjaśnia nazwę metody: na każdym etapie dobieramy najlepszy kasek, nie troszcząc się specjalnie o przyszłość. Popatrzmy na kilka przykładów zastosowania nowej metody.

Problem plecakowy, czyli niełatwe jest życie turysty piechura

Wczujmy się teraz w rolę turysty wybierającego się na dłuższą pieszą wycieczkę po górach. Aby urealnić przykład, niech naszym zadaniem będzie dotarcie na szczyt pewnej góry w Pirenejach, gdzie znajduje się punkt zbiorczy, który nasi wspólni znajomi wybrali na zorganizowanie „przyjęcia” na łonie natury. Do punktu docelowego zmierza w sumie pięć osób — każda z nich zobowiązała się dostarczyć imponującą ilość wiktuałów, tak aby umówioną imprezę uczynić isticie królewską ucztą. Nie będziemy wnikać w zbędne szczegóły, usiłując odgadnąć, co niosą ze sobą pozostałe cztery osoby, zajmiemy się jedynie naszym prywatnym problemem, który napotkaliśmy, przygotowując wyprawę. Założmy, że zostaliśmy obarczeni zadaniem dostarczenia kilku gatunków dobrych serów i niespecjalnie wiemy, jak upakować je w wolnej przestrzeni plecaka.

Nasz plecak posiada gwarantowaną przez producenta pojemność 60 litrów, z czego zostało nam $M = 20$ litrów na część kulinarną. Reszta już jest wypełniona niezbędnymi do przeżycia w górach elementami, pozostał nam jedynie dylemat optymalnego wypełnienia reszty plecaka. Chcemy wziąć w sumie trzy gatunki sera (s_1 , s_2 i s_3). W domowej lodówce owe sery znajdują się w ilościach w_1 , w_2 i w_3 litrów. Każdy z serów jest doskonały, niemniej możemy im przypisać orientacyjne ceny c_1 , c_2 i c_3 , które pozwalają ustawić je w swoistym rankingu jakości.

Naszym celem jest wzięcie z każdego gatunku sera takiej jego ilości:

$$0 \leq x_1, x_2, x_3 \leq 1,$$

aby w sumie nie przekroczyć maksymalnej pojemności części plecaka przeznaczonej na sery:

$$\sum_{i=1}^3 w_i x_i \leq M$$

oraz zabrać jak najwartościowszy ładunek, tzn. *zmaksymalizować* funkcję $\sum_{i=1}^3 c_i x_i$.

Aby rozważania uczynić mniej teoretycznymi, popatrzmy na konkretny przykład kilku konfiguracji danych:

$$w_1 = 16, \quad w_2 = 12, \quad w_3 = 10,$$

$$c_1 = 80, \quad c_2 = 70, \quad c_3 = 60.$$

Kilka przykładów zamieszczonych w tabeli 9.1 ilustruje różnorodność potencjalnych rozwiązań w przypadku nietrywialnej konfiguracji danych wejściowych (taka wystąpiłaby, gdyby suma w_i była mniejsza od M — Czytelnik z łatwością odgadnie właściwe rozwiązanie w przypadku takiej sytuacji).

Chwilowo spośród trzech wymyślonych ad hoc rozwiązań optymalnym jest drugie, nie nam jednak nie gwarantuje, że nie istnieją lepsze konfiguracje parametrów x_i .

Trzeba być może w tym miejscu podkreślić, że podstawowa idea, na której została zbudowana procedura żarłok, nie gwarantuje optymalności. Wręcz przeciwnie, w typowym przypadku otrzymane rozwiązanie⁸ będzie tylko *prawie* optymalne!

⁸ Jeśli oczywiście istnieje!

Tabela 9.1. Przykładowe rozwiązania problemu plecakowego

Lp.	(x_1, x_2, x_3)	$\sum_{i=1}^3 c_i x_i$	$\sum_{i=1}^3 w_i x_i$
1	$\left(\frac{1}{1}, \frac{1}{4}, \frac{1}{10}\right)$	103,5	20
2	$\left(\frac{2}{3}, \frac{1}{2}, \frac{1}{3}\right)$	108,3	20
3	$\left(\frac{2}{3}, \frac{1}{5}, \frac{2}{3}\right)$	107,3	19,7

Przy takim postawieniu sprawy można dość szybko zniechęcić się do omawianej metody... o ile nie przypomnimy sobie uwagi zawartej na samym wstępie tego rozdziału: problemy, które będziemy chcieli rozwiązywać, mogą wymusić adaptację omawianych metaalgorytmów, każda próba bezmyślnego ich stosowania spali (prawdopodobnie) na panewce.

Aby to dokładniej zilustrować, przeanalizujemy kilka możliwych strategii rozwiązania problemu plecakowego przy użyciu algorytmu „żarłocznego”. Pierwsze, pozornie optymalne, rozwiązanie polega na próbach wypełniania plecaka za pomocą najdroższego sera (s_i): jeśli jego całkowita objętość mieści się w wolnej przestrzeni, to bierzemy go w całości, w przypadku przeciwnym — ucinamy taki jego kawałek, aby nie przekroczyć objętości M i zużyć możliwie największy kawałek tego sera. Następnie zajmujemy się w sposób analogiczny kolejnym w rankingu cen serem itd.

Cóż, wystarczy przetestować „ręcznie” kilka konfiguracji otrzymanych za pomocą tej metody, aby się przekonać, że nie daje ona najlepszych rezultatów. Najlepszym przykładem może tu być analiza tabeli 9.1, zwłaszcza pozycji 1. i 2.

Przyczyna nieoptymalności rozwiązania jest relatywnie prosta: efekt końcowy (funkcja, którą chcemy zmaksymalizować) zależy nie tylko od aktualnej wartości wkładanych serów, ale i od ich objętości. Może zatem należy patrzeć w pierwszej kolejności nie na parametr c_i , ale na w_i ?

Kilka prób dokonanych z ołówkiem w rękę prowadzi nas jednak do niezbyt zachęcających rezultatów także i w tym przypadku. I znowu możemy zwątpić w sens metody...

Jeśli obie analizowane skrajności nie prowadzą do optymalnego rozwiązania, to jedyne, co nam pozostaje, to zmienić strategię postępowania w taki sposób, aby obiektywnie uwzględniała oba parametry (w_i, c_i) *jednocześnie*. Okazuje się, że jeśli wstępnie poustawiamy dane wejściowe w taki sposób, aby dla dowolnego i zachować stosunek:

$$\frac{c_{i+1}}{w_{i+1}} \leq \frac{c_i}{w_i},$$

to algorytm „żarłoczny” prowadzi do rozwiązania optymalnego. Aby nie nasycać tego podręcznika zbędną porcją matematyki, dowód powyższego twierdzenia sobie darujemy, gdyż nie jest on istotny.

Popatrzmy na program w C++, który rozwiązuje nasz dylemat plecakowy:



greedy.cpp

```
const int n=3;
void greedy(double M, double W[n], double C[n], double X[n])
{
    int i;
    double Z=M; // pozostaje do wypełnienia
    for(i=0; i<n; i++)
    {
        if(W[i]>Z)
```

```

break;
    X[i]=1;
    Z=Z-W[i];
}
if(i<n)
    X[i]=Z/W[i];
}
int main()
{
double W[n]={10,12,16}, C[n]={60,70,80}, X[n]={0,0,0};
greedy(20,W,C,X);
double p=0;
for(int i=0; i<n; p+=X[i]*C[i],i++)
    cout << i << "\t" << W[i] << "\t" << C[i] << "\t" << X[i] << endl;
cout << "Razem:"<<p<<endl;
}

```

Okazuje się, że rozwiązaniem optymalnym jest wektor $X = \left(1, \frac{5}{6}, 0\right)$ — w takiej kolejności danych, w jakiej są zamieszczone na listingu, gdzie nastąpiła już wstępna „obróbka” wg zacytowanego wcześniej wzoru.

Wniosek z analizy problemu plecakowego powinien być dla Czytelnika następujący: przed przystąpieniem do kodowania programu w naszym ulubionym języku programowania (niekoniecznie w C++) warto poświęcić kilka minut na refleksję, co może znakomicie zwiększyć jakość otrzymanego rozwiązania końcowego.

Programowanie dynamiczne

Zalety programowania rekurencyjnego uwidaczniają się w prostocie i naturalności formułowania rozwiązań. Niestety rekurencja ma swoje drugie oblicze, o którym łatwo zapomnieć, rozważając ją w kategoriach czysto matematycznych. Chodzi oczywiście o to, jak naprawdę formuła rekurencyjna zostanie wykonana przez komputer, ile będzie kosztowało zrealizowanie wywołań rekurencyjnych, powrotów z nich, kombinowanie rezultatów cząstkowych, etc.

Może się zatem okazać, że formalnie szybki algorytm rekurencyjny (rozumując w kategoriach klasy O) będzie znacznie wolniejszy, niż to wynika z obliczeń teoretycznych.

Sposobów na zaradzenie temu zjawisku jest kilka (patrz np. rozdział 6.), między innymi jest wśród nich... pisanie tylko procedur iteracyjnych!

Wprowadzanie rewolucji w programowaniu w postaci powszechnego zakazu stosowania rekurencji nie jest bynajmniej celem tej książki. Postawmy zatem problem inaczej: *czy jest możliwe spożytkowanie korzyści płynących z rekurencyjnego formułowania rozwiązań, ale bez używania rekurencji?*

Wbrew pozorom nie jest to paradoks — technika *programowania dynamicznego* bazuje właśnie na tym — zdawałoby się niemożliwym do zrealizowania — postulatcie. Jej wynalazca, Richard E. Bellman został uhonorowany w 1979 nagrodą IEEE Medal of Honor za jej odkrycie (ponad 20 lat po fakcie!).

Technika nadaje się szczególnie dobrze do rozwiązywania problemów o charakterze numerycznym:

- ♦ obliczanie najkrótszej drogi w grafach (grafy poznamy szczegółowo w rozdziale 10.);
- ♦ wyliczenie pewnej skomplikowanej wartości podanej za pomocą równania rekurencyjnego.

W wersji oryginalnej (podanej przez R. Bellmana) metoda programowania dynamicznego opiera się na zasadzie zwanej zasadą optymalności. Aby nie zaciemniać wywodu, pominię ją tutaj, podając wyłącznie interpretację algorytmiczną.

Konstrukcja programu wykorzystującego zasadę programowania dynamicznego może być sformułowana w trzech etapach:

♦ **Koncepcja:**

- ♦ Dla danego problemu P stwórz rekurencyjny model jego rozwiązania (wraz z jednoznacznym określeniem przypadków elementarnych).
- ♦ Stwórz tablicę, w której będzie można zapamiętywać rozwiązania przypadków elementarnych i rozwiązania podproblemów, które zostaną obliczone na ich podstawie.

♦ **Inicjacja:**

- ♦ Wpisz do tablicy wartości numeryczne, odpowiadające przypadkom elementarnym.

♦ **Progresja:**

- ♦ Na podstawie wartości numerycznych wpisanych do tablicy, używając formuły rekurencyjnej, oblicz rozwiązanie problemu wyższego rzędu i wpisz je do tablicy.
- ♦ Postępuj w ten sposób do momentu osiągnięcia pożądanej wartości.

Być może powyższy zapis brzmi enigmatycznie, ale jak to wyniknie z dalszych przykładów, metoda jest naprawdę nieskomplikowana. Zanim jednak przejdziemy do ilustracji tej techniki programowania, porównajmy ją z wcześniej poznaną metodą „dziel i zwyciężaj”.

„dziel i zwyciężaj”

- ♦ Problem rzędu N rozłóż na podproblemy niższego rzędu i rozwiąż je.
- ♦ Połącz rozwiązania podproblemów w celu otrzymania rozwiązania globalnego.

„programowanie dynamiczne”

- ♦ Mając dane rozwiązanie problemu elementarnego, wylicz na jego podstawie problem wyższego rzędu.
- ♦ Kontynuuj obliczenia aż do momentu otrzymania rozwiązania rzędu N .

Nowa technika ma pewien posmak optymalności: raz znalezione rozwiązanie pewnego podproblemu zostaje zarejestrowane w tablicy i w miarę potrzeb jest później wykorzystywane. Nie był to bynajmniej przypadek metody „dziel i zwyciężaj”, która pozwalała na wielokrotne wyliczanie tych samych wartości.

Nowo poznaną metodę zilustrujemy kilkoma przykładami o różnym stopniu skomplikowania, zaczynając od... doskonale nam znanego problemu obliczania elementów ciągu Fibonacciego (patrz rozdział 2.).

Ciąg Fibonacciego

Przypomnimy (po raz kolejny) definicję ciągu Fibonacciego:

$$\begin{aligned} fib(0) &= 0, \\ fib(1) &= 1, \\ fib(n) &= fib(n-1) + fib(n-2) \quad \text{gdzie } n \geq 2 \end{aligned}$$

Rozwiązanie rekurencyjne testowaliśmy już kilkakrotnie, spróbujmy teraz zaadoptować rekurencyjną procedurę obliczania tego ciągu do podanych powyżej zasad konstrukcji programu, wykorzystujących programowanie dynamiczne:

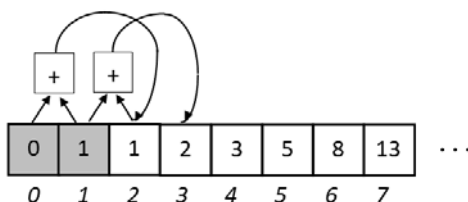
- ◆ *Koncepcja* — wzór rekurencyjny już mamy, pozostaje tylko zadeklarować tablicę $fib[n]$ do składowania obliczanych wartości.
- ◆ *Inicjacja* — początkowymi wartościami w tablicy fib będą oczywiście warunki początkowe: $fib[0] = 0$ i $fib[1] = 1$.
- ◆ *Progresja algorytmu* — ten punkt zależy ściśle od wzoru rekurencyjnego, który implementujemy za pomocą tablicy. W naszym przypadku wartością $fib[i]$ w tablicy (dla $i \leq 2$) jest suma dwóch poprzednio obliczonych wartości: $fib[i-1]$ i $fib[i-2]$. Obie te wartości zostały zapamiętane w tablicy, zupełnie jak w programie rekurencyjnym, który zapamiętuje je... na stosie wywołań rekurencyjnych.

Zauważmy jednak, że tej analogii nie można posunąć zbyt daleko, bowiem nasze postępowanie ma charakter sekwencji instrukcji elementarnych bez dodatkowych wywołań proceduralnych, tak jak to czyni każdy program rekurencyjny.

Powyższe uwagi są zilustrowane na rysunku 9.2.

Rysunek 9.2.

Obliczanie wartości
elementów ciągu
Fibonacciego



Zupełnie już dla formalności podam procedurę, która realizuje omówione uprzednio obliczenia:



fib-dyn.cpp

```
void fib_dyn(int x, int f[])
{
    f[0]=0;
    f[1]=1;
    for(int i=2; i<x; i++)
        f[i]=f[i-1]+f[i-2];
}
```

Równania z wieloma zmiennymi

Nieco bardziej skomplikowana sytuacja występuje w przypadku równań rekurencyjnych posiadających więcej niż jedną zmienną. Popatrzmy na następujący wzór:

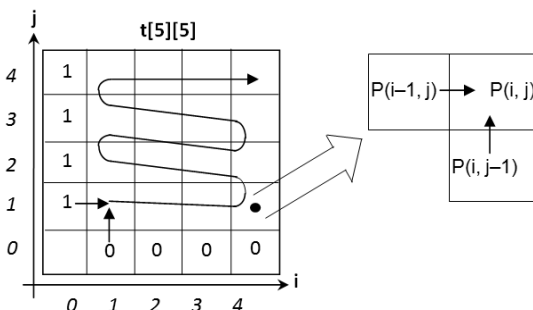
$$P(i, j) = \begin{cases} 1 & \text{dla } i = 0 \text{ oraz } j > 0, \\ 0 & \text{dla } i > 0 \text{ oraz } j = 0, \\ \frac{P(i-1, j) + P(i, j-1)}{2} & \text{dla } i > 0 \text{ oraz } j > 0. \end{cases}$$

Mamy tu do czynienia z dwiema zmiennymi, i oraz j . Interesuje nas obliczenie wartości parametru P . Powyższy wzór jest dość nieprzyjemny już na pierwszy rzut oka — można również udowodnić, że jest bardzo kosztowny, jeśli chodzi o czas obliczeń. Mamy zatem doskonały przykład dowodzący, że jeśli nie musimy stosować rekurencji, to najlepiej byłoby tego w ogóle nie czynić... pod warunkiem że posiadamy alternatywne drogi rozwiązania.

Technika programowania dynamicznego taką drogę podpowiada. Sposób obliczenia wzoru rekurencyjnego jest trywialny, jeśli wpadniemy na pomysł użycia tablicy *dwuwymiarowej*, której współrzędne pozioma i i pionowa będą odpowiadać zmiennym i oraz j . Popatrzmy na rysunek 9.3 przedstawiający ogólną ideę programu obliczającego wartości $P(i, j)$.

Z uwagi na specyfikę problemu wygodnie będzie zainicjować tablicę już na samym wstępie warunkami początkowymi (zera i jedynki w odpowiednich miejscach), chociaż w zoptymalizowanej wersji można by tę część wbudować w pętlę główną programu. Do obliczenia wartości $P(i, j)$ potrzebna jest znajomość dwóch sąsiednich komórek: dolnej — $P(i, j-1)$ oraz tej znajdującej się z lewej strony — $P(i-1, j)$. Uwaga ta prowadzi nas do spostrzeżenia, że naturalnym sposobem obliczania wartości $P(i, j)$ będzie posuwanie się zygzakiem zaznaczonym na rysunku 9.3.

Rysunek 9.3.
Dwuwymiarowy
wzór rekurencyjny



Gdy mamy te wszystkie informacje, realizacja programowa jest natychmiastowa:



dyn.cpp

```
const int n=5;
void dynam(double P[n][n])
{
    int i,j;
    for(i=1;i<n;i++)
    {
        P[i][0]=0;
        P[0][i]=1;
    }
    for(j=1;j<n;j++)
        for(i=1;i<n;i++)
            P[i][j]=(P[i-1][j]+P[i][j-1])/2.0;
}
```

Nietrudno jest zauważyć, że program powyższy jest dokładnym odbiciem wzoru rekurencyjnego — jedyny nasz wysiłek polega w zasadzie na tym, żeby znaleźć *prawidłowy* sposób wypełniania tablicy. Celowo podkreślam, że *prawidłowy*, bowiem w przypadku problemów dwu- i więcej wymiarowych (jeśli możemy pozwolić sobie na takie określenie nawiązujące do wymiarów tablicy) możemy bardzo łatwo popełnić błąd polegający na próbie wykorzystania wartości z tablicy, które na danym etapie nie są jeszcze obliczone. Tego typu potknięcia są czasami bardzo trudne do wykrycia, więc należy przy tym szczególnie uważać.

Najdłuższa wspólna podsekwencja

Programowanie dynamiczne dobrze nadaje się do rozwiązywania problemów algorytmicznych dotyczących przeszukiwania ciągów znaków. Powód jest oczywisty: klasyczne reprezentacje ciągów znaków zakładają, że mamy do czynienia z tablicami wypełnionymi kodami ASCII (lub dowolnymi innymi).

Porównywanie sekwencji kodów znajduje zastosowanie w wielu obszarach, np.:

- ◆ porównywanie plików i tekstów;
- ◆ określanie „odległości” dzielącej ciągu znaków (ile trzeba wykonać przekształceń, aby przeobrazić jeden ciąg w drugi – jest to tzw. odległość Levenshteina);
- ◆ umiejętność klasyfikowania i porównywania przydaje się np. w analizie kodów DNA (wyszukiwanie podobieństw w łańcuchu dziedziczenia).

Spróbujmy zająć się klasycznym problemem odszukania najdłuższej wspólnej podsekwencji dwóch (ang. *longest common subsequence*) ciągów X i Y . Algorytmu, który postaramy się wspólnie „wynaleźć”, nie należy mylić z klasycznymi algorytmami opisanymi w poprzednim rozdziale, gdyż sekwencje, o których mowa, mogą być nieciągłe.

Zamodelujmy rekurencyjnie nasz problem. Celem ćwiczenia jest znalezienie pewnej nieznannej (na razie) funkcji $LCS(X_i, Y_j)$, która zwróci najdłuższą wspólną podsekwencję dla ciągów X i Y . Weźmy dla przykładu ciągi *PKOYTEK* i *MKJAJEOTI*. Są one dość krótkie, tak że nawet na pierwszy rzut oka można zidentyfikować spodziewany rezultat: *KOT*.

Jak najprościej odszukać funkcję LCS? We wszelkich problemach natury rekurencyjnej powinniśmy odszukać przypadek elementarny, który będzie podpowiadał dalszą dekompozycję algorytmu. W przypadku LCS założymy w tym celu, że dwa ciągi są zakończone tym samym znakiem x_i (lub y_i) stanowiącym w związku z tym ich LCS. Po usunięciu z nich tego znaku należy zbadać LCS ciągów skróconych (dokonujemy dekompozycji), a do wyniku dodamy wcześniej odnaleziony, wspólny element.

Zapis rekurencyjny tej reguły może być następujący:

$$LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) + x_i$$

Operator $+$ jest użyty jako funkcja sklejająca napisy (ciągi znaków).

Trochę bardziej złożony jest przypadek, gdy ostatnie znaki obu ciągów są *różne od siebie*. Nasze poszukiwanie rozgałęzia się na dwie odnogi:

- ◆ Usuwamy ostatni znak z ciągu Y i porównujemy z X , czyli badamy $LCS(X_i, Y_{j-1})$.
- ◆ Usuwamy ostatni znak z ciągu X i porównujemy z Y , czyli badamy: $LCS(X_{i-1}, Y_j)$.

Z dwóch otrzymanych wyników interesuje nas oczywiście *najdłuższy* z nich!

Podsumowując, funkcja rekurencyjna LCS może zostać sformułowana jako:

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & i = 0 \text{ lub } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + x_i & x_i = y_j \\ \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & x_i \neq y_j \end{cases}$$

Pierwszy wiersz dotyczy sytuacji, gdy ciąg znaków porównujemy z ciągiem pustym. Jest oczywiste, że funkcja LCS powinna wówczas zwrócić także ciąg pusty (oznaczony symbolem \emptyset). Pozostałe dwa warunki są powtórzeniem zidentyfikowanych wcześniej przypadków dekompozycji rekurencyjnej.

Poniżej przedstawiam program, który jest dosłownym przetłumaczeniem powyższej formuły. Pokazana w programie funkcja zwraca poszukiwany ciąg LCS. Na użytek programu przykładowego na *ftp*, który jest bardziej rozbudowany niż przedstawiana wersja drukowana, kluczowe zmienne (napisy, tablice) są zadeklarowane jako *globalne*.

*lcs.cpp*

```

const int M=7, N=9;

string X="PKOYTEK"; //M
string Y="MKJAEOTI"; //N

int i, j;
string S[M+1][N+1]; // tablica ciągów LCS
string LCS_ciag (string X, string Y)
{
    for (i=0; i<M; i++) S[i][0] = ""; //inicjalizacja
    for (i=0; i<N; i++) S[0][i] = "";
    for (i=1; i<M+1; i++)
        for (j=1; j<N+1; j++)
            if (X[i-1] == Y[j-1])
                S[i][j]=S[i-1][j-1] + X[i-1];
            else
                if ( S[i][j-1].length() > S[i-1][j].length() )
                    S[i][j]=S[i][j-1];
                else
                    S[i][j]=S[i-1][j];
    return S[M][N];
}

```

W „poważniejszym” programie warto oczywiście zadbać o większą parametryzację funkcji i hermetyzację kodu!

W praktyce użycie dwuwymiarowej tablicy zapamiętującej wyliczane w danych krokach algorytmu podciągi LCS o zmiennej długości jest kosztowne czasowo (operacje na ciągach znaków wymagają używania funkcji bibliotecznych, które są dość złożone), a zużycie pamięci może być znaczne. W związku z tymi wadami czasami stosuje się podejście polegające na zapamiętywaniu nie ciągów LCS, ale ich *długości*. Wzór rekurencyjny pozostaje niemal taki sam, jedynie zamiast sklejać ciągi znaków, dodaje się wyniki obliczeń (długości ciągów):

$$C(X_i, Y_j) = \begin{cases} 0 & i = 0 \text{ lub } j = 0 \\ C(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \\ \max(C(X_i, Y_{j-1}), C(X_{i-1}, Y_j)) & x_i \neq y_j \end{cases}$$

W kodzie C++ kluczowy przypadek elementarny z wiersza drugiego jest oznaczony komentarzem (*).

Odtworzenie samego LCS jest możliwe poprzez rekurencyjne przejście „od tyłu” śladem decyzji podejmowanych przez algorytm:

```

int C[M+1][N+1]; // tablica długości LCS

int LCS_dlugosc (string X, string Y)
{ // C – tablica długości ciągów LCS
    for (i=0; i<M; i++) C[i][0] = 0; //inicjacja

    for (i=0; i<N; i++) C[0][i] = 0;

    for (i=1; i<M+1; i++)
        for (j=1; j<N+1; j++)
            if (X[i-1] == Y[j-1])
                C[i][j]=C[i-1][j-1] + 1; //(*)
            else
                C[i][j]=max( C[i][j-1], C[i-1][j] );
}

```

```

return C[M][N];
}
// W tej funkcji odtwarzamy ciąg LCS:
string LCS_wypisz (int i, int j)
{
    if ( (i==0) || (j==0) )
        return "";
    if (X[i-1] == Y[j-1])
        return (LCS_wypisz(i-1, j-1) + X[i-1]);
    else
        if (C[i][j-1] > C[i-1][j])
            return LCS_wypisz(i, j-1);
        else
            return LCS_wypisz(i-1, j);
}
int main()
{
    cout << "LCS:" << LCS_ciag(X,Y) << endl;
    cout << "Długość LCS:" << LCS_dlugosc(X,Y) << endl;
    cout << "Wypisz LCS:" << LCS_wypisz(M,N) << endl;
}

```

(Powyższy kod znajduje się w tym samym pliku źródłowym co poprzednio).

W pliku źródłowym znajdują się dodatkowo funkcje pozwalające drukować zawartość tablic C i S , proponuję spróbować w ramach ćwiczenia rozrysować je na kartce papieru i zobaczyć, jak funkcjonuje algorytm dla różnych ciągów.

Inne techniki programowania

Informatyka czasami kojarzy się osobom niewykształconym z magią, gdyż jakimś cudem urządzenia elektroniczne potrafią rozwiązywać zadania, sprawiać wrażenie myślących partnerów w grach komputerowych, wspomagać nużące, codzienne czynności. Okazuje się jednak, że sami informatycy niekiedy dochodzą do wniosku, że ich dziedzina wiedzy ma jakiś związek z magią. Okazją do takich przemyśleń mogą być wykłady z dziedziny sztucznej inteligencji (rozdział 12.) lub właśnie zaawansowane techniki algorytmiczne, zwłaszcza te zwane heurystycznymi.

Czym jest heurystyka? W logice termin ten oznacza umiejętność wykrywania nowych faktów i związków między faktami, dzięki którym dochodzi się do poznania nowych prawd⁹. W informatyce heurystyka jest metodą znajdowania rozwiązań przybliżonych, ale zadowalających, zwłaszcza w sytuacji gdy rozwiązania idealne pozostają poza zasięgiem możliwości technicznych lub... nam na nich nie zależy. Heurystyczne algorytmy przeszukiwania są dopasowane do dziedziny przedmiotu: znając rozpatrywane zagadnienia (np. zasady pewnej gry, sposób postępowania przeciwnika w rozgrywkach, prawa fizyki, sposób działania wirusów komputerowych...), można tę wiedzę wbudować w program, aby został on niejako naprowadzony na wynik.

Metody heurystyczne mogą być użyte do wstępnego zawężenia przestrzeni rozwiązań, tak aby na końcu zastosować algorytm klasyczny, który wyliczy wiarygodny rezultat końcowy. W przypadku metody klasycznej przeszukiwania dziedziny rozwiązań ścieżki poszukiwań często prowadzi na „przysłowiowe manowce” i oznaczają tylko stratę czasu. Człowiek może zatem pomóc komputerowi, aby ten nie tracił czasu na wyszukiwanie rozwiązań na „ścieżkach” prowadzących do porażki lub niezbyt interesujących z punktu widzenia spodziewanego wyniku.

Stosując pewne uproszczenia, schemat algorytmu heurystycznego zakłada istnienie modelu naszego problemu do rozwiązania zawierającego:

⁹ Podaje, w wersji uproszczonej, za słownikiem wyrazów obcych PWN.

- ♦ *Stany* — opisujące bieżące konfiguracje danych lub decyzje.
- ♦ *Operatory* — przekształcające stany w inne stany, najlepiej w kierunku poszukiwanego rozwiązania.

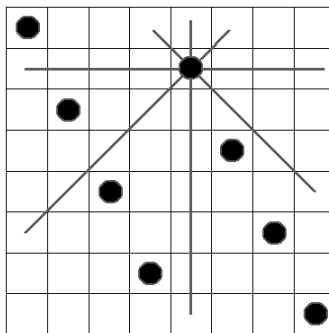
Stany są zazwyczaj umieszczane w wygodnej strukturze danych — np. drzewie, liście, grafie (o tych opowiemy w następnym rozdziale), ale oczywiście należy do nich dobudować całą obsługę algorytmiczną, która aktywnie używa wielu funkcji wspomagających, np. funkcji obliczających bieżące rozwiązanie i funkcji heurystycznych, które będą sterowały algorytmem. Algorytm heurystyczny podczas dokonywania przekształceń może w danym momencie wyliczać przy pomocy funkcji heurystycznych (te trzeba odnaleźć lub... wymyślić) pewne dodatkowe informacje, które pozwolą nam łatwiej podejmować decyzje dotyczące przekształceń jednych stanów w inne, tak aby zbliżyć się do optymalnego rozwiązania. Heurystyka, w zależności od postaci zagadnienia, którym się zajmujemy, może przybrać konkretną, matematyczną formę (funkcja wyliczająca pewne parametry sterujące algorytmem) lub, w postaci ogólniejszej, stanowić pewną metodę wyszukiwania rozwiązania niejako na skróty.

Przykład złożonego algorytmu wykorzystującego heurystykę:

- ♦ Chcemy ustawić K hetmanów na szachownicy o wymiarach $N \times N$, tak aby nie atakowały się wzajemnie. Przykład na rysunku 9.4 dla 8 hetmanów i klasycznej szachownicy. Dla jednego z hetmanów zaznaczone jest jego pole „rażenia”.
- ♦ „Stanem” w naszym przypadku będzie oczywiście bieżący rozkład szachownicy, zaczynając od szachownicy pustej.
- ♦ Poszukując optymalnego rozkładu, umieszczamy figury np. w kolejnych wierszach i konstruujemy drzewo wyborów. W zależności od dokonanego w danym etapie wyboru w kolejnych krokach zawężają nam się możliwości, gdyż wcześniej położone hetmany „atakują” pewien obszar szachownicy¹⁰.

Rysunek 9.4.

Problem 8 hetmanów



Zauważmy, że podejście ślepe, typu brute-force, zakładałoby wygenerowanie wszystkich możliwych rozkładów i przetestowanie, czy spełniają warunek nieatakowania się wzajemnie hetmanów. Można się domyślać, że tego typu rozwiązanie zazwyczaj jest zbyt czasochłonne, aby je zastosować (czasami wręcz niemożliwe). Spróbujmy zastanowić się, jak zrobić to w inny sposób.

Jak dokonać optymalnego wyboru? Otóż w tym miejscu wkracza heurystyka, czyli nasza zdolność oceny bieżącej konfiguracji na szachownicy. Jeśli generując kolejne pozycje, na których można położyć nowego hetmana, jednocześnie wyliczymy ich „wartość” względem pewnej (tajemniczej!) funkcji heurystycznej f , to możemy wyniki posortować i wybrać taki, który np. jest optymalny według zastosowanej heurystyki.

¹⁰ Dla przypomnienia: hetman atakuje w każdym kierunku (pion, poziomy i przekątny) oraz może przeskoczyć o dowolną liczbę pól.

Jak zdefiniować funkcję f ? Trzeba oczywiście się dobrze zastanowić, gdyż jakość tej funkcji determinuje wyniki działania algorytmu. W naszym przypadku zazwyczaj przyjmuje się jako kryterium np. liczbę niezagrażonych atakiem pól w kolejnych wierszach, tak, aby nie zawęzić kolejnych wyborów.

Wadą heurystyk jest ich aspekt praktyczny: często zamiast klarownego modelu matematycznego (dającego się przełożyć na kod) wbudowujemy w algorytm pewne reguły postępowania opierające się na intuicji i doświadczeniu. Jakość heurystyk badamy doświadczalnie, analizując... wyniki działania algorytmów!

Klasyczne metody heurystyczne oczywiście nie wyczerpują możliwych metod programowania. Warto wiedzieć o innych, ciekawych metodach wyszukiwania wyników, łamiących pewne przyzwyczajenia i „konwenanse”:

- ◆ *Losujemy* zbiór możliwych stanów (wyborów) w przestrzeni dopuszczalnych rozwiązań i obliczamy dla nich wcześniej zdefiniowane funkcje celu. Wybieramy najlepsze rozwiązanie i.. uznajemy je za rozwiązanie problemu.
- ◆ Stosujemy algorytmy zwane *genetycznymi*, które w swojej mechanice działania przypominają zjawiska ewolucji biologicznej. Aby nie trywializować bogatej dziedziny, jaką są algorytmy genetyczne (lub ogólniej, ewolucyjne), pominę tutaj szczegóły. Metoda algorytmu genetycznego budzi często uśmiech u osób nieorientowanych, ale wynika on często z tej nieszczęśliwej analogii biologicznej. Tak naprawdę mamy tu do czynienia z pewną formą modelowania rozwiązywania zagadnień optymalizacyjnych, gdzie parametry problemu kodujemy w strukturze danych zwanej chromosomem, który może być... ciągiem binarnym. Prawda, że brzmi to już bardziej informatycznie?

Czytelnik ma prawo zadać sobie słuszne pytanie: dlaczego nie pogłębiam poruszanych zagadnień opisywanych w tym punkcie, gdyż brzmią one co najmniej intrygująco? Powodem jest oczywiście bogactwo tematyczne. Opisywane (a raczej wzmiankowane) techniki doczekały się odrębnych grubych publikacji tylko im poświęconym. Niniejszy podręcznik zostałby po prostu rozsadzony próbą szerszego opisania choćby tylko samych zagadnień optymalizacyjnych! Warto jednak wiedzieć, że takie techniki programowania istnieją, osoby zajmujące się studiowaniem informatyki na pewno w toku studiów będą miały okazję się z nimi zetknąć.

Uwagi bibliograficzne

W tym rozdziale mieliśmy okazję poznać kilka prostych technik programowania, których efektywne użycie może znacznie zwiększyć sprawność programisty w rozwiązywaniu problemów za pomocą komputera. Oczywiście nie są to wszystkie metaalgorytmy, które można napotkać w literaturze problemu — wybór padł na te techniki, których pojęcie nie jest zbyt trudne i które nie wymagają pogłębionych studiów informatycznych.

Czytelnika zainteresowanego głębszymi studiami w dziedzinie technik programowania szczególnie zachęcam do sięgnięcia po [HS78] — książkę napisaną bardzo prostym językiem (cóż z tego, że angielskim) i zawierającą bardzo szczegółowe omówienie wielu różnorodnych strategii i technik programowania. Osoby zainteresowane wykorzystaniem struktur drzewiastych w rozwiązywaniu problemów algorytmicznych mogą połączyć lekturę ostatniej pozycji z [Nil82]. W przypadku braku dostępu do oryginalnego tytułu Nilssona dużo cennych informacji znajduje się również w [BC89]. Ostatnia praca, którą można polecić, to [CP84], ale może być dość trudna do zdobycia — jest to skrypt, w związku z tym należy go szukać nie na francuskim rynku wydawniczym, ale w tamtejszych bibliotekach uczelnianych.

Rozdział 10.

Elementy algorytmiki grafów

Grafy są niczym innym, jak *strukturą danych* i poświęcenie im osobnego rozdziału może wzbudzić pewne zdziwienie u Czytelnika. Zabieg ten wydaje się jednak konieczny z uwagi na szczególne znaczenie grafów w algorytmice. Stwierdzenie, iż bez tej struktury danych rozwiązanie wielu problemów algorytmicznych byłoby niemożliwe, nie jest przesadą.

Grafy posiadają dość złożoną podbudowę teoretyczną (w zasadzie można nawet wyodrębnić osobny dział matematyki im tylko poświęcony), ale w naszej prezentacji postaramy się unikać zbytniego formalizmu.

Odrobina teorii zostanie przedstawiona jedynie w celu ścisłego umiejscowienia omawianego problemu, ale z założenia będzie to niezbędne minimum.

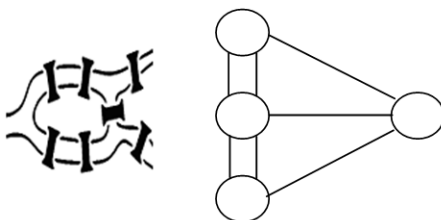
Czytelnikom zainteresowanym głębiej teorią grafów można w zasadzie polecić dowolny podręcznik algorytmiki, gdyż ta struktura danych zajmuje poczesne miejsce w literaturze przedmiotu. Interesujące podejście, będące mieszaniną matematyki i informatyki, prezentuje [Hel86], ale nie jestem jednak w stanie stwierdzić, czy tytuł ten jest już dostępny na rynku wydawniczym w formie książkowej, czy też pozostał na zawsze uproszczonym skrypcem uczelnianym. Z bardziej dostępnych pozycji polecam [CLR01] lub [Sed03].

Celem tego rozdziału jest zaprezentowanie podstawowej wiedzy (temat jest bowiem ogromny) dotyczącej grafów i sposobów ich reprezentacji w programach. Poznamy niezbędne słownictwo związane z tą strukturą danych, jak również przedstawimy kilka typowych algorytmów, które ich dotyczą.

Patrząc z perspektywy historycznej, grafy „narodziły się” w roku 1736 dzięki szwajcarskiemu matematykowi L. Eulerowi. Za ich pomocą rozwiązał on problem, który stawiali sobie mieszkańcy Koenigsberga (Królewca), a mianowicie: jak przemierzyć wszystkie siedem mostów znajdujących się w tym mieście tak, aby nie przechodzić dwukrotnie przez ten sam. Rysunek 10.1 pokazuje fragment mapy zawierający centralną wyspę Knipawę, opływaną przez rzekę Pregolę, która następnie po opłynięciu wyspy rozgałęzia się na dwie odnogi. Z prawej strony „mapki” jest pokazana umowna reprezentacja grafowa, która zakłada, że każdy most jest reprezentowany przez krawędź grafu, a węzły reprezentują fragmenty lądu (wyspy). Niejako przy okazji możemy poznać typową konwencję dotyczącą wierzchołków i krawędzi: te pierwsze reprezentują pewne obiekty (np. miasta, fragmenty lądu), te drugie — relacje wiążące ze sobą obiekty, np. drogi, mosty, połączenia lotnicze...

Rysunek 10.1.

*Model grafowy problemu
mostów w Królewcu*



Euler wykazał w swojej pracy, że zagadnienie jest nierozwiązywalne, a decyduje o tym konfiguracja mostów i wylotów (szersza dyskusja na ten temat znajduje się w punkcie „Cykle w grafach”, strona 214).

Ta historyczna anegdota stanowi jednocześnie doskonały przykład na to, do czego grafy mogą się w praktyce przydać: wszelkie zadania algorytmiczne, w których w grę wchodzi problem odnajdywania (optymalnych) dróg, mogą być przez grafy doskonale modelowane. Oczywiście nie tylko one — grafy znakomicie nadają się do rozwiązywania wielu innych zagadnień, np.:

- ◆ Analiza bezpieczeństwa sieci (czy uszkodzenie jednego z węzłów komunikacyjnych wpływa na możliwość przesłania informacji z punktu A do B?)
- ◆ Analiza obwodów elektronicznych (np. czy w danym projekcie nie występują zwarcia).
- ◆ Odszukiwanie informacji pierwotnych i odziedziczonych w dokumentach hipertekstowych (tzn. zawierających odsyłacze).
- ◆ Problem optymalnego doboru (np. licealiści aplikują na kilka uczelni na raz, aby wyrównać swoje szanse, a same uczelnie posiadają swoje własne ograniczenia w postaci liczby miejsc i kryteriów doboru najlepszych kandydatur).
- ◆ Kompilatory (oraz programy scalające kod, tzn. linkery) używają grafów do zbudowania sieci zależności funkcji i modułów

Programista, który dobrze pozna i zrozumie możliwości związane z użyciem grafów, praktycznie podwaja swoje kompetencje związane z umiejętnością sprawnego *modelowania* problemów do rozwiązania. Dość paradoksalną cechą wielu zagadnień algorytmicznych jest fakt, że potrafią się one rozwiązać niemalże same pod warunkiem wykonania dobrego modelu całości, a grafy, będące rozszerzeniem struktur drzewiastych, pasują do wielu zagadnień natury abstrakcyjnej i praktycznej.

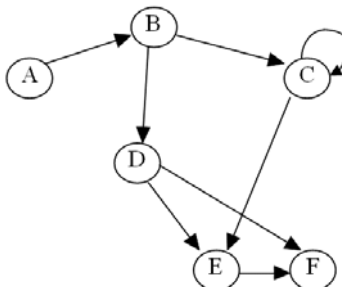
Następny paragraf będzie poświęcony podstawowemu słownictwu związanemu z grafami, po czym przejdziemy do sposobów reprezentowania ich w programach komputerowych.

Definicje i pojęcia podstawowe

Niezbyt duże grafy doskonale dają się przedstawiać w postaci wiele mówiących rysunków, takich jak np. 10.2.

Rysunek 10.2.

Przykład grafu



Grafem G nazywamy parę (X, Γ) , gdzie X oznacza zbiór tzw. węzłów (albo wierzchołków), a Γ zbiór $(x, y) \in X^2$ jest zespołem krawędzi.

Graf jest *planarny*, jeśli daje się go narysować na płaskim rysunku w taki sposób, aby jego krawędzie nie przecinały się. Zagadka: czy graf z rysunku 10.2 jest planarny? Rysunki poglądowe prostych grafów w książkach algorytmicznych mogą prowadzić do drobnego nieporozumienia, polegającego na tym, iż Czytelnik przypisuje rysunkom grafów bardzo duże znaczenie. W praktyce tylko grafy modelujące struktury fizyczne (np. mapy, ścieżki obwodów elektronicznych) są w jakiś sposób powiązane ze swoimi pierwotnymi obiektami poprzez wygląd. W zależności od potrzeb można bowiem tak dobrać odległości węzłów i długości ścieżek, aby struktura danych — z założenia abstrakcyjna — była powiązana ze swoim pierwowzorem ze świata realnego.

Liczba chromatyczna grafu jest najmniejszą liczbą kolorów, którymi można pokolorować wierzchołki grafu w taki sposób, aby każde dwa wierzchołki połączone krawędzią miały różne kolory.

Graf jest *skierowany* (ang. *directed graph*, czasami zwany też digrafem¹), jeśli krawędziom został przypisany jakiś kierunek (na rysunkach symbolizowany przez strzałkę). Graf skierowany czasami przedstawia się jako listę krawędzi $X \rightarrow Y$, strzałka w tej notacji odpowiada strzałce na krawędzi pomiędzy danymi wierzchołkami grafu.

Jeśli weźmiemy pod uwagę dwa węzły grafu, X i Y , połączone krawędzią, to węzeł X jest *węzłem początkowym*, a węzeł Y odpowiednio *węzłem końcowym*.

Stopień wejściowy wierzchołka grafu jest to liczba krawędzi dochodzących do wierzchołka. Analogicznie *stopień wyjściowy*, gdy określa liczbę krawędzi wychodzących od danego wierzchołka.

Grafem regularnym nazywamy graf, w którym każdy wierzchołek ma taki sam stopień.

Graf z rysunku 10.2 posiada 6 węzłów: A, B, C, D, E i F , niektóre z nich są połączone pomiędzy sobą krawędziami: (A, B) , (B, C) , (B, D) , (D, F) , (D, E) , (E, F) i (E, C) . Węzeł C ma charakter specyficzny, bowiem wychodzi z niego krawędź, która... wraca z powrotem do swojego węzła początkowego! W niektórych zagadnieniach algorytmicznych i takie dziwne krawędzie są potrzebne, bowiem można za ich pomocą modelować więcej sytuacji niż tylko z użyciem samych węzłów i krawędzi.

Ciąg wierzchołków, w którym każde dwa kolejne wierzchołki są połączone krawędzią, nazywamy *drogą* (ewentualnie *ścieżką*) w grafie. Droga (ścieżka) prosta charakteryzuje się tym, iż wszystkie należące do niej wierzchołki i krawędzie są różne.

Graf, w którym każde dwa wierzchołki są połączone drogą, nazywamy *spójnym*.

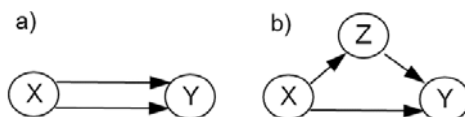
Podgraf grafu jest to graf, którego wierzchołki stanowią podzbiór zbioru wierzchołków grafu G , a krawędzie podzbiór zbioru krawędzi G . W opisach algorytmicznych podgrafy zwane są niekiedy grafami zredukowanymi.

Numery węzłów (lub też symboliczne: etykiety literowe) służą w zasadzie tylko do rozróżniania węzłów, bez przypisywania im jednakże jakiejś określonej kolejności. Programista może jednak w razie potrzeby narzucić numerom węzłów dodatkowe znaczenie (np. w teorii gier będzie to rekord opisujący stan gry).

Z definicji pomiędzy dwoma węzłami może istnieć tylko jedna krawędź, ale możliwe jest bardzo łatwe przejście z grafu, który nie jest zgodny z naszą definicją (patrz rysunek 10.3a), do grafu standardowego poprzez dołożenie dodatkowych wierzchołków (rysunek 10.3b).

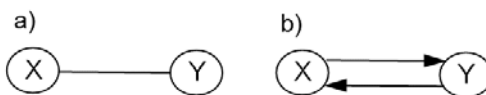
¹ Podaję ten termin tylko dla kompletności wykładu, osobiście nie uważam, aby promowanie tej kalki z języka angielskiego było potrzebne.

Rysunek 10.3.
„Normalizowanie”
grafu (1)



Pojęcie grafu skierowanego ma charakter najogólniejszy, gdyż graf *nieskierowany* (patrz rysunek 10.4a) może być bardzo łatwo przetransformowany na *skierowany* (rysunek 10.4b).

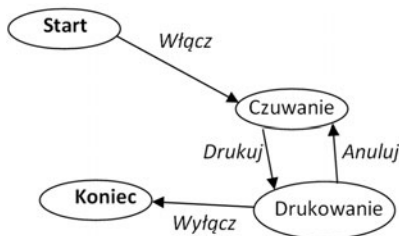
Rysunek 10.4.
„Normalizowanie”
grafu (2)



Przypisanie krawędziom grafu wartości (najczęściej liczbowych, ale mogą to również być etykiety innego typu) jest celowe dla pewnych zastosowań. Zmienia nam się wówczas definicja grafu, gdyż zamiast dwójki (X, I) mamy (X, I, V) . Trzeci parametr V oznacza właśnie zbiór wartości odpowiadających danym krawędziom, może to być jakaś umowna etykieta, „waga”, koszt, polecenie sterujące...

W ramach ilustracji rysunek 10.5 pokazuje, jak proste jest modelowanie przy pomocy grafu tzw. automatów skończonych. Bez wnikania w definicje automatów można przyjąć, że „stan” (węzeł grafu) reprezentuje pewną informację na temat systemu (urządzenia, mechanizmu). Przejścia pomiędzy stanami są wyzwalane przez reakcję na pewne zdarzenie (patrz etykieta krawędzi)².

Rysunek 10.5.
Graf przejść dla maszyny
stanów skończonych



W teorii grafów można napotkać jeszcze sporo innych definicji i pojęć, ale przyjmijmy, że nowe pojęcia, jeśli okażą się niezbędne, będą systematycznie wprowadzane w trakcie wykładu.

W dalszej części rozdziału przejdziemy do opisu kilku typowych metod reprezentowania grafów w pamięci komputera, co pozwoli nam na łatwiejszą prezentację samych algorytmów grafowych. Na dobry początek zaczniemy od omówienia dość popularnego zagadnienia występujących w grafach *cykli*.

Cykle w grafach

Droga w grafie, która kończy się w tym samym wierzchołku, w którym się zaczęła, jest zwaną *cyklem*.

Cyklem skierowanym (występującym z definicji w grafie skierowanym!) nazywamy cykl, w którym wszystkie pary sąsiadujących wierzchołków są uporządkowane zgodnie z kierunkiem krawędzi.

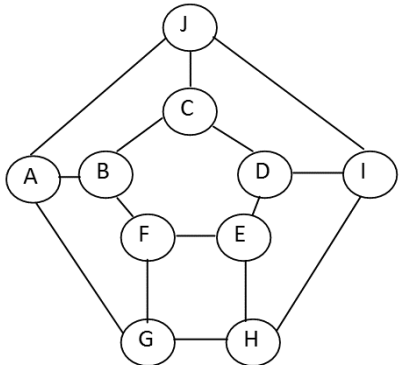
Graf skierowany, w którym nie występuje ani jeden cykl skierowany, zwany jest *acyklicznym grafem skierowanym* (ang. *directed acyclic graph*, czasami oznaczany skrótem DAG³).

² Osoby interesujące się poważniej informatyką lub elektroniką na pewno podczas nauki natrafią na teorię automatów i pojęcie automatu skończonego (maszyna Moore’a).

³ Kolejna dziwna kalka językowa, podają gwoli formalności!

Drogę przechodzącą dokładnie raz przez każdy wierzchołek i wracającą do wierzchołka wyjściowego nazywamy cyklem Hamiltona⁴ (patrz: rysunek 10.6, idąc alfabetycznie od węzła *A* do węzła *I*).

Rysunek 10.6.
Cykl Hamiltona



Załóżmy teraz, iż nasz graf reprezentuje w węzłach miasta, a krawędzie zawierają „wagi”, które mogą np. oznaczać odległości lub koszt przebycia danego odcinka drogi (patrz tabela 10.1).

Tabela 10.1. *Problem komiwojażera*

	Gdańsk	Poznań	Wrocław	Warszawa
Gdańsk	0	296	432	339
Poznań	296	0	178	310
Wrocław	432	178	0	344
Warszawa	339	310	344	0

W takim grafie wyszukanie cyklu Hamiltona o minimalnej sumie wag krawędzi oznacza rozwiązanie słynnego *problemu komiwojażera* (ang. *traveling salesman problem*), który chce wystartować z jednego miasta, odwiedzić jak najmniejszym kosztem wszystkie pozostałe i powrócić do punktu wyjścia. Problem komiwojażera należy do klasy problemów NP-zupełnych⁵.

W praktyce rozwiązanie problemu komiwojażera uzyskuje się przy pomocy algorytmów genetycznych, dających dobre, aczkolwiek przybliżone wyniki.

Cyklu Hamiltona nie należy mylić z *cyklem Eulera* dotyczącym krawędzi — tam ten sam wierzchołek może być odwiedzany wiele razy. W związku z pracą Eulera wspomnianą na początku rozdziału do dziś funkcjonuje pojęcie *grafu eulerowskiego*, czyli takiego, w którym da się skonstruować cykl Eulera — drogę, która pozwala na przejście każdej krawędzi grafu tylko raz.

Innym znanym problemem algorytmicznym z kategorii wyszukiwania dróg optymalnych jest problem chińskiego listonosza, gdzie szukamy drogi optymalnej, zakładając, że odwiedzamy każdą krawędź grafu (ulicę) co najmniej raz. Nazwa problemu wzięła się z powodu opublikowania w 1962 roku pierwszego opisu tego problemu w chińskim czasopiśmie matematycznym, a samo zagadnienie zostało spopularyzowane przez chwytliwą nazwę zaproponowaną przez Alana Goldmana.

⁴ Sir William Rowan Hamilton był znanym irlandzkim matematykiem żyjącym w latach 1805 – 1865. Prace Hamiltona i Eulera wzmogły zainteresowanie teorią grafów i jej szerokimi zastosowaniami.

⁵ Nie chciałem w tej książce wnikać w tematykę maszyny Turinga, która jest niezbędna do precyzyjnego zdefiniowania pojęcia algorytmu klasy NP-zupełnej. Przyjmijmy dla uproszczenia, że problem optymalizacyjny jest NP-zupełny, gdy wraz ze wzrostem liczby danych (np. węzłów grafu) nie można dla nich znaleźć algorytmów dających *dokładne* rozwiązanie problemu w czasie wielomianowym, zależnym od wielkości danych.

Wróćmy jeszcze do cyklu Eulera.

Warto zapamiętać stworzoną na podstawie prac Eulera następującą właściwość (czasami zwaną też twierdzeniem Eulera, co jest określeniem dobrym ale niewystarczającym, gdyż twierdzeń określanych jego nazwiskiem funkcjonuje w matematyce jeszcze co najmniej kilka):



Definicja

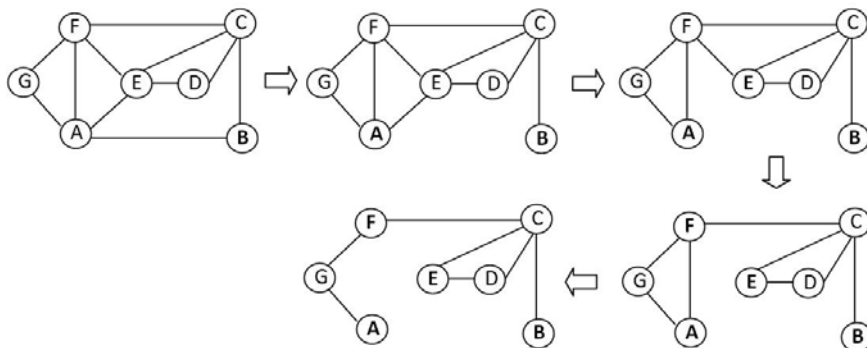
Graf nieskierowany zawiera cykl Eulera, jeśli jest *spójny* i jego wierzchołki mają *parzysty stopień*. W grafie skierowanym, zawierającym cykl Eulera, oprócz spójności musi występować taka sama liczba krawędzi wchodzących i wychodzących dla każdego wierzchołka.

Twierdzenie Eulera jest przydatne do zbadania warunku na istnienie cyklu Eulera, ale niestety nie pomaga nam go odszukać. Na szczęście istnieje prosty i elegancki algorytm Henry'ego Fleury'ego⁶, który pozwala nam znaleźć taki cykl w bardzo intuicyjny sposób:

1. Wybierz dowolny wierzchołek.
2. Wybierz krawędź spośród krawędzi wychodzących z wierzchołka, w którym aktualnie się znajdujemy. Wybieramy krawędź, po której jeszcze nie przechodziliśmy, i dodatkowo staramy się, aby wybrana krawędź nie była krawędzią cięcia grafu (chyba że nie ma wyboru).
3. Zapamiętaj wybraną krawędź i usuń ją z grafu.
4. Przejdź przez wybraną krawędź do następnego wierzchołka.
5. Powtarzaj kroki 2, 3 i 4 aż do momentu przejścia wszystkich krawędzi i powrotu do wierzchołka początkowego.

W każdym kroku algorytmu Fleury'ego tworzony jest graf zredukowany, to znaczy graf początkowy pozbawiony usuniętych wierzchołków. Pozostałe wierzchołki muszą, co oczywiste, tworzyć jeden spójny element. Inaczej to formułując, nie należy pociąć grafu na dwa kawałki. Jeśli stosując się do zasad algorytmu, uda nam się dojść do wierzchołka, z którego wyruszyliśmy, i przejść przez wszystkie krawędzie, to otrzymana droga jest cyklem Eulera.

Rysunek 10.7 ilustruje algorytm Fleury'ego w kilku jego pierwszych krokach.



Rysunek 10.7. Algorytm Fleury'ego na przykładzie

Zgodnie z przepisem wybieramy dowolny wierzchołek, niech to będzie B . Z dwóch jego „sąsiadów” do wyboru (A i C) podążamy arbitralnie w kierunku A , „wymazując” krawędź $B-A$. Podobnie postępując, idziemy np. od A do E i od E do F (prawa dolna część rysunku). W tym momencie przy

⁶ Algorytm został opisany w roku 1883, ciekawa analiza źródeł historycznych dotyczących jego publikacji i samego autora znajduje się na forum <http://mathforum.org>.

węzle F mamy do wyboru: C, A i G. Czy możemy wybrać C? Nie, gdyż wówczas nasz graf zredukowany zostałby podzielony na dwie rozłączne części! Wybieramy zatem np. A, G, F itp. aż do samego końca, czyli węzła B.

Algorytm Fleury’ego stanowi świetną ilustrację typowego sposobu opisywania algorytmów grafowych. Aby go zaimplementować przy pomocy komputera, trzeba jednak dysponować kilkoma narzędziami, których na razie nam brakuje. Należą do nich umiejętność modelowania grafów jako struktur danych i odrobina teorii. W dalszej części rozdziału nadrobimy te braki, zaczynając od pokazania sposobów reprezentowania grafów w pamięci komputera.

Sposoby reprezentacji grafów

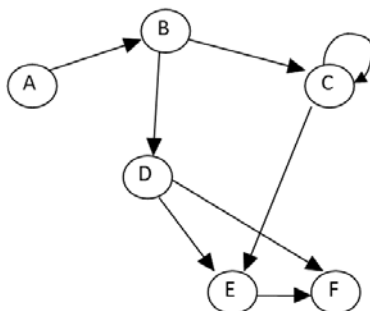
Poznane uprzednio struktury danych, takie jak tablice, listy i drzewa, dobrze nadają się do reprezentowania grafów. Dwie reprezentacje można uznać jednak za dominujące: za pomocą *tablicy dwuwymiarowej* (tzw. macierz sąsiedztwa) i tzw. *słownika węzłów* (tzw. lista sąsiedztwa), cząstami stosuje się także rozwiązanie oparte na zapisywaniu listy krawędzi przy pomocy *zbioru*.

Reprezentacja tablicowa

Nasze rozważania rozpoczniemy od najprostszego przypadku, czyli *tablicy dwuwymiarowej*. Jeśli umówimy się, że wiersze będą oznaczały węzły początkowe krawędzi grafu, a kolumny ich węzły końcowe, to przy takiej umowie graf z rysunku 10.2 może być przedstawiony w postaci tablicy z rysunku 10.8.

Rysunek 10.8.

Tablicowa reprezentacja grafu skierowanego



	A	B	C	D	E	F
A		1				
B			1	1		
C			1		1	
D					1	1
E						1
F						

Jedynka na pozycji (x, y) oznacza, że pomiędzy węzłami x i y istnieje krawędź skierowana w stronę y. W każdym innym przypadku tablica będzie zawierała na przykład zero. Oczywiście 0 lub 1 można zastąpić wartościami logicznymi (*true* i *false*).



Uwaga

Reprezentacja zerojedynkowa pozwala na intuicyjne zapisywanie grafów skierowanych przy użyciu zwykłych tablic dwuwymiarowych. Jeśli zmienimy typ tablicy z liczbowego na np. znakowy, to w komórkach tablicy będziemy mogli zapisywać także nazwy etykiet lub inne wartości. Oczywiście nasze algorytmy będą musiały „nauczyć się” poprawnej interpretacji zawartości takich tablic, np. brak wpisu mógłby oznaczać brak krawędzi, wpis znaku ‘q’ jednocześnie istnienie krawędzi i przypisanie jej etykiety ‘q’.

Zauważmy, że reprezentacja tablicowa ma jedną istotną zaletę: jest bardzo prosta do implementacji programowej w dowolnym w zasadzie języku programowania, a ponadto korzystanie z niej nie jest trudne. Wada: jedynie grafy o ustalonej z góry liczbie węzłów mogą być łatwo reprezentowane w postaci tablic.

Implementacja grafów w C++ przy użyciu tablic będzie wymagała w kodzie C++ deklaracji podobnych do poniższych:

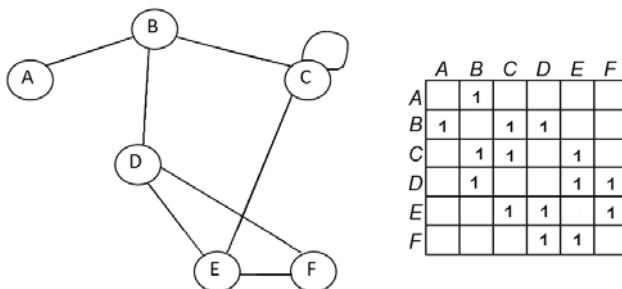
```
const int n=5;
int G[n][n];
```

Umawiając się, że indeksy $(x, y) \in (0, 1, 2, 3 \text{ i } 4)$ oznaczają węzły grafu A, B, C, D i E, poprzez wpis pewnej wartości do komórki $G[x][y]$ możemy zakodować istnienie krawędzi x-y łączącej dwa wierzchołki. Wartość w tablicy może oznaczać długość odcinka, jego koszt — wszystko zależy od naszej interpretacji.

Jak zaimplementować przy pomocy tablicy graf niekierowany? Można to uczynić, powielając wpisy dotyczące danej krawędzi (np. A-B i B-A). Przykład jest przedstawiony na rysunku 10.9.

Rysunek 10.9.

Tablicowa reprezentacja grafu skierowanego



Niestety, deklarując na sztywno maksymalną wielkość tablicy, zużywamy sporo pamięci, co może sprawić realny kłopot podczas próby uruchomienia takiego programu.



Ostrzeżenie

Próba uruchomienia programu zawierającego bardzo dużą, dynamiczną tablicę może się zakończyć przepełnieniem stosu (znany nam już komunikat *Stack overflow!*).

Jeśli rozmiar tablicy nie stanowi problemu, to ciągle pewnym kłopotem jest czytelne zarządzanie dodawaniem i usuwaniem węzłów (z krawędziami nie ma oczywiście kłopotu). Owszem, ponosząc pewien koszt programistyczny, w C++ możemy stworzyć klasę, która będzie skutecznie udawała graf o zmiennej liczbie węzłów i będzie oparta ciągle na tablicy, ale w praktyce będzie to ciężkie i nieefektywne rozwiązanie. Dodatkowo, gdy nasz graf posiada małą liczbę krawędzi, to stopień efektywnego wykorzystania takiej tablicy jest bardzo niski i większość komórek pozostanie pusta.



Uwaga

Olbrzymią zaletą reprezentacji tablicowej jest bezpośredni i tani (z punktu widzenia złożoności algorytmicznej) dostęp do informacji; aby sprawdzić, czy dwa węzły są połączone krawędzią, wystarczy odczytać odpowiednią komórkę tablicy (stały czas dostępu, niezależny od położenia węzła ani rozmiaru grafu).

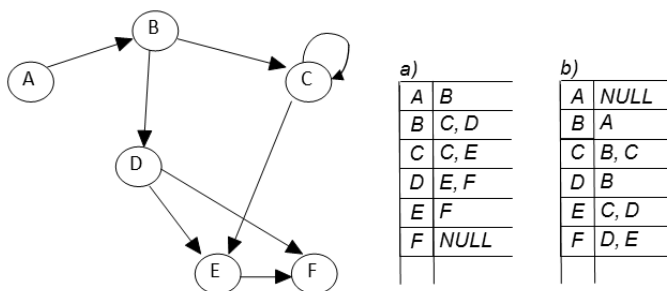
Słowniki węzłów

Aby przedstawić graf o liczbie węzłów, która może ulegać zmianie w trakcie wykonywania się programu, lepiej jest użyć tzw. *słownika węzłów* (tzw. lista sąsiedztwa, choć jest to nieco mylące określenie).

Słownik węzłów może dotyczyć dwóch typów węzłów: *następników* (węzłów *odchodzących*) lub *poprzedników* (węzłów *dochodzących*) danego węzła. Idea jest przedstawiona na rysunku 10.10.

Rysunek 10.10.

Reprezentacja grafu za pomocą słownika węzłów



Pojęcie „słownik” jest użyte z uwagi na charakter wyszukiwania: *kluczem* jest węzeł, który wskazuje na *dane*, czyli stowarzyszone węzły przylegające (taka sama reguła dotyczy zarówno grafów skierowanych, jak i zwykłych).

Słownik może być zwykłą listą (w ostateczności tablicą) wskaźników do list węzłów (brzmi to okropnie, ale naprawdę nie ma czego się bać!), odpowiednio *odchodzących* (rysunek 10.10a) lub *dochodzących* (rysunek 10.10b) do danego węzła za pomocą krawędzi. Niektóre algorytmy dotyczące grafów potrzebują właśnie tego typu informacji, stąd celowość dysponowania taką reprezentacją. Biorąc pod uwagę, że słownik węzłów można łatwo zrealizować z użyciem listy list, znika nam automatycznie problem napotkany przy reprezentacji tablicowej — liczba węzłów grafu może być w zasadzie nieograniczona.



Ostrzeżenie

Wadą reprezentacji z użyciem list jest *niebezpośredni* dostęp do informacji o tym, czy dwa węzły są połączone krawędzią: należy przeszukać listę (zbiór) węzłów przylegających, co determinuje złożoność liniową takiej operacji.

Listy kontra zbiory

Ciekawą możliwość implementacyjną oferuje rozwiązanie oparte na zapisywaniu listy krawędzi przy pomocy *zbioru*. Zakładając, że elementami zbioru byłyby pary krawędzi (np. skierowane), to graf z rysunku 10.10 wystarczyłoby zapisać jako zbiór par krawędzi:

{A, B}, {B, C}, {B, D}, {C, C}, {C, E}, {D, E}, {D, F}, {E, F}, {F, NULL}.

Pozostaje oczywiście kwestia implementacji podstawowych operacji w strukturze danych opartej na takim zbiorze:

- ♦ dodawanie i usuwanie pary wierzchołków,
- ♦ przeszukiwanie zbioru (iteratory i funkcje wyszukujące pary spełniające określone kryteria).

Ponieważ tablice, listy i zbiory nie stanowią dla nas tajemnic (rozdział 5.), to nie wątpię, iż Czytelnik poradzi sobie ze stworzeniem struktury danych (np. klasy *Graf*) w języku C++. W tej książce nie będę zajmował się opisywaniem takiej klasy, gdyż prezentowane dalej algorytmy stałyby się mocno zagmatwane. Stosując skrajnie obiektowe podejście, należałoby się zastanowić nad kilkoma klasami, które implementowałyby odpowiednio węzeł, listę wskaźników do list węzłów, listy węzłów odchodzących (przychodzących). Zamiast listy węzłów można użyć, jak pisałem, zbioru węzłów, wszystko zależy od tego, czy zależy nam na jakiejś formie uporządkowania węzłów przylegających o danego wierzchołka.

Mnogość możliwych implementacji pozostawia dużą swobodę programiście, który powinien zastanowić się nad dopasowaniem struktury danych do dziedziny problemu. Jeśli znamy maksymalną liczbę węzłów i model tablicowy byłby akceptowalny w kwestii zajętości pamięci, to nie ma co się zastanawiać — prostota użycia jest kluczowym argumentem.

Podstawowe operacje na grafach

Wiele algorytmów dotyczących grafów daje się łatwo wyrazić przy użyciu specjalnej notacji matematycznej, którą poznamy właśnie w tym paragrafie.

Mając dane dwa grafy: $G_1 = (X, \Gamma_1)$ i $G_2 = (X, \Gamma_2)$, możemy na nich zdefiniować następujące operacje:

Suma grafów

$$G_3 = G_1 + G_2 = (X, \Gamma_1 + \Gamma_2).$$

Graf wynikowy G_3 zawiera wszystkie krawędzie grafów G_1 i G_2 .

Kompozycja grafów

$$G_3 = G_1 \circ G_2 = (X, \Gamma_3) = \{(x, y) \mid \exists z \in X : (x, z) \in \Gamma_1 \text{ oraz } (z, y) \in \Gamma_2\}$$

Krawędzie (x, y) grafu wynikowego G_3 spełniają warunek, że istnieje pewien węzeł z , taki, że (x, z) należy do G_1 , a (z, y) należy do G_2 .

Kompozycja grafów może być dość łatwo zrealizowana programowo, np. tak jak na listingu znajdującym się poniżej (dla uproszczenia ograniczymy się tylko do reprezentacji tablicowej):



kompoz.cpp

```
void kompozycja(int g1[n][n], int g2[n][n], int g3[n][n])
{
    int z;
    for(int x=0; x<n; x++)
        for(int y=0; y<n; y++)
            {
                z=0;
                while(1) // pętla nieskończona
                {
                    if(z==n)
                        break; // wyjście z pętli
                    if( (g1[x][z]==1) && (g2[z][y]==1) )
                        g3[x][y]=1;
                    z++;
                }
            }
}
```

(Pełna wersja programu zawiera rozbudowany przykład w funkcji *main*).

Potęga grafu

Potęga G^p jest zdefiniowana w sposób rekurencyjny:

$$G^0 = D,$$

$$\forall p \geq 1, G^p = G^{p-1} \circ G = G \circ G^{p-1}.$$

D oznacza tzw. graf *diagonalny*, czyli taki, w którym istnieją wyłącznie krawędzie typu (x, x) . Z potęgą grafu jest związane dość ciekawe twierdzenie: (x, y) należy do G^p wtedy i tylko wtedy, jeśli w G istnieje droga o długości p , która prowadzi od węzła x do węzła y .

Graf jest dość ciekawym wytworem z punktu widzenia matematyki, gdyż zupełnie naturalnie pozwala on przez samą swoją konstrukcję wyrazić *relacje binarne* zdefiniowane na zbiorze swoich wierzchołków X .

Elementarnym przykładem niech będzie pojęcie *symetrii*: jeśli istnienie krawędzi (x, y) implikuje istnienie krawędzi (y, x) , to możemy powiedzieć o grafie, że jest on *symetryczny*. W podobny sposób można zdefiniować całkiem sporo innych relacji binarnych, z których większość... nie ma żadnego praktycznego zastosowania. Wyjątkiem jest relacja przechodniości, która oznacza, że każda droga grafu G o długości większej lub równej 1 jest „podtrzymywana” przez jakąś krawędź.

Dlaczego relacja przechodniości jest taka ważna? Otóż przechodniość — paradoksalnie — nic nie oznacza. Jest ona po prostu wygodnym środkiem do zdefiniowania tzw. *domknięcia przechodniego grafu*, oznaczanego typowo przez $G^+ = (X, \Gamma^+)$, gdzie:

$$G^+ = \{(x, y) \mid \text{istnieje droga od } x \text{ do } y \text{ w grafie } G\}.$$

Jeśli umiemy dokonać domknięcia przechodniego grafu, to umiemy odpowiedzieć na ważne pytanie, czy możliwe jest przejście po krawędziach grafu od jednego wierzchołka do drugiego. Za-uważmy, że domknięcie przechodnie nie daje przepisu na przejście od danego wierzchołka do wierzchołka y : dowiadujemy się tylko, czy jest to możliwe.

Jednym z możliwych sposobów na obliczenie domknięcia przechodniego grafu jest wyliczenie go w sposób przedstawiony poniżej:

$$G^+ = G \cup G^2 + \dots + G^n.$$

(n oznacza liczbę wierzchołków grafu, czyli nieco bardziej formalnie $n = |X|$).

Zaletą powyższego algorytmu jest prostota zapisu, wadą — czego nietrudno się domyślić — złożoność realizacji i duży koszt otrzymywanych algorytmów.

Czytelnik dysponujący dużą ilością wolnego czasu może bez zbytniego wysiłku wymyślić co najmniej jeden algorytm, który realizuje domknięcie przechodnie wg powyższego przepisu. Warto może jednak z góry uprzedzić, że nie będzie to miało specjalnego sensu, gdyż istnieje inny algorytm, który przewyższa jakością wszelkie wariacje algorytmów otrzymanych na podstawie potęgowania grafów. Jest to słynny algorytm *Roy-Warshalla*, który zostanie omówiony w paragrafie następnym.

Algorytm Roy-Warshalla

Algorytm omawiany w tym paragrafie charakteryzuje się kilkoma cechami, które powodują, że w zasadzie jest on bezkonkurencyjny, jeśli chodzi o obliczanie domknięcia przechodniego grafu. Przede wszystkim nie używa on żadnych grafów dodatkowych (czego nie da się uniknąć w przypadku algorytmów opartych na potęgowaniu), a ponadto pozwala dość łatwo odtworzyć drogę, którą należy pójść, aby przejść po krawędziach od jednego wierzchołka do drugiego.

Algorytm bazuje na operacji Θ , która dla grafu $G = (X, \Gamma)$ jest zdefiniowana w sposób następujący:

$$\Theta_k = \Gamma \cup (y, z) \mid (x, k) \in \Gamma \text{ oraz } (k, y) \in \Gamma.$$

Zapis powyższy oznacza, że dla danego wierzchołka k do zbioru krawędzi Γ dorzucamy krawędzie łączące *poprzedniki* i *następniki* tego wierzchołka.

Jest możliwe udowodnienie, że domknięcie przechodnie grafu może być obliczone poprzez *sukcesywną kompozycję* operacji Θ , tzn. dla grafu o wierzchołkach $1, \dots, n$:

$$\Gamma^+ = \Theta_n(\Theta_{n-1} \dots (\Theta_1) \dots).$$

Algorytm zapisuje się bardzo prosto w C++:



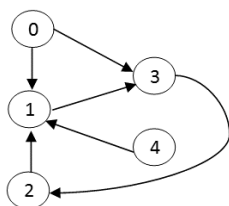
warshall.cpp

```
void warshall(int g[n][n])
{
    for(int x=0; x<n; x++)
        for(int y=0; y<n; y++)
            for(int z=0; z<n; z++)
                if(g[y][z] == 0)
                    g[y][z]=g[y][x]*g[x][z];
}
```

W celu dokładnego zrozumienia tego programu prześledźmy jego wykonanie na przykładzie prostego grafu 5-węzłowego przedstawionego na rysunku 10.11.

Rysunek 10.11.

Przykładowe wykonanie algorytmu Roy-Warshalla



	0	1	2	3	4
0		x		x	
1				x	
2		x			
3			x		
4		x			

	0	1	2	3	4
0		x	x	x	
1		x	x	x	
2		x	x	x	
3		x	x	x	
4		x	x	x	

Efekt wykonania algorytmu Roy-Warshalla

(Zamiast tradycyjnych jedynek na rysunku zostały użyte znaki x).

Z tablicy pokazanej na rysunku 10.11 możemy odczytać m.in. następujące informacje:

- ◆ Nie jest możliwe dojście do węzłów o numerach 0 i 4.
- ◆ Z węzła o numerze 1 możemy dojść do 2, 3 i... 1 (natrafiliśmy na tzw. *obwód zamknięty*).

Nawet na tak prostym przykładzie możemy już co najmniej poczuć ogromne możliwości, jakie oferuje nam algorytm *Roy-Warshalla*.

Jest on niesłychanie prosty zarówno ideowo, jak i w zapisie, co klasyfikuje go do grona algorytmów, które prywatnie określam jako eleganckie (J. Bentley używa do tego celu wyrażenia: *perła programistyczna*).

Algorytm *Roy-Warshalla* może być w dość prosty sposób zmodyfikowany, tak aby dostarczyć informacji nie tylko o istnieniu drogi wiodącej od wierzchołka x do wierzchołka y , ale oprócz tego podać przepis, którędy należy pójść.

W celu odnalezienia drogi (oczywiście jeśli w ogóle ona istnieje!) przyporządkujemy macierzy reprezentującej graf tzw. *macierz kierowania ruchem* (ang. *routing matrix*) R . Jest ona zdefiniowana w sposób następujący:

- ◆ $R[x, y] = 0$, jeśli nie ma drogi, która wiedzie od x do y .
- ◆ $R[x, y] = z$, gdzie z oznacza następny wierzchołek na drodze od x do y .

Konstrukcja macierzy umożliwia w naturalny sposób odtworzenie drogi wiodącej od danego wierzchołka do innego:

**route.cpp**

```

void pisz(int x, int y, int R[n][n])
{
    int k;
    if(R[x][y]==0)
        cout << "Droga nie istnieje\n";
    else
    {
        cout << x << endl;
        k=x;
        while(k!=y)
        {
            k=R[k][y];
            cout << k << endl;
        }
    }
}

```

Wiemy już, jak wypisać drogę na podstawie macierzy R , zatem pora najwyższa na przedstawienie algorytmu, który ją prawidłowo dla danego grafu wyliczy.

Przedstawiona poniżej procedura `route` zakłada, że macryca R przekazana jej w parametrze została zainicjowana uprzednio w następujący sposób:

- ♦ $R[x, y] = 0$, jeśli nie istnieje krawędź (x, y) .
- ♦ $R[x, y] = y$ w przeciwnym wypadku.

Zapis procedury jest banalnie prosty:

```

void route(int R[n][n])
{
    for(int x=0; x<n; x++)
        for(int y=0; y<n; y++)
            if(R[y][x]!=0) // wiemy, jak dojść z 'y' do 'x'
                for(int z=0; z<n; z++)
                    if( (R[y][z]==0) && (R[x][z]!=0) )
                        R[y][z]=R[y][x];
}

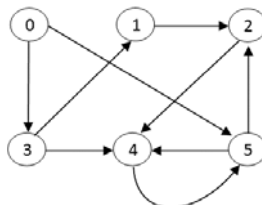
```

Algorytm jest oczywistą wariacją algorytmu poznanego uprzednio i wygląda podobnie jak on niewinnie. Jednak matematyczny dowód na to, że działa poprawnie, bynajmniej nie jest prosty.

Popatrzmy na efekt wykonania procedury `route` dla grafu przedstawionego na rysunku 10.12 (kilka przykładów).

Rysunek 10.12.

Poszukiwanie
drogi w grafie



```

Droga od 0 do 2: 0 3 1 2
Droga od 1 do 0: Droga nie istnieje
Droga od 1 do 5: 1 2 4 5
Droga od 2 do 0: Droga nie istnieje
Droga od 4 do 2: 4 5 2
Droga od 5 do 3: Droga nie istnieje

```

Kolejnym problemem, który omówimy, będzie znajdowanie w grafie drogi optymalnej pod względem kosztów.

Algorytm Floyd-Warshalla

Algorytm, który omówimy w tym paragrafie, pozwala na odnalezienie najkrótszych dróg między wszystkimi parami wierzchołków w grafie. Aby umożliwić jego wykonanie, musimy dysponować wartościami przypisywanymi krawędziom grafu. W tym celu założymy, że dysponujemy macierzą W , w której są zapamiętane wartości przypisane krawędziom grafu:

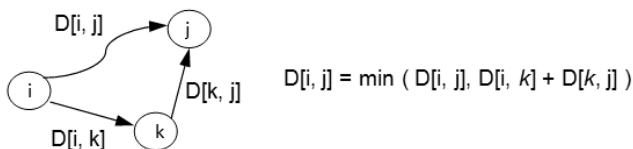
- ◆ $W[i, i] = 0$;
- ◆ $W[i, j]$ = wartość przypisana krawędzi lub ∞ (inaczej: bardzo duża liczba).

Ideę algorytmu w zrozumiały sposób prezentuje następujący przykład:

Założmy, że szukamy optymalnej drogi od i do j . W tym celu przechadzamy się po grafie, próbując ewentualnie znaleźć inny, pośredni wierzchołek k , którego wbudowanie w drogę umożliwiłoby otrzymanie lepszego wyniku niż już obliczone $D[i, j]$. Znajdujemy pewne k i zadajemy pytanie: czy przejście przez wierzchołek k poprawi nam wynik, czy nie?

Rysunek 10.13 ilustruje odpowiedź na powyższe pytanie w nieco bardziej poglądowej formie niż goły wzór matematyczny (przedstawiony obok).

Rysunek 10.13.
Algorytm
Floyda-Warshalla (1)



Jest oczywiste, że w przypadku większej liczby takich optymalnych wierzchołków pośrednich należy wybrać najlepszy z nich!

Algorytm można zatem zapisać rekurencyjnie w postaci wzoru:

$$D_{i,j}^{(k)} = \begin{cases} w_{i,j}, & k = 0 \\ \min(D_{i,j}^{(k-1)}, D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}) & k \geq 1 \end{cases}$$

Przypadek elementarny jest związany z sytuacją, gdy na drodze pomiędzy i oraz j nie ma wierzchołków pośrednich (krawędź).

Przedstawiony poniżej program jest najprostszą formą algorytmu *Floyda*, która wyłącznie *oblicza* wartość optymalnej drogi, ale jej nie *zapamiętuje*. Program w C++ wykorzystuje ideę programowania dynamicznego (wyniki cząstkowe służą późniejszym wyliczeniom):



floyd.cpp

```
const int n=7;
int G[n][n];

void floyd(int g[n][n])
{
    for(int k=0; k<n; k++)
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                g[i][j]=min( g[i][j], g[i][k]+g[k][j]);
}
```



```

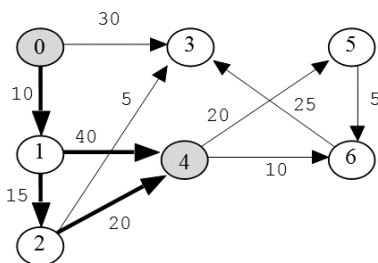
int main()
{
    for(int i=0;i<n;i++) // inicjalizacja grafu
        for(int j=0;j<n;j++)
            G[i][j]=10000; // 10000 = umowna „nieskończoność”
    G[0][3]=30; G[0][1]=10; // graf, jak na rysunku w książce
    G[1][2]=15; G[1][4]=40; G[2][3]=5; G[2][4]=20; G[4][5]=20; G[4][6]=10; G[5][6]=5; G[6][3]=25;
    floyd(G); // Wywołujemy algorytm i sprawdzamy wynik:
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
        {
            if(G[i][j]==10000)
                cout << i << " --> " << j << "[drogi nie ma]\n";
            else
                if(i!=j)
                    cout << i << " --> " << j << "="<<G[i][j]<<endl;
        }
}

```

Popatrzmy na rysunek 10.14, który przedstawia przykład wyboru optymalnej drogi przez algorytm *Floyda*.

Załóżmy, że interesuje nas optymalna droga od wierzchołka nr 0 do wierzchołka nr 4. Z uwagi na dość prostą topografię grafu widać, że mamy do wyboru dwie drogi: 0-1-4 i nieco dłuższą: 0-1-2-4.

Rysunek 10.14.
Algorytm
Floyda-Warshalla (2)



Elementarne obliczenia wykazują, że druga trasa jest efektywniejsza (koszt: 45) od pierwszej (koszt: 50).

Brak możliwości odtworzenia optymalnej drogi jest dość istotną wadą, gdyż o ile w przypadku małych grafów możemy ją ewentualnie odczytać sami, to przy naprawdę dużych grafach jest to praktycznie niewykonalne.

Potrzebna jest nam zatem jakaś prosta modyfikacja algorytmu *Floyda*, która nie zmieniając jego zasadniczej idei, umożliwi zapamiętanie drogi.

Jak się okazuje, rozwiązanie nie jest trudne. Do oryginalnego algorytmu (patrz listing wyżej) należy wprowadzić następującą poprawkę:



floyd2.cpp

```

...
    if( g[i][k]+g[k][j]<g[i][j])
    {
        g[i][j]=g[i][k]+g[k][j];
        R[i][j]=k;
    }
...

```

Optymalna droga będzie zapamiętywana w macierzy kierowania ruchem R . Czytelnikowi nie powinno sprawić zbytniego kłopotu napisanie procedury, która odtwarza znajdującą się w niej drogę. Załóżmy, że początkowo macierza R jest wyzerowana. Aby odtworzyć optymalną drogę od wierzchołka i do wierzchołka j , patrzmy na wartość $R[i][j]$. Jeśli jest ona równa zero, to mamy do czynienia z przypadkiem elementarnym, tzn. z krawędzią, którą należy przejść. Jeśli nie, to droga wiedzie od i do $R[i][j]$ i następnie od $R[i][j]$ do j . Zważywszy na fakt, iż powyższe dwie poddrogi mogą nie być elementarne, łatwo zauważyć rekurencyjny charakter procedury:

```
void droga(int i, int j)
{
    int k = R[i][j];
    if (k != 0)
    {
        droga(i,k);
        cout << k << " ";
        droga(k,j);
    }
}
```

Popatrzmy na koniec na przykład wywołania naszych pieczołowicie stworzonych procedur:

```
int main()
{
    initG();    // zerowanie grafu, wszędzie G[i][j] = ∞
    initR();    // zerowanie macierzy kierowania ruchem, wszędzie R[i][j] = 0

    G[0][3]=30; G[0][1]=10; G[1][2]=15; G[1][4]=40; G[2][3]=5;
    G[2][4]=20; G[4][5]=20; G[4][6]=10; G[5][6]=5; G[6][3]=25;

    floyd(G);  // wywołanie algorytmu i jego sprawdzenie:
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
        {
            if(G[i][j]==10000)
                cout << i << " --> " << j << "[drogi nie ma]\n";
            else
            {
                if(i!=j)
                {
                    cout << i << " --> " << j << "=<<G[i][j]<<".droga przez:";
                    droga(i,j);
                    cout << endl;
                }
            }
        }
}
```

Niektóre wyniki:

```
0 --> 4=45, droga przez: 1 2
0 --> 5=65, droga przez: 1 2 4
0 --> 6=55, droga przez: 1 2 4
1 --> 0[drogi nie ma]
1 --> 4=35, droga przez: 2
1 --> 5=55, droga przez: 2 4
1 --> 6=45, droga przez: 2 4
2 --> 5=40, droga przez: 4
```

Z uwagi na strukturę algorytmu widać od razu, że jest on klasy $O(N^3)$. Ciekawe jest, iż klasa algorytmów poszukujących wszystkich ścieżek w grafie jest bardzo podobna do klasy algorytmów zajmujących się pojedynczą ścieżką (patrz: kolejny punkt).

Algorytm Dijkstry

Kolejną odmianą algorytmu służącego do wyznaczania najmniejszej odległości od *jednego ustalonego* wierzchołka grafu s (skierowanego lub nie) do wszystkich pozostałych wierzchołków będzie klasyczny algorytm *Dijkstry* — jest on najbardziej znanym algorytmem poszukiwania najkrótszej drogi w grafie. Jedynym ograniczeniem w tym algorytmie jest dysponowanie grafem o dodatnich wagach, ale w przypadku zastosowań typu kartografia ograniczenie to nie ma specjalnego znaczenia.

Oznaczenia zastosowane w opisie algorytmu:

S — zbiór wierzchołków, których najkrótsza droga od źródła została już odnaleziona.

s — węzeł źródłowy (tzw. źródło).

t — węzeł docelowy (tzw. ujście).

$V-S$ — pozostałe wierzchołki.

D — tablica najlepszych oszacowań odległości od źródła do każdego wierzchołka v w grafie G .

$w[u, v]$ — waga krawędzi $u-v$ w grafie.

Pr — tablica poprzedników.

Algorytm przedstawimy w pseudojęzyku programowania, co zapewni nam zwięzłość opisu (choć oczywiście utrudni próby implementacji w konkretnym języku programowania):

```

Inicjacja:
Dla każdego węzła  $v$  wykonaj
{  $D[v] = \infty$ ;  $Pr[v] = \text{NULL}$ ; }
 $D[s] = 0$  // węzeł źródłowy!
 $S = \text{NULL}$ 
 $Q = \{\text{wstaw wszystkie wierzchołki grafu do kolejki}\}$ 
while not Empty( $Q$ )
     $u = \text{wyjmij z } Q \text{ element o najmniejszym } D[u]$  // kolejka priorytetowa!
     $S = S \cup u$ 
    dla każdego wierzchołka  $v$  z listy następników wierzchołka  $u$  wykonuj
    {
        // tzw. relaksacja, sprawdzamy, czy bieżące oszacowanie
        // najkrótszej drogi do  $V$  (tzn.  $D[v]$ ) może zostać ulepszone,
        // jeśli przejdziemy przez  $u$  ( $u$  staje się poprzednikiem  $v$ ):
        if  $D[v] > D[u] + w[u, v]$  then
            {
                 $D[v] = D[u] + w[u, v]$ 
                 $Pr[v] = u$ 
            }
    }

```

Algorytm może nie jest zbyt czytelny, ale jego idea jest intuicyjnie prosta:

- ♦ Kluczowe znaczenie ma tablica oszacowań D , w której na początku wszystkie wierzchołki, oprócz źródłowego (s), o którym wiemy, że odległość wynosi 0 , otrzymują tymczasowe oszacowanie ∞ .
- ♦ W dalszych krokach wierzchołki połączone krawędzią z wierzchołkiem s otrzymują oszacowania tymczasowe równe odległości od s (wartość krawędzi). Wybieramy z nich wierzchołek v o *najmniejszej* wartości oszacowania $D[v]$ i patrzymy na jego następniki. Jeśli droga z s do któregoś z nich, przechodząc przez v , ma mniejszą długość od dotychczasowej estymacji, to zmniejszamy ją (relaksacja).
- ♦ Odszukujemy wierzchołek o najmniejszym oszacowaniu tymczasowym i kontynuujemy algorytm aż do momentu dotarcia do t .

Dla kompletności algorytmu podam też sposób na odczytanie najkrótszej ścieżki od s do t (pseudokod):

```

path (s, t)
{
  S = NULL // pusta sekwencja
  u = t
  while(true)
  {
    S = u + S // wstaw u na początek S
    if (u == s) then break;
    u = Pr[u]
  }
}

```

Algorytm Dijkstry wykonuje się w najgorszym przypadku w czasie proporcjonalnym do V^2 , jednak implementując zawartą w nim kolejkę przy pomocy kopca, można osiągnąć rezultat $O(E \log V)$.

Algorytm Bellmana-Forda

Wadą algorytmu Dijkstry jest niemożność zaaplikowania go na grafach ze ścieżkami o wagach ujemnych, gdyż zakłada on, iż dodanie do ścieżki dodatkowych krawędzi może ją jedynie wydłużyć, zatem waga ujemna zakłóciłaby tę regułę.

Richard E. Bellman i Lester Randolph Ford Jr. zaproponowali modyfikację algorytmu Dijkstry polegającą na tym, że zamiast relaksacji opartej na wybranym węźle zapewniającym minimalną wagę (podejście zachłanne), dokonujemy relaksacji *wszystkich* węzłów $|V|-1$ razy, co zapewnia poprawne propagowanie minimalnych odległości poprzez graf ($|V|$ oznacza liczbę wierzchołków).

Algorytm w pseudokodzie jest dość podobny do algorytmu Dijkstry:

```

Inicjacja:
Dla każdego węzła v wykonaj
{ D[v] = ∞; Pr[v] = NULL; }
D[s] = 0 // węzeł źródłowy!

// relaksacja:
for i=1 to |V[G]| - 1 // liczba krawędzi minus 1
  Dla każdej krawędzi (u,v) wykonaj
    if D[v] > D[u] + w(u,v) then
      {
        D[v] = D[u] + w(u,v)
        Pr[v] = u
      }
// czy w grafie występują cykle ujemne?

Dla każdej krawędzi (u,v) wykonaj
  if D[v] > D[u] + w(u,v) then
    Pisz „Graf posiada cykl ujemny”

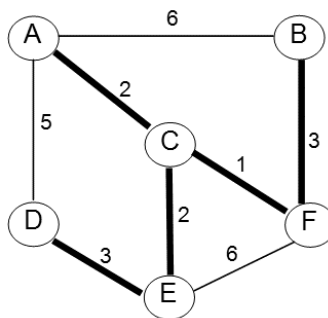
```

Jeśli graf zawiera cykl złożony z krawędzi o wagach ujemnych, to algorytm Bellmana-Forda nie może w takim grafie odnaleźć poszukiwanej drogi (ale może wykryć taką sytuację). Złożoność algorytmu wynosi $O(|V| \cdot |E|)$.

Drzewo rozpinające minimalne

W teorii grafów można napotkać na pojęcie tzw. minimalnego drzewa rozpinającego. Jest to zbiór krawędzi grafu nieskierowanego łączącego wszystkie wierzchołki tak, że suma wszystkich wag (lub długości) krawędzi wchodzących w ich skład jest możliwie najmniejsza. Definicję powinien wyjaśnić przykład grafu i jego minimalnego drzewa rozpinającego przedstawiony na rysunku 10.15.

Rysunek 10.15.
Drzewo rozpinające
minimalne



Wbrew pozorom nie jest to przykład akademicki, gdyż omawiane zagadnienie znajduje szereg praktycznych zastosowań:

- ♦ Przy projektowaniu układów elektronicznych możemy zbadać rozpraszanie najtańszym kosztem sygnału po punktach lutowniczych.
- ♦ Projektujemy okablowanie sygnału telewizji kablowej po wszystkich mieszkaniach (wierzchołki) z użyciem szacht instalacyjnych (krawędzie).
- ♦ Zakładamy sieć telefoniczną w wiosce, pomiędzy domami o trudnym dostępie (teren górzysy: pagórki, strumienie).

Najbardziej znanymi algorytmami rozwiązującymi zagadnienie poszukiwania w grafie drzewa rozpinającego są: algorytm *Kruskala*⁷ i algorytm *Prima*⁸ — oba z gatunku „żarłoczych”: do rozrastającego się w trakcie poszukiwań drzewa dodawane są wyłącznie krawędzie o najmniejszej wadze.

Algorytm Kruskala

- ♦ Tworzymy las (zbiór drzew) L z wierzchołków grafu i na początku traktujemy każdy z wierzchołków jako osobne drzewo.
- ♦ Tworzymy zbiór S wszystkich krawędzi grafu i sortujemy go w porządku niemalejącym względem ich wag.
- ♦ W trakcie działania algorytmu wybieramy i usuwamy ze zbioru S krawędź o najmniejszej wadze (dlatego warto mieć je wcześniej posortowane).
- ♦ Badamy, czy dobierana krawędź należy do dwóch różnych drzew. Jeśli tak, to należy je scalić, w przeciwnym wypadku *odrzucaamy ją*.
- ♦ Algorytm kontynuujemy do czasu, aż wszystkie wierzchołki znajdą się w pojedynczym drzewie docelowym (rozwiązaniu).

Spróbujmy odtworzyć działanie algorytmu Kruskala dla przykładowego grafu z rysunku 10.15. Kolejno dobierane (odrzucające wierzchołki) pokazuje tabela 10.2. Dobierając krawędź, należy ją „pogrubzić”, dzięki czemu bardzo łatwo będzie ocenić, czy kolejna krawędź „dotyka” wcześniej utworzonego drzewa w dwóch miejscach (cykl), czyli tak naprawdę stanowi jego część. Algorytm w postaci zaprezentowanej opisowo na pewno brzmi enigmatycznie, ale wystarczy użyć ołówka i kartki papieru, aby go pomyślnie przetestować.

⁷ Joseph Kruskal opracował go w 1956 roku.

⁸ Robert C. Prim opracował go w 1957 roku.

Tabela 10.2. Kroki algorytmu Kruskala

Dobierana krawędź (pogrubiona)	Komentarz
1	Akceptujemy
1,2	Akceptujemy
1, 2, 2	Akceptujemy
1, 2, 3, 3	Akceptujemy
1, 2, 2, 3, 3	Akceptujemy
1, 2, 2, 3, 3, 5	Odrzucamy (to samo drzewo)
1, 2, 2, 3, 3, 5, 6	Odrzucamy (to samo drzewo)
1, 2, 2, 3, 3, 5, 6, 6	Odrzucamy (to samo drzewo)

Algorytm Prima

- ◆ Mając dany graf wejściowy $G(V,E)$, wybieramy z niego dowolny wierzchołek i rozpoczynamy tworzenie drzewa.
- ◆ Dla każdego kolejno dodawanego do drzewa wierzchołka powtarzamy następujące kroki:
 - ◆ Dodajemy do drzewa krawędź o najmniejszej wadze, osiągalną z *jakiegokolwiek* wierzchołka skonstruowanego do tej pory drzewa (jeśli występuje kilka krawędzi o tej samej wadze, wybieramy arbitralnie jedną z nich). Do oceny, którą krawędź wybrać, najwygodniej jest sortować krawędzie przylegające ich wagami.
 - ◆ Dodajemy do drzewa wybraną krawędź i nowy wierzchołek osiągnięty po przejściu przez tę krawędź.

Spróbujmy odtworzyć działanie algorytmu Prima dla naszego przykładowego grafu (rysunek 10.15). Kroki algorytmu i przetwarzane obiekty pokazuje tabela 10.3 (rozpoczynamy np. od wierzchołka C).

Tabela 10.3. Kroki algorytmu Prima

Wierzchołki	Wagi osiągalnych krawędzi	Wybierany wierzchołek
C	1, 2, 2	F
C, F	2, 2, 3, 6	E
C, E, F	2, 3, 3, 6	A
A, C, E, F	3, 3, 5, 6, 6	D
A, C, D, E, F	3, 5, 6, 6	B

Rozpoczynając od wierzchołka C, możemy poruszać się w trzech kierunkach, poprzez krawędzie o wagach 1, 2 i 2 (odpowiednio wierzchołki F, A i E). Wybieramy oczywiście F i w tym momencie, z dostępnych dwóch wierzchołków C i F możemy osiągnąć wierzchołki o wagach 2, 2, 3 i 6 (wierzchołka C-F nie wliczamy). Wybieramy według najniższej wagi wierzchołek A i kontynuujemy algorytm, aż do osiągnięcia wszystkich wierzchołków.

Przeszukiwanie grafów

Dużo interesujących zadań algorytmicznych, w których użyto grafu do modelowania pewnej sytuacji, wymaga systematycznego przeszukiwania grafu, ślepego lub kierującego się pewnymi zasadami praktycznymi (tzw. *heurystykami*). W szczególności przeszukiwanie grafów jest przydatne we wszelkich zagadnieniach związanych z tzw. teorią gier, ale do tej kwestii jeszcze powrócimy w rozdziale 12. Teraz skupimy się na dwóch najprostszych technikach przechadzania

się po grafach: strategii „w głąb” (ang. *depth-first search*, oznaczana często DFS) i strategii „wszerz” (ang. *breadth-first search*, oznaczana często BFS). Analizując przykłady, będziemy się koncentrować na samym procesie przeszukiwania, bez zastanawiania się nad jego celowością. Pamiętajmy zatem, że w ostateczności przeszukiwanie grafu ma czemuś służyć: odnalezieniu optymalnej strategii gry, rozwiązaniu łamigłówki lub konkretnego problemu technicznego przedstawionego za pomocą grafów. Algorytmy powinny zawierać zatem nie tylko sam „motor” przeszukiwania, ale i logikę podejmowania decyzji (funkcję celu).



Uwaga

Pokazane dalej przykłady będą używały wyłącznie *reprezentacji tablicowej* grafów. Zabieg ten pozwala na uproszczenie prezentowanych przykładów, ale należy pamiętać, że nie jest to jedyna możliwa reprezentacja! W przypadku algorytmów przeszukiwania pracujących na bardzo dużych grafach użycie tablicy jest niemożliwe. Jedynym wyjściem w takiej sytuacji jest użycie reprezentacji opartej na np. *słowniku węzłów*. Wiąże się z tym modyfikacja wspomnianych algorytmów, zatem dla ułatwienia zostaną one również podane w pseudokodzie, tak aby możliwe było ich przepisanie na użytek konkretnej struktury danych.

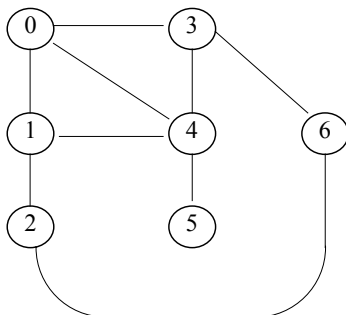
Strategia „w głąb” (przeszukiwanie zstępujące)

Nazwa tytułowej techniki eksploracji grafów jest związana z topologicznym kształtem ścieżek, po których się przechadzamy podczas badania grafu. Algorytm przeszukiwania „w głąb” bada daną drogę aż do jej całkowitego wyczerpania. Jego cechą przewodnią jest zatem *maksymalna eksploatacja raz obranej drogi przed ewentualnym wybraniem następnej*. Strategii „w głąb” dokonuje tzw. ekspansji węzłów, tzn. generuje listę potomków danego węzła i je przetwarza.

Rysunek 10.16 przedstawia niewielki graf, który posłuży nam za ilustrację problemu.

Rysunek 10.16.

Przeszukiwanie grafu „w głąb”



Lista wierzchołków przyległych:

0 - 1, 3, 4
 1 - 0, 2, 4
 2 - 1, 6
 3 - 0, 4, 6
 4 - 0, 1, 3, 5
 5 - 4
 6 - 2, 3

Lista wierzchołków *przyległych* do danego wierzchołka jest dla ułatwienia wypisana obok grafu⁹.

Algorytm przeszukiwania „w głąb” zapisuje się dość prosto w C++:



Listing

depthf.cpp

```

const int n=7;

int G[n][n], V[n]; // G — graf n×n, V — przechowuje informację,
// czy dany wierzchołek był już badany (1) lub nie (0)
void zwiedzaj(int G[n][n], int V[n], int i)
{

```

⁹ Kolejność elementów na tej liście jest związana z użyciem reprezentacji tablicowej, w której indeks tablicy (czyli numer węzła) z góry narzuca pewien porządek wśród węzłów.

```

V[i]=1; //zaznaczamy wierzchołek jako „badany”
cout << "Badam wierzchołek " << i << endl;
    for(int k=0;k<n;k++)
        if(G[i][k]!=0) //istnieje przejście
            if(V[k]==0) zwiedzaj(G,V,k);
    }

void szukaj(int G[n][n], int V[n])
{
    int i;
    for(i=0;i<n;i++)
        V[i]=0; //wierzchołek nie był jeszcze badany
    for(i=0;i<n;i++)
        if(V[i]==0) zwiedzaj(G,V,i);
}

```

Jak łatwo zauważyć, składa się on z dwóch procedur: *szukaj*, która inicjuje sam proces przeszukiwania, i *zwiedzaj*, która tak ukierunkowuje proces przeszukiwania, aby postępował on naprawdę „w głąb”. Procedura *zwiedzaj* przeszukuje listę wierzchołków przylegających do wierzchołka *i*, zatem jej właściwa treść (w pseudokodzie) przedstawia się w ten sposób:

```

zwiedzaj(i)
{
    zaznacz i jako „zbadany”;
    dla każdego wierzchołka k przyległego do i
        jeśli k nie był już zbadany
            zwiedzaj(k)
}

```

W przypadku tej procedury istotne jest oznaczenie zwiedzanego wierzchołka jako „zbadanego”, co eliminuje wielokrotne wywoływania dla tego samego wierzchołka!



Uwaga

W zależności od implementacji informatycznej grafu (tablica, listy lub zbiory węzłów sąsiadujących) można uzyskać nieco inną kolejność odwiedzania węzłów, nie zmienia to jednak samego charakteru tej strategii.

Uruchomienie programu (czyli wykonanie instrukcji *szukaj*(*G*,*V*)) poinformuje nas, że kolejność przeszukiwanych wierzchołków jest następująca: 0, 1, 2, 6, 3, 4 i 5.

Lista wierzchołków przyległych do danego wierzchołka jest dla ułatwienia wypisana obok grafu. Zastanówmy się, czy jest to rzeczywiście przeszukiwanie „w głąb”. Zgodnie z pętlą *for* zawartą w procedurze *szukaj* pierwszym przeszukiwanym wierzchołkiem będzie 0 i on też zostanie jako pierwszy zaznaczony jako zbadany (1). Przylegają do niego trzy wierzchołki: 1, 3, i 4 i dla nich zostanie kolejno wywołana procedura *zwiedzaj* (tym razem rekurencyjnie). Wierzchołek 1 zostaje zaznaczony jako „zbadany”, a następnie badana jest lista wierzchołków przyległych do niego (0, 2 i 4). Ponieważ wierzchołek 0 został już wcześniej przebadany, to następnym będzie 2, dla którego ponownie zostanie wywołana procedura *zwiedzaj*. (Oczywiście zanim to nastąpi, zostanie on zaznaczony jako zbadany). Wierzchołkami przyległymi do 2 są 1 i 6, ale ponieważ 1 został już zbadany, procedura *zwiedzaj* zostanie wywołana tylko dla 6 itd.

Postępując dalej tą drogą, można odtworzyć sposób pracy algorytmu przeszukiwania „w głąb” dla całego grafu. Dla konkretnego zagadnienia przeszukiwania należy oczywiście nasz algorytm wzbogacić o jakąś funkcję porównawczą (funkcję celu).

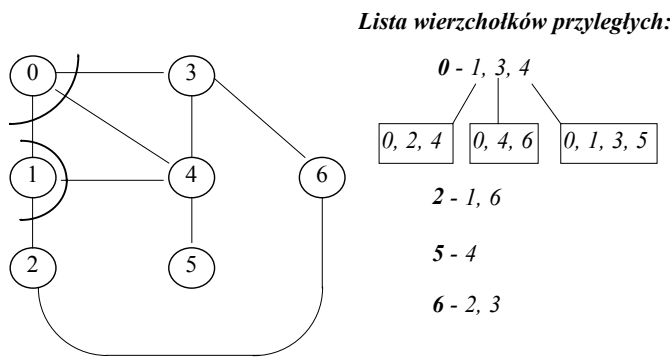
Strategia „wszerz”

Strategia przeszukiwania „wszerz” bada najpierw wszystkie poziomy grafu o jednakowej głębokości.

Do analizy przeszukiwania „wszerz” użyjemy takiego samego grafu, jak w przykładzie poprzednim. Rysunek został jednak uzupełniony o elementy ułatwiające zrozumienie nowej idei przeszukiwania.

Załóżmy, że rozpoczynamy od wierzchołka 0. Na liście wierzchołków przyległych znajdują się kolejno: 1, 3 i 4 i te właśnie wierzchołki zostaną jako pierwsze przebadane podczas przeszukiwania. Dopiero potem algorytm weźmie pod uwagę listy wierzchołków przyległych, wierzchołków już przebadanych: (0, 2, 4), (0, 4, 6) i (0, 1, 3, 5). W konsekwencji kolejność przeszukiwania grafu z rysunku 10.17 będzie taka: 0, 1, 3, 4, 2, 6 i 5.

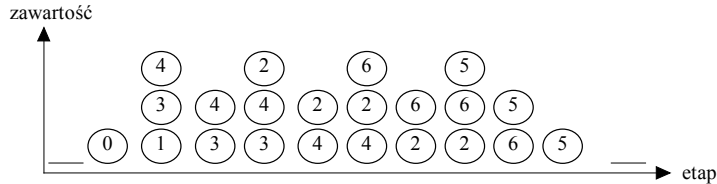
Rysunek 10.17.
Przeszukiwanie grafu „wszerz”



Jak jednak zapamiętać podczas przeszukiwania danego wierzchołka i , że mamy jeszcze ewentualne inne wierzchołki czekające na przebadanie? Okazuje się, że najlepiej jest do tego wykorzystać zwykłą *kolejkę*¹⁰, która sprawiedliwie obsłuży wszystkie wierzchołki, zgodnie z kolejnością ich wchodzenia do kolejki (poczekalni).

Zawartość kolejki dla naszego przykładu przedstawiać się będzie zatem jak na rysunku 10.18.

Rysunek 10.18.
Zawartość kolejki podczas przeszukiwania grafu „wszerz”



Algorytm przeszukiwania „wszerz”, zapisany w C++, przedstawia się następująco:



breadthf.cpp

```
int G[n][n]. V[n];  
// G — graf n×n, V — przechowuje informację, czy dany wierzchołek  
// był już badany (1) lub nie (0)  
#include "kolejka.h"  
  
void szukaj(int G[n][n], int V[n], int i)  
// rozpoczynamy od wierzchołka 'i'  
{  
    FIFO<int> kolejka(n);  
    kolejka.wstaw(i);  
  
    int s;
```

¹⁰ Patrz rozdział 5. i przykład w pliku kolejka.h.

```

while(!kolejka.pusta())
{
    kolejka.obsluz(s); // bierzemy z kolejki pewien wierzchołek 's'
    V[s]=1;           // zaznaczamy wierzchołek 's' jako „badany”

    for(int k=0; k<n; k++)
        if(G[s][k]!=0) // istnieje przejście
            if(V[k]==0) // 'k' nie był jeszcze badany
            {
                V[k]=1; // zaznaczamy wierzchołek 'k' jako „badany”
                kolejka.wstaw(k);
            }
    }
}

```

Sens tego algorytmu może być wyjaśniony znacznie czytelniej w pseudokodzie:

```

szukaj(i)
{
    wstaw i do kolejki;
    dopóki kolejka nie jest pusta wykonuj:
    {
        wyjmij z kolejki pewien wierzchołek s;
        zaznacz s jako „zbadany”;
        dla każdego wierzchołka k przyległego do s
            jeśli k nie był już zbadany
            {
                zaznacz k jako „zbadany”;
                wstaw k do kolejki;
            }
    }
}

```

Na *ftp* znajduje się przykład pokazujący algorytm w działaniu na konkretnym grafie, wraz z instrukcjami kontrolnymi ułatwiającymi jego zrozumienie (program wypisuje na ekranie informację o wstawianych i wyjmowanych elementach).

Inne strategie przeszukiwania

Klasyczne strategie przeszukiwania grafów omówione w poprzednich podpunktach oczywiście nie wyczerpują wszystkich możliwych sposobów odwiedzania węzłów, niemniej stanowią tak naprawdę szablony, które można modyfikować. Oto kilka nieomówionych wcześniej metod:

♦ Strategia z powracaniem

Wariant metody przeszukiwania „w głąb”, w którym zamiast generowania wszystkich potomków badanego węzła generujemy tylko jednego „potomka” i, w przypadku gdy nowy węzeł nie spełnia kryterium celu lub końcowego, jest dalej rozszerzany. Gdy w pewnym momencie otrzymany węzeł spełnia kryterium celu końcowego przeszukiwań grafu lub nie można dla niego wygenerować nowego potomka, to następuje powrót do najbliższego przodka, dla którego jest możliwe ich wygenerowanie.

Strategia z powracaniem eliminuje ryzyko zbędnego generowania węzłów, w klasycznej metodzie w głąb część węzłów otrzymanych w kolejnych krokach może nie być w ogóle testowana w czasie wykonywania algorytmu.

♦ Strategia A*

Celem strategii A* jest wyznaczenie najtańszej drogi w grafie pomiędzy wierzchołkiem początkowym a docelowym (lub docelowymi). Jej składnikiem jest funkcja heurystyczna $f(x) = h(x) + g(x)$, która jest sumą heurystycznej estymacji $h(x)$ kosztu drogi łączącej węzeł x z węzłem celu i $g(x)$ — kosztu drogi łączącej węzeł początkowy z węzłem x . Algorytm A* został opisany w 1968 roku przez Petera Harta, Nilsa Nilssona i Bertrama Raphaela.

♦ Metoda podejżdżania (ang. *hill-climbing*)

Po ekspansji węzłów najbardziej obiecujący z nich jest wybierany do dalszej ekspansji. Przemieszczając się podczas poszukiwań, wykorzystujemy *lokalną* optymalizację i nie zezwalamy na powroty — strategia jest zatem nieodwracalna.

Problem właściwego doboru

Kończąc rozważania dotyczące grafów, pragnę zaprezentować bardzo ciekawe i złożone zagadnienie *problemu doboru* (lub inaczej *minimalizowania konfliktów*). Będzie to kolejny dowód na to, że dobry model ułatwia odnalezienie właściwego rozwiązania.

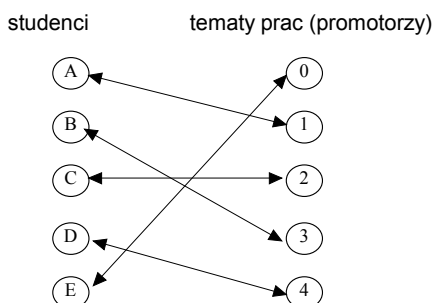
Ponieważ sformułowanie zagadnienia w postaci czysto matematycznej jest bardzo nieczytelne, prześledźmy jego ideę na przykładzie wziętym z życia.

Wyobraźmy sobie następującą sytuację:

- ♦ Mamy N studentów i N tematów prac magisterskich.
- ♦ Do każdej pracy magisterskiej jest przypisany jeden promotor (profesor danej uczelni), zatem z obu stron mamy do czynienia z czynnikiem ludzkim.
- ♦ Każdy student ma pewną opinię (preferencję) na temat danej pracy i z pewnością woli jedne tematy od innych.
- ♦ Również nie każdy profesor lubi jednakowo wszystkich studentów i z pewnością wolałby pracować ze znanym mu studentem X niż z niezbyt dobrze kojarzącym mu się studentem Y , który systematycznie opuszczał jego wykłady.

Oczywiście *problem doboru* nie ogranicza się wyłącznie do kręgów akademickich i może być odnaleziony w przeróżnych postaciach w rozmaitych dziedzinach życia. Dlaczego jest on rozwiązywany za pomocą grafów? Cóż, chyba najlepiej zilustruje to rysunek 10.19.

Rysunek 10.19.
Problem doboru



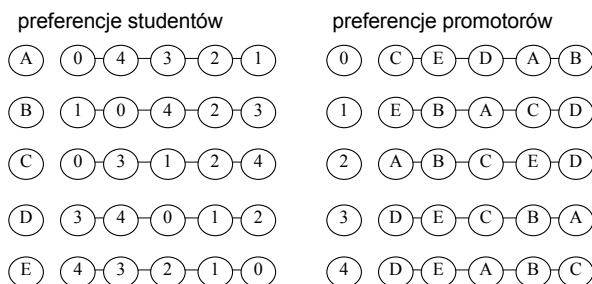
Rysunek przedstawia jedno z możliwych rozwiązań problemu doboru dla $N = 5$ studentów i prac. Zadanie jest przedstawione w postaci specjalnego grafu, w którym węzły są pogrupowane według kategorii i ustawione obok siebie. Warto sobie jednak zdawać sprawę, że taka forma wizualizacji jest przydatna wyłącznie dla człowieka, gdyż komputer nie widzi różnicy pomiędzy ustawieniem „ładnym” i „brzydkim” (struktura graficzna konkretnego doboru jest po prostu grafem, w którym mamy do czynienia z pewną liczbą *par węzłów*). Jeśli węzły i oraz j są ze sobą połączone, oznacza to, że zostały one *dobrane* (nieważne, czy dobrze czy źle). Oznacza to, że niedopuszczalne jest wykorzystanie węzła więcej niż jeden raz.

Analizując problem doboru, stajemy nieuchronnie wobec problemu *wyrażania preferencji*. Każdy student musi mieć opinię o danej pracy i jej promotorze, każdy promotor musi jasno określić swoje preferencje dotyczące określonych osób. Okazuje się, że naturalną metodą są tzw. *listy rankingowe*: opinią studenta X na temat pracy Y będzie jej pozycja na jego liście rankingowej prac magisterskich, podobne listy będą musieli stworzyć profesorowie o studentach.

Omówiona sytuacja jest przedstawiona na rysunku 10.20.

Rysunek 10.20.

Listy rankingowe
w problemie doboru



Nietrudno zauważyć, że o ile samo dobranie N dwójek $\{\text{student}, \text{praca}\}$ jest trywialne, to jednocześnie sprostanie bardzo zróżnicowanym wymaganiom tylu osób nie jest bynajmniej takie proste. Weźmy pod uwagę następującą propozycję: $D-0$, $E-1$, $A-2$, $B-3$, $C-4$. Jest to niewątpliwie jakieś rozwiązanie problemu doboru (bowiem żaden węzeł nie jest wykorzystany więcej niż raz), ale czy na pewno dobre? Student D dostał temat 0 , który na jego liście zajmował dalekie, trzecie miejsce. Zgodnie ze swoimi wymaganiami wołałby on zapewne dostać temat 3 . Temat 3 przypadł jednak studentowi B . Promotor zajmujący się tematem 3 , na swojej liście preferencyjnej umieścił bardzo wysoko studenta D , a tymczasem dostał studenta A ! Mamy więc dość zabawną sytuację:

$D-0$

$B-3$

D woli bardziej 3 od 0

3 woli bardziej D od B

Rozwiązanie zaproponowane powyżej jest zwane *niestabilnym*, gdyż rodzi potencjalne konflikty personalne. Ideałem byłoby znalezienie takiego algorytmu, który proponowałby możliwie najbardziej stabilny wybór, uwzględniający w największym możliwym stopniu dostarczone listy rankingowe. Pamiętając o tzw. czynniku ludzkim, powinno być jasne, dlaczego zadanie nie jest łatwe do rozwiązania: listy rankingowe będą miały po prostu bardzo nierównomierne rozkłady. Pewne tematy będą lubiane przez przeważającą większość, inne znajdą się na szarym końcu. O ile samo dobranie N dwójek wydaje się niekłopotliwe z programistycznego punktu widzenia, to sprawdzenie stabilności wydaje się dość złożone. Kłopot sprawia tu mnogość potencjalnych rozwiązań, z których każde należałoby sprawdzić pod kątem jego stabilności. Zatem algorytm typu *brute-force*, który najpierw losuje potencjalne rozwiązanie (jest ich przecież skończona liczba), a potem sprawdza jego stabilność, byłby bardzo nieefektywny.

Zagadnienie doboru było wszechstronnie studiowane i wydaje się, że zostało znalezione rozwiązanie, które charakteryzuje się pewną inteligencją w porównaniu z bezmyślnym algorytmem typu *brute-force*. Jego idea polega na systematycznym powtarzaniu schematu *cząstkowego doboru*:

- ◆ Student i proponuje temat j , który znajduje się najwyżej na jego liście rankingowej:
 - ◆ Jeśli promotor j nie wybrał jeszcze studenta, to „związek” (i, j) jest *tymczasowo* akceptowany.
 - ◆ Jeśli promotor j zaakceptował już tymczasowo studenta k , to związek (k, j) może zostać złamany na rzecz studenta j , pod warunkiem że promotor lubi bardziej j niż wcześniej wybranego k . W konsekwencji student k znów staje się wolny i w jednym z następnych etapów będzie musiał proponować temat ze swojej listy rankingowej, *następny po uprzednio odrzuconym*.

Używając danych z rysunku 10.20, algorytm mógłby potoczyć się według etapów z tabeli 10.4.

Pora już na omówienie kodu C++, który zajmie się rozwiązaniem problemu właściwego doboru. Jego względna prostota opiera się na wykorzystaniu jedynie tablic liczb całkowitych, dzięki czemu wszelkie manipulacje danymi ulegają maksymalnemu uproszczeniu¹¹.

¹¹ Wszelkie dane liczbowe są zgodne z rysunkiem 10.17.

Tabela 10.4. Problem doboru na przykładzie

Propozycja	Aktualne doборы	Reakcja
A proponuje 0		0 jest wolny i akceptuje A
B proponuje 1	(A, 0)	1 jest wolny i akceptuje B
C proponuje 0	(B, 1)	0 jest zajęty, ale ponieważ woli C od A, to związek (A, 0) jest złamany na rzecz (C, 0)
itd.	(A, 0) (B, 1) (C, 0)	itd.



dobor.cpp

```
#include <iostream>
using namespace std;

int nastepny[5]={-1,-1,-1,-1,-1}; //zapamiętuje ostatni wybór,
// na samym początku nastepny [-1+1] = 0, później posuwamy się
// o 1 pozycję dalej podczas danego etapu wyboru

int dobor[5]={-1,-1,-1,-1,-1}; //rozwiązanie zadania

int wybiera[5][5]={ //preferencje studentów
    {0,4,3,2,1}, /* A */
    {1,0,4,2,3}, /* B */
    {0,3,1,2,4}, /* C */
    {3,4,0,1,2}, /* D */
    {4,3,2,1,0}}; /* E */

//preferencje promotorów: lubi [i][0] = nr A na liście 'i'
// lubi [i][1] = nr B na liście 'i' itd.

int lubi[5][5]={
    /* A B C D E */
    {3,4,0,2,1},
    {2,1,3,4,0},
    {0,1,2,4,3},
    {4,3,2,0,1},
    {2,3,4,0,1}};
```

Algorytm doboru można zamknąć w rozbudowanej funkcji main:

```
int main()
{
    int student, wybierajacy, promotor, odrzucony;

    for(student=0; student<5; student++)
    {
        wybierajacy=student;
        while(wybierajacy!=-1)
        {
            nastepny[wybierajacy]++;
            promotor=wybiera[wybierajacy][nastepny[wybierajacy]];
            if(dobor[promotor]==-1) //promotor (i jego temat) jest wolny
            {
                dobor[promotor]=wybierajacy;
                wybierajacy=-1;
            }
        }
        else
            if(lubi[promotor][wybierajacy]<
                lubi[promotor][dobor[promotor]])
            {
                odrzucony=dobor[promotor];
                dobor[promotor]=wybierajacy;
                wybierajacy=odrzucony;
            }
    }
}
```

```

    }
}

for(int i=0; i<5; i++)
    cout << "(Promotor " << i << ", student " << (char)(dobar[i]+'A')
        << ")\n";
}

```

Spróbujmy przeanalizować pracę programu, ukazując poszczególne wybory dokonywane przez studentów i informując o łamanych związkach:

- ◆ Wybierającym staje się *A* i próbuje on temat (promotora) *0*.
- ◆ Temat (promotor) *0* był wolny i zostaje on przyznany studentowi *A*.
- ◆ Wybierającym staje się *B* i próbuje on temat (promotora) *1*.
- ◆ Temat (promotor) *1* był wolny i zostaje on przyznany studentowi *B*.
- ◆ Wybierającym staje się *C* i próbuje on temat (promotora) *0*.
- ◆ Promotor *0* porzuca swój aktualny wybór *A* na rzecz *C*.
- ◆ Wybierającym staje się porzucony *A* i próbuje on temat (promotora) *4*.
- ◆ Temat (promotor) *4* był wolny i zostaje on przyznany studentowi *A*.
- ◆ Wybierającym staje się *D* i próbuje on temat (promotora) *3*.
- ◆ Temat (promotor) *3* był wolny i zostaje on przyznany studentowi *D*.
- ◆ Wybierającym staje się *E* i próbuje on temat (promotora) *4*.
- ◆ Promotor *4* porzuca swój aktualny wybór *A* na rzecz *E*.
- ◆ Wybierającym staje się *A* i:
 - ◆ Próbuje on temat (promotora) *3*.
 - ◆ Próbuje on temat (promotora) *2*.
- ◆ Temat (promotor) *2* był wolny i zostaje on przyznany studentowi *A*.

Ostateczne wyniki:

```

(Promotor 0, student C)
(Promotor 1, student B)
(Promotor 2, student A)
(Promotor 3, student D)
(Promotor 4, student E)

```

Omówiony algorytm doboru nie jest idealny, gdyż jak łatwo się przekonać, testując go praktycznie, liniowy charakter pętli `for`, która czyni aktywnymi uczestnikami wyłącznie studentów (oni bowiem proponują, a promotorzy czekają biernie na nadchodzące oferty), nie wpływa na sprawiedliwość ostatecznego wyniku. Skomplikowane wersje powyższego algorytmu zmieniają uczestników aktywnych w danym etapie na uczestników biernych i odwrotnie. Powodem prezentacji obecnej wersji była jej prostota i chęć pokazania ciekawej techniki rozwiązywania pozornie złożonych zagadnień.

Podsumowanie

Na tym zakończymy naszą krótką przygodę z grafami. Jak już wspomniałem na początku, poznaliśmy wyłącznie elementy teorii grafów. Liczę jednak na to, że zaprezentowany do tej pory materiał — pomimo że znacznie ograniczony wobec bogactwa istniejących tematów — przyda się znacznej liczbie Czytelników, zachęcając ich być może do sięgnięcia po zacytowaną na początku rozdziału literaturę.

Zadania

Wybór reprezentatywnego dla rekurencji zestawu zadań wcale nie był łatwy dla autora tej książki — dziedzina ta jest bardzo rozległa i w zasadzie wszystko w niej jest w jakiś sposób interesujące. Ostatecznie, co zwykłem podkreślać, zadecydowały względy praktyczne i prostota.

Zadanie 1.

Zastanówmy się, jak używając grafów, zamodelować sieć Internet. Co może być węzłem, a co krawędzią? Dla uproszczenia pomińmy sieci lokalne oraz użytkowników końcowych, a radiolinie i światłowody uznajmy za to samo medium transportowe.

Zadanie 2.

Jak zamodelować, używając słownika węzłów (listy sąsiedztwa), graf *nieskierowany*?

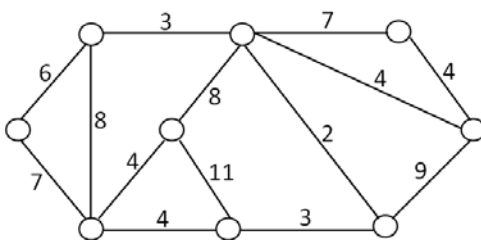
Zadanie 3.

Dla prostego grafu (może to być jeden z pokazanych na rysunkach w tym rozdziale) proszę obliczyć ich zajętość w pamięci komputera dla reprezentacji tablicowej i z użyciem list. Należy przyjąć pewne umowne miary, np. jednostkami mogą być węzły i krawędzie, oraz uwzględnić ewentualne wskaźniki i wszelkie struktury pomocnicze, jeśli takie okażą się konieczne.

Zadanie 4.

Dla grafu z rysunku 10.21 należy zaaplikować algorytm Kruskala i Prima.

Rysunek 10.21.
Drzewo rozpinające
— ćwiczenie



Rozdział 11.

Algorytmy numeryczne

Przez dziesiątki lat pierwszym i głównym zastosowaniem komputerów było szybkie dokonywanie obliczeń (do dziś dla wielu ludzi słowa „komputer” i „kalkulator” są synonimami). Dziedzina tych zastosowań pozostaje ciągle aktualna, lecz należy zdawać sobie sprawę, że wielokrotne wymyślanie tych samych rozwiązań ma znikomy sens praktyczny. W minionych latach powstała cała gama gotowych programów, potrafiących rozwiązywać typowe problemy matematyczne (np. rozwiązywanie układów równań, interpolacja i aproksymacja, całkowanie i różniczkowanie, przekształcenia symboliczne itd.) i osobie szukającej wyrafinowanych możliwości można polecić zakup takiego narzędzia, np. Matlab (<http://www.mathworks.com>), Mathcad (<http://www.mathsoft.com>), Mathematica (<http://www.wolfram.com>). Pakiety te, czasami występujące także w wersjach darmowych, mogą nie tylko automatyzować wykonywanie obliczeń, ale dodatkowo współpracują z innymi językami programowania i pozwalają na graficzne prezentowanie wyników.

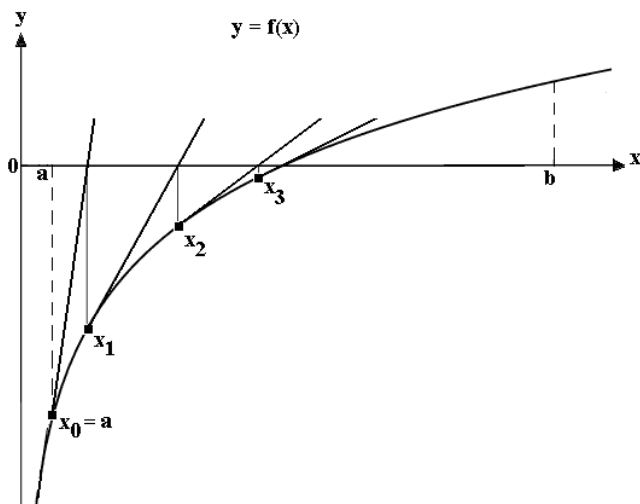
Celem tego rozdziału będzie ukazanie kilku przydatnych metod z dziedziny algorytmów numerycznych, takich, które potencjalnie mogą znaleźć zastosowanie jako część większych projektów programistycznych. Nie będę się zbyt koncentrował na matematycznych uzasadnieniach prezentowanych programów, ale postaram się pokazać, jak algorytm numeryczny daje się przetłumaczyć na gotowy do użycia kod C++. Algorytmy cytowane w tym rozdziale zostały opracowane głównie na podstawie prac: dostępnego w Polsce skryptu [Kla87] oraz klasycznego dzieła [Knu69], ale Czytelnik nie powinien mieć trudności z dotarciem do innych podręczników poruszających tematykę algorytmów numerycznych, gdyż powstało ich dość sporo w ostatnich latach. Wszystkie prezentowane w tym rozdziale programy należy traktować jako implementacje „edukacyjne”, pokazujące wykorzystanie języka C++ do rozwiązywania problemów obliczeniowych, ale mniej wymagający użytkownik, potrzebujący konkretnej procedury, może z nich korzystać jak ze swego rodzaju „książki kucharskiej”.

Poszukiwanie miejsc zerowych funkcji

Jednym z częstych problemów, z jakimi mają do czynienia matematycy, jest poszukiwanie miejsc zerowych funkcji. Metod numerycznych, które umożliwiają rozwiązanie takiego zadania za pomocą komputera, jest dość sporo, my ograniczymy się do jednej z prostszych — do tzw. *metody Newtona*. W skrócie polega ona na systematycznym przybliżaniu się do miejsca zerowego za pomocą stycznych do krzywej, tak jak to pokazuje rysunek 11.1.

Metoda Newtona zakłada szereg ograniczeń na badaną funkcję $y = f(x)$, np. w analizowanym przedziale $[a, b]$ znajduje się dokładnie jeden pierwiastek, funkcja na krańcach przedziału przyjmuje wartości różniące się znakiem, a jej pierwsza i druga pochodna mają stały znak w tym przedziale.

Rysunek 11.1.
Algorytm Newtona
odszukiwania
miejsc zerowych



Z punktu widzenia programisty algorytm *Newtona* sprowadza się do iteracyjnego powtarzania następującego algorytmu (i oznacza etap iteracji):

- ◆ $z_i = z_{i-1} - \frac{f(z_{i-1})}{f'(z_{i-1})};$
- ◆ stop, jeśli $|f(z_i)| < \varepsilon.$

Symbol ε oznacza pewną stałą dodatnią (np. $0,00001$), gwarantującą zatrzymanie algorytmu. Oczywiście na samym początku inicjujemy z_0 pewną wartością początkową, musimy ponadto znać jawnie równania f i f' (funkcji i jej pierwszej pochodnej)¹.

Zaproponujemy rekurencyjną² wersję algorytmu, który przyjmuje jako parametry m.in. wskaźniki do funkcji reprezentujących f i f' .

Popatrzmy dla przykładu na obliczanie za pomocą metody *Newtona* zera funkcji $f(x) = 3x^2 - 2$. Procedura zero jest dokładnym tłumaczeniem podanego na samym początku wzoru:



newton.cpp

```
const double epsilon=0.0001;
double f(double x)    //funkcja f(x) = 3x^2-2
{
    return 3*x*x-2;
}
double fp(double x)   //pochodna f'(x) = (3x^2-2)' = 6x
{
    return 6*x;
}
double zero(double x0, double(*f)(double), double(*fp)(double) )
{

```

¹ Musimy je wpisać do kodu programu w C++ „na sztywno”.

² Zauważmy, że mamy tu do czynienia z przykładem tzw. rekurencji końcowej, którą w naturalny sposób przekształcić można na równoważną algorytmicznie wersję iteracyjną.

```

if(fabs(f(x0))<epsilon)
    return x0;
else
    return zero(x0-f(x0)/fp(x0),f,fp);
}
int main()
{
    cout << "Zero funkcji 3x*x-2 wynosi " << zero(1,f,fp) <<endl;
} //wynik 0,816497

```

Użycie wskaźników do funkcji pozwala uczynić procedurę `zero` bardziej uniwersalną, ale oczywiście nic nie stoi na przeszkodzie, aby używać tych funkcji w sposób bezpośredni.

Iteracyjne obliczanie wartości funkcji

Jak efektywnie obliczać wartość wielomianów, dowiemy się szczegółowo w rozdziale 13., przy okazji omawiania tzw. *schematu Hornera*. Obecnie zajmiemy się dość rzadko używanym w praktyce, ale czasami użytecznym algorytmem iteracyjnego obliczania wartości funkcji.

Załóżmy, że interesuje nas pewna funkcja $y = f(x)$. Przekształćmy ją w tzw. postać uwikłaną, czyli zapiszmy w postaci:

$$F(x, y) = 0.$$

Oznaczmy pochodną cząstkową, liczoną względem zmiennej y przez $F'_y(x, y)$, przy czym $F'_y(x, y) \neq 0$ w każdej iteracji.

Przyjmując pewne uproszczenia, można za pomocą metody *Newtona* obliczyć jej wartość dla pewnego x w sposób iteracyjny:

$$\begin{aligned} \blacklozenge \quad y_{n+1} &= y_n - \frac{F(x, y_n)}{\frac{\partial F}{\partial y}(x, y)|_{y_n}}, \\ \blacklozenge \quad \text{stop, jeśli } |y_{n+1} - y_n| &< \varepsilon. \end{aligned}$$

Wartość początkowa y_0 powinna być jak najbliższa wartości poszukiwanej y .

W opisywanej metodzie problem obliczania wartości funkcji $f(x)$ sprowadzony został do poszukiwania miejsca zerowego funkcji $F(x, y)$!

Zalety metody *Newtona* szczególnie uwidaczniają się w przypadku niektórych funkcji, gdzie iloraz może (ale nie musi) znacznie się uprościć.

Przykład: dla $y = \frac{1}{x}$ mamy $F(x, y) = x - \frac{1}{y}$ oraz $F'_y(x, y) = \frac{1}{y^2}$. Z warunku $F(x, y) = 0$ otrzymujemy wzór iteracyjny $y_{n+1} = 2y_n - x(y_n)^2$.

Powyższe wzory przekładają się na program C++ w następujący sposób:



wartf.cpp

```

const double epsilon=0.00000001;

double wart(double x, double yn)
{
    double yn1=2*yn-x*yn*yn;

```

```

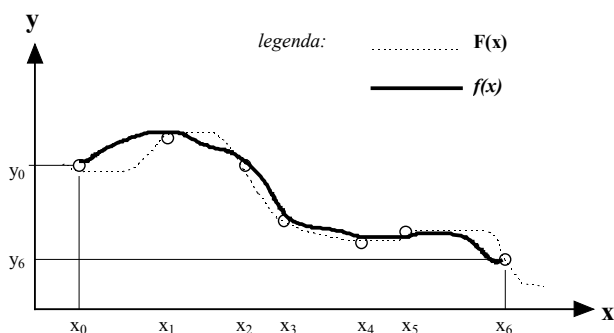
if( fabs(yn-yn1)<epsilon) //fabs = wartość absolutna
    return yn1;
else
    return wart(x,yn1);
}
int main()
{
    cout << "Wartość funkcji y=1/x dla x=7 wynosi " << wart(7,0.1);
}
//wynik: 14.2857

```

Interpolacja funkcji metodą Lagrange'a

W poprzednich paragrafach tego rozdziału bardzo często korzystaliśmy jawnie z wzorów funkcji i jej pochodnej. Cóż jednak począć, gdy dysponujemy fragmentem wykresu funkcji (tzn. znamy jej wartości dla skończonego zbioru argumentów) lub też wyliczanie na podstawie wzorów byłoby zbyt czasochłonne z uwagi na ich skomplikowaną postać? Na pomoc, w obu przypadkach, przychodzą tzw. metody *interpolacji funkcji*, tzn. przybliżania jej za pomocą prostej funkcji (np. wielomianu określonego stopnia), tak aby funkcja interpolacyjna przechodziła dokładnie przez znane nam punkty wykresu funkcji jak na rysunku 11.2.

Rysunek 11.2.
Interpolacja funkcji $f(x)$
za pomocą wielomianu
 $F(x)$



W zobrazowanym na nim przykładzie dysponujemy 7 parami $(x_0, y_0) \dots (x_6, y_6)$ i na tej podstawie udało nam się obliczyć wielomian $F(x)$, dzięki któremu obliczanie wartości $f(x)$ staje się o wiele prostsze (choć czasami wyniki mogą być dalekie od prawdy).

Wielomian interpolacyjny konstruuje się za pomocą kłopotliwego obliczeniowo wyznacznika *Vandermonde'a*, który pozwala na wyliczenie współczynników poszukiwanego wielomianu³. Jeśli jednak zależy nam tylko na wartości funkcji w pewnym punkcie z , to istnieje prostsza i efektywniejsza metoda *Lagrange'a*:

$$F(z) = (z - x_0)(z - x_1) \dots (z - x_n) \sum_{j=0}^n \frac{y_j}{(z - x_j) \prod_{i=0, i \neq j}^n (x_j - x_i)}.$$

Pomimo dość makabrycznej postaci wzór powyższy tłumaczy się bezpośrednio na kod C++ za pomocą dwóch zagnieżdżonych pętli *for*.

Przedstawiony poniżej listing niesie ze sobą jednak ryzyko wystąpienia dzielenia przez zero (jeśli wartość z jest równa któremuś z węzłów). W ramach ćwiczenia proponuję rozbudować poniższą funkcję, wzbogacając ją o kontrolę wyliczanych wartości i odpowiednią sygnalizację błędów.

³ Spójrz np. http://pl.wikipedia.org/wiki/Macierz_Vandermonde'a.



interpol.cpp

```
const int n=3; // stopień wielomianu interpolującego
// tabela wartości funkcji y[i] = f(x[i]):
double x[n+1]={3.0, 5.0, 6.0, 7.0};
double y[n+1]={1.732, 2.236, 2.449, 2.646};
// (jest to w istocie funkcja obliczająca pierwiastek z wartości 'x'
double interpol(double z, double x[n], double y[n])
{ // zwraca wartość funkcji w punkcie 'z'
  double wnz=0, om=1, w;
  for(int i=0; i<=n; i++)
  {
    om=om*(z-x[i]);
    w=1.0;
    for(int j=0; j<=n; j++)
      if(i!=j) w=w*(x[i]-x[j]);
      wnz=wnz+y[i]/(w*(z-x[i]));
  }
  return wnz=wnz*om;
}
int main()
{
  double z=4.5;
  cout << "Wartość funkcji sqrt(x) w punkcie " << z << " wynosi " <<
    interpol(z,x,y) <<endl;
}
```

Różniczkowanie funkcji

W poprzednich paragrafach tego rozdziału bardzo często korzystaliśmy ze wzorów funkcji i jej pochodnej wpisanych wprost w kod C++. Czasami jednak obliczenie pochodnej może być kłopotliwe i pracochłonne, przydają się wówczas metody, które radzą sobie z tym problemem bez potrzeby korzystania z jawnego wzoru funkcji.

Jedną z popularniejszych metod różniczkowania numerycznego jest tzw. wzór *Stirlinga*. Jego wyprowadzenie leży poza zakresem tej publikacji, dlatego zdecydowałem się zademonstrować jedynie rezultaty praktyczne, nie wnikając w uzasadnienie matematyczne.

Wzór *Stirlinga* pozwala w prosty sposób obliczyć pochodne f' i f'' w punkcie x_0 , dla pewnej funkcji $f(x)$, której wartości znamy w postaci tabelarycznej:

$$\dots(x_0-2h, f(x_0-2h)), (x_0-h, f(x_0-h)), (x_0, f(x_0)), (x_0+h, f(x_0+h)), (x_0+2h, f(x_0+2h))\dots$$

Parametr h jest pewnym stałym krokiem w dziedzinie wartości x .

Metoda *Stirlinga* wykorzystuje tzw. *tablicę różnic centralnych*, której konstrukcję przedstawia rysunek 11.3.

Różnice δ są obliczane w identyczny sposób w całej tabeli, np. :

$$\delta f(x_0 - \frac{3}{2}h) = f(x_0 - h) - f(x_0 - 2h) \text{ itd.}$$

Przyjmując upraszczające założenie, że zawsze będziemy obliczali pochodne dla punktu centralnego $x = x_0$, wzory *Stirlinga* przyjmują następującą postać:

$$f'(x) = \frac{1}{h} \left(\frac{\delta f(x - \frac{1}{2}h) + \delta f(x + \frac{1}{2}h)}{2} - \frac{1}{6} \frac{\delta^3 f(x - \frac{1}{2}h) + \delta^3 f(x + \frac{1}{2}h)}{2} + \frac{1}{30} \frac{\delta^5 f(x - \frac{1}{2}h) + \delta^5 f(x + \frac{1}{2}h)}{2} + \dots \right)$$

$$f''(x) = \frac{1}{h^2} \left(\delta^2 f(x) + \frac{1}{12} \delta^4 f(x) - \frac{1}{90} \delta^6 f(x) + \dots \right)$$

Rysunek 11.3.
Tablica różnic
centralnych
w metodzie Stirlinga

x	$f(x)$	$\delta f(x)$	$\delta^2 f(x)$	$\delta^3 f(x)$	$\delta^4 f(x)$
x_0-2h	$f(x_0-2h)$				
	—	$\delta f(x_0 - \frac{3}{2}h)$			
x_0-h	$f(x_0-h)$		$\delta^2 f(x_0 - h)$		
		$\delta f(x_0 - \frac{1}{2}h)$		$\delta^3 f(x_0 - \frac{1}{2}h)$	
x_0	$f(x_0)$		$\delta^2 f(x_0)$		$\delta^4 f(x_0)$
		$\delta f(x_0 + \frac{1}{2}h)$		$\delta^3 f(x_0 + \frac{1}{2}h)$	
x_0+h	$f(x_0+h)$		$\delta^2 f(x_0 + h)$		
		$\delta f(x_0 + \frac{3}{2}h)$			
x_0+2h	$f(x_0+2h)$				

Punktów kontrolnych funkcji może być oczywiście znacznie więcej niż 5. Tutaj skoncentrujemy się na bardzo prostym przykładzie z pięcioma wartościami funkcji, co prowadzi do tablicy różnic centralnych niskiego rzędu.

Wzorcowy program w C++, wyliczający pochodne dla pewnej funkcji $f(x)$, może wyglądać następująco:



pochodna.cpp

```
const int n=5; // rząd obliczanych różnic centralnych wynosi n-1

double t[n][n+1]=
{
{0.8, 4.80}, // pary (x[i], y[i]) dla y = 5x^2+2x
{0.9, 5.85}, // (wpisane są dwie pierwsze kolumny, a nie wiersze)
{1, 7.00},
{1.1, 8.25},
{1.2, 9.60}
};

struct POCHODNE{double f1,f2;};

POCHODNE stirling(double t[n][n+1])
// funkcja zwraca wartości f'(z) i f''(z), gdzie 'z' jest elementem
// centralnym: tutaj t[2][0]; tablica 't' musi być uprzednio centralnie
// zainicjowana, jej poprawność nie jest sprawdzana
{
POCHODNE res;
double h=(t[4][0]-t[0][0])/((double)(n-1)); // krok argumentu 'x'
for(int j=2;j<=n;j++)
for(int i=0;i<=n-j;i++)
t[i][j]=t[i+1][j-1]-t[i][j-1];
res.f1=((t[1][2]+t[2][2])/2.0-(t[0][4]+t[1][4])/12.0)/h;
res.f2=(t[1][3]-t[0][5])/12.0/(h*h);
return res;
}

int main()
{
POCHODNE res=stirling(t);
cout << "f'=" << res.f1 << ", f''=" << res.f2 << endl;
}
```

Jeśli już omawiamy różniczkowanie numeryczne, to warto podkreślić związaną z nim dość niską dokładność. Im mniejsza wartość parametru h , tym większy wpływ na wynik mają błędy zaokrągleń, z kolei zwiększenie h jest niezgodne z ideą metody *Stirlinga* (która ma przecież przybliżać *prawdziwe* różniczkowanie!). Metoda *Stirlinga* nie jest odpowiednia dla różniczkowania na krańcach przedziałów zmienności argumentu funkcji. Zainteresowanych tematem zapraszam zatem do studiowania właściwej literatury przedmiotu, wiedząc, że temat jest bogatszy, niż się to wydaje.

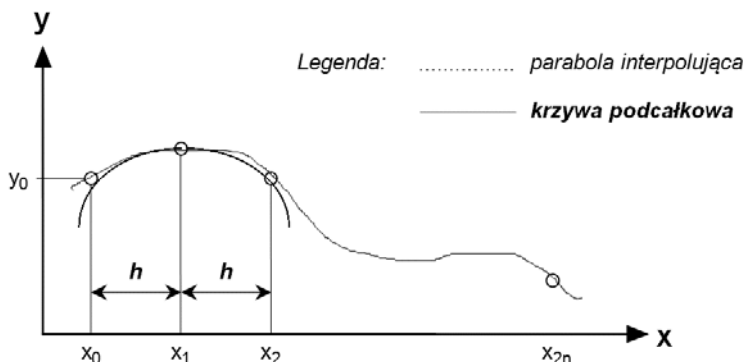
Całkowanie funkcji metodą Simpsona

Całkowanie niektórych funkcji może być niekiedy skomplikowane z uwagi na trudność obliczenia symbolicznego całki danej funkcji. Czasami trzeba wykonać dość sporo niełatwych przekształceń (np. podstawienia, rozkład na szeregi itp.), aby otrzymać pożądany rezultat.

Z pomocą przychodzą tu jednak metody interpolacji (czyli przedstawiania skomplikowanej funkcji w prostszej obliczeniowo, przybliżonej postaci). Ideę całkowania numerycznego przedstawia rysunek 11.4.

Rysunek 11.4.

Przybliżone
całkowanie funkcji



Na danym etapie i trzy kolejne punkty funkcji podcałkowej są przybliżane parabolą, co zapewnia dość dobrą dokładność całkowania (dla niektórych krzywych wyniki mogą być wręcz identyczne z tymi otrzymanymi z całkowania na kartce papieru). Dla rozpatrywanego fragmentu całka cząstkowa wyniesie:

$$\int_{x_0}^{x_2} f(x) dx = \frac{f(x_0) + 4f(x_1) + f(x_2)}{3h}.$$

Wzór powyższy, zwany wzorem *Simpsona*, wystarczy zastosować dla każdego przedziału całkowanego obszaru, złożonego z 3 kolejnych punktów krzywej $f(x)$. Jedynym wymogiem jest takie dobranie odstępów h , aby były one jednakowe. Zakładając zatem całkowania od a do b , przy podziale na $2n$ odcinków będziemy mieli $h = (b-a)/2n$. Całka globalna będzie, oczywiście, sumą całek cząstkowych, obliczonych jak niżej:



simpson.cpp

```
const int n=4; // liczba punktów = 2n+1

// funkcja x^2-3x+1 w przedziale [-5, 3]
double f[2*n+1]={41, 29, 19, 11, 5, 1, -1, -1, 1};
```

```
double simpson(double f[2*n+1], double a, double b)
// funkcja zwraca całkę funkcji f(x) w przedziale [a, b],
// której wartości są podane tabelarycznie w 2n+1 punktach
{
    double s=0, h=(b-a)/(2.0*n);
    for(int i=0; i<2*n; i+=2) // skok co dwa punkty!
        s+=h*(f[i]+4*f[i+1]+f[i+2])/3.0;
    return s;
}
```

Oczywiście całkowanie metodą Simpsona można również zastosować do scałkowania funkcji znanej w postaci analitycznej, a nie tylko tabelarycznej, trzeba tylko przekazać odpowiedni wskaźnik do funkcji rozpisanej w wersji analitycznej:

```
double fun(double x)
{
    return x*x-3*x+1;
}

double simpson_f(double(*f)(double), double a, double b, int N)
// funkcja zwraca całkę znanej w postaci wzoru funkcji f(x)
// w przedziale [a, b], N – liczba podziałów
{
    double s=0, h=(b-a)/(double)N;
    for(int i=1; i<=N; i++)
        s+=h*(f(a+(i-1)*h)+4*f(a-h/2.0+i*h)+f(a+i*h))/6.0;
    return s;
}
```

Poniżej podaję sposób wywołania obu wariantów funkcji „simpson”:

```
int main()
{
    cout << "Wartość całki =" << simpson(f,-5,3) << endl; // 82.667
    cout << "Wartość całki =" << simpson_f(fun,-5,3,8) << endl; // 82.667
}
```

Rozwiązywanie układów równań liniowych metodą Gaussa

Potrzeba rozwiązywania układów równań liniowych zachodzi w wielu dziedzinach, szczególnie technicznych. Biorąc pod uwagę, że w samym rozwiązywaniu układów równań nie ma nic odkrywczego (uczono nas już tego w szkole podstawowej!), cenne wydaje się dysponowanie procedurą komputerową, która wykona za nas tę żmudną pracę.

Aby komputer mógł rozwiązać dany układ równań, musimy go uprzednio zapisać w postaci rozszerzonej, tzn. nie eliminując współczynników równych zero i pisząc zmienne w określonej kolejności. To wszystko ma na celu prawidłowe skonstruowane macierzy rozszerzonej układu.

Układ równań:

$$\begin{aligned} 5x+z &= 9 \\ x-z+y &= 6 \\ 2x-y+z &= 0 \end{aligned}$$

musi zatem zostać przedstawiony jako:

$$\begin{aligned}5x + 0y + 1z &= 9 \\ 1x + 1y - 1z &= 6 \\ 2x - 1y + 1z &= 0\end{aligned}$$

co pozwoli na zapisanie całości w postaci macierzowej:

$$\begin{pmatrix} 5 & 0 & 1 \\ 1 & 1 & -1 \\ 2 & -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 9 \\ 6 \\ 0 \end{pmatrix}.$$

Wymnożenie tych macierzy powinno spowodować powrót do klasycznej, czytelnej postaci.

Zaletą reprezentacji macierzowej jest możliwość zapisania wszystkich współczynników liczbowych w jednej tablicy $N \times (N+1)$ i operowania nimi podczas rozwiązywania układu. Operacje na tej macierzy będą odbiciem przekształceń dokonywanych na równaniach (np. w celu eliminacji zmiennych, dodawania równań stronami itd.).

Z uwagi na łatwość implementacji programowej bardzo szeroko rozpowszechnioną metodą rozwiązywania układów równań liniowych jest tzw. *eliminacja Gaussa*. Przebiega ona zasadniczo w dwóch etapach: sprowadzania macierzy układu do tzw. *macierzy trójkątnej*, wypełnionej zerami poniżej przekątnej, oraz *redukcji wstecznej*, mającej na celu wyliczanie wartości poszukiwanych zmiennych:

- ♦ W pierwszym etapie eliminujemy zmienną x z wszystkich oprócz pierwszego wiersza (poprzez klasyczne dodawanie wiersza bieżącego, pomnożonego przez współczynnik, który spowoduje eliminację).
- ♦ W etapie drugim postępujemy identycznie ze zmienną y i wierszem 2. w celu ostatecznego otrzymania macierzy trójkątnej.

Popatrzmy na przykładzie:

- ♦ eliminacja x z wierszy 2. i 3. (efekt dodawania wierszy jest pokazany w etapie następnym):

$$\begin{array}{l} \begin{array}{l} 5x + 0y + 1z = 9 \\ 1x + 1y - 1z = 6 \\ 2x - 1y + 1z = 0 \end{array} \end{array} \begin{array}{l} \xrightarrow{*(-0,2)} \\ \xleftarrow{*(-0,4)} \end{array}$$

- ♦ eliminacja y z wierszy 1. i 3. (w pierwszym nie ma już nic do zrobienia):

$$\begin{array}{l} 5x + 0y + 1z = 9 \\ 0x + 1y - 1,2z = 4,2 \\ 0x - 1y + 0,6z = -3,6 \end{array} \begin{array}{l} \\ \xleftarrow{*1} \end{array}$$

$$\begin{array}{l} 5x + 0y + 1z = 9 \\ 0x + 1y - 1,2z = 4,2 \\ 0x - 1y + 0,6z = -3,6 \end{array} \begin{array}{l} \\ \xleftarrow{*1} \end{array}$$

- ♦ otrzymujemy ostatecznie macierz *trójkątną*:

$$\begin{array}{l} 5x + 0y + 1z = 9 \\ 0x + 1y - 1,2z = 4,2 \\ -0x + 0y - 0,6z = 0,6 \end{array}$$

Mając macierz w takiej postaci, można już pokusić się o wyliczenie zmiennych (*redukcja wsteczna*, idziemy od ostatniego do pierwszego wiersza układu):

$$\begin{aligned}z &= -0,6/0,6 = -1 \\ y &= 1,2z + 4,2 = 3 \\ x &= (9 - z)/5 = 2\end{aligned}$$

Metoda nie jest zatem skomplikowana, choć jej zapis w C++ może się wydać początkowo nieczytelny. Jediną niebezpieczną operacją metody eliminacji *Gausa* jest... eliminacja zmiennych, która czasami może prowadzić do dzielenia przez zero (jeśli na etapie *i* eliminowana zmienna w danym równaniu nie występuje). Biorąc jednak pod uwagę, że zamiana wierszy miejscami nie wpływa na rozwiązanie układu, niebezpieczeństwo dzielenia przez zero może być łatwo oddalone poprzez taki właśnie wybieg.

Oczywiście zamiana wierszy może okazać się niemożliwa ze względu na niespełnienie warunku, jakim jest znalezienie poniżej wiersza *i* takiego wiersza, który ma konfliktową zmienną różną od zera. W takim przypadku układ równań nie ma rozwiązania, co też jest pewną informacją dla użytkownika!

Oto pełna treść programu wykonującego eliminację *Gausa*⁴, wraz z danymi przykładowymi:



Listing

gauss.cpp

```
const int N=3;
double x[N]; //wyniki

double a[N][N+1]=
{
    {5 , 0, 1, 9},
    {1 , 1,-1, 6},
    {2, -1, 1, 0}
};

int gauss(double a[N][N+1], double x[N])
{
    int max;
    double tmp;

    for(int i=0; i<N; i++) // eliminacja
    {
        max=i;
        for(int j=i+1; j<N; j++)
            if(fabs(a[j][i])>fabs(a[max][i])) //fabs = wartość bezwzględna
                max=j;
        for(int k=i; k<N+1; k++) // zamiana wierszy wartościami
        {
            tmp=a[i][k];
            a[i][k]=a[max][k];
            a[max][k]=tmp;
        }
        if(a[i][i]==0)
            return 0; // Układ sprzeczny!
        for(int j=i+1; j<N; j++)
            for(int k=N; k>=i; k--)
                // mnożenie wiersza j przez współczynnik „zerujący”:
                a[j][k]=a[j][k]-a[i][k]*a[j][i]/a[i][i];
    }
    for(int j=N-1; j>=0; j--) // redukcja wsteczna
    {
        tmp=0;
        for(int k=j+1; k<=N; k++)
            tmp=tmp+a[j][k]*x[k];
        x[j]=(a[j][N]-tmp)/a[j][j];
    }
    return 1; // wszystko w porządku!
}
```

⁴ http://www.algorytm.org/index.php?option=com_content&task=view&id=82&Itemid=28

```
int main()
{
    if(!gauss(a,x))
        cout << "Układ (1) jest sprzeczny!\n";
    else
    {
        cout << "Rozwiązanie:\n";
        for(int i=0;i<N;i++)
            cout << "x["<<i<<"]="<<x[i] << endl;
    }
}
```

Uwagi końcowe

W tym krótkim rozdziale nie mogłem poruszyć wielu zagadnień z dziedziny obliczeń numerycznych, jednak przedstawione zestawienie zawiera z pewnością wybór najczęściej używanych w praktyce programów. Uwagi zawarte na jego wstępie pozostają aktualne, warto jednak wspomnieć, że implementowanie algorytmów numerycznych z użyciem C++ jest czasami robione nieco na siłę, gdyż język ten nie wspomaga w bezpośredni sposób modelowania zagadnień natury czysto obliczeniowej. Matematykom i fizykom potrzebującym sprawnych narzędzi obliczeniowych można polecić w jego miejsce którąś z nowoczesnych implementacji *Fortranu*. Chociaż nie jest on językiem ogólnego zastosowania (w odróżnieniu od np. C++ i Pascala), wraz z jego kompilatorem są zazwyczaj dostarczane bardzo bogate biblioteki procedur obliczeniowych (odwracanie macierzy, całkowanie, interpolacja itd.) — te wszystkie procedury, które programista C++ musi zwykle pisać od zera. Oczywiście do języka C++ także są dostępne biblioteki numeryczne (patrz np. GNU Scientific Library — <http://www.gnu.org/software/gsl>).

Rozdział 12.

Czy komputery mogą myśleć?

Umieszczenie w podręczniku algorytmiki o charakterze ogólnym rozdziału poświęconego dziedzinie zwanej dość mylnie „sztuczną inteligencją” wiąże się z całym szeregiem niebezpieczeństw. Przede wszystkim jest to dziedzina tak ogromnie rozległa, iż trudno się pokusić o stworzenie jakiegoś dobrego streszczenia opisującego zagadnienie bez zbytnich uproszczeń. Jest to wręcz niemożliwe. Do worka zwanego sztuczną inteligencją wrzuca się sporo bardzo różnych dziedzin: teorię gier, planowanie, algorytmy ewolucyjne (wyszukujące najlepsze rozwiązania na zasadzie zbliżonej do ewolucji biologicznej), systemy eksperckie, etc. Po drugie, zagadnienia związane ze sztuczną inteligencją są na ogół dość trudne i trzeba naprawdę wiele wysiłku, aby uczynić z nich temat frapujący. Udało się to bez wątpienia Nilssonowi w [Nil82], lecz on przeznaczył na to zadanie kilkaset stron!

Mój dylemat polegał więc na wyborze kilku interesujących przykładów i na opisaniu ich na tyle prostym językiem, aby osoba bez wykształcenia informatycznego nie miał zbytnich kłopotów z ich zrozumieniem. Wybór padł na elementy teorii gier. Temat ten wiąże się z odwiecznym marzeniem człowieka, aby znaleźć optymalną strategię dla danej gry, pozwalającą na pewne jej wygranie.

Pytanie zawarte w tytule rozdziału jest bardzo bliskie powszechnym odczuciom komputerowych laików. Fakt, że jakaś maszyna potrafi grać, rysować, animować jest dla nich jednoznaczny z myśleniem. „Oczywiście jest to błędne przekonanie”, powie informatyk, który wie, że w istocie komputery są wyłącznie skomplikowanymi automatami, z możliwościami zależnymi od programów, w które je wyposażymy. W tych właśnie programach tkwi możliwość *symulowania inteligentnego zachowania* komputera, zbliżającego jego sposób postępowania do ludzkiego. W chwili obecnej potrafimy jedynie kazać komputerowi *naśladować* zachowania inteligentne, gdyż stopień skomplikowania ludzkiego mózgu¹ przewyższa najbardziej nawet złożony komputer. Pamiętajmy jednak, że wcale nie jest powiedziane, iż za kilka lat nie powstanie technologia, która pozwoli skonstruować ideowy odpowiednik ludzkiego mózgu i nauczyć go rozwiązywania problemów niedostępnych nawet dla człowieka!

Proszę zatem traktować ten rozdział jedynie jako zachęający wstęp do dalszego studiowania bardzo rozległej dziedziny sztucznej inteligencji. Bardzo polecam lekturę [Nil82], na rynku polskim jest również dostępne ciekawe opracowanie [BC89] dotyczące metod przeszukiwania, odgrywających tak istotną rolę w dziedzinie sztucznej inteligencji. Warto też polecić [Kas03], książkę wykraczającą poza tradycyjne opisy zagadnień sztucznej inteligencji, chociażby dzięki bogatej faktografii poświęconej historycznemu rozwojowi badań nad sztuczną inteligencją, zawierającą wiele ciekawych odnośników do bogatej bibliografii przedmiotu.

¹ Na dodatek zasada działania ludzkiego mózgu wcale nie jest do końca rozumiana przez współczesną naukę.

Przegląd obszarów zainteresowań Sztucznej Inteligencji

Sztuczna Inteligencja (nazwijmy ją *SI*) jest dziedziną wiedzy, która postawiła sobie za zadanie badania nad zagadnieniem niemalże filozoficznym, związanym z odwiecznym poszukiwaniem odpowiedzi na pytanie zadane w tytule tego rozdziału: czy komputery mogą myśleć? W 1950 roku² Alan Turing zaproponował test, który miał na celu odpowiedzieć na to pytanie. W systemie opartym na dialogach komputer występuje *jako jeden z graczy*. W sytuacji, w której odpowiedzi gracza-komputera będą nierozpoznawalne od odpowiedzi gracza-człowieka, można uznać, że jest on inteligentny. Sam test doczekał się nawet nagrody w wysokości 100 tys. dolarów dla twórcy pierwszego komputera, którego odpowiedzi będą nierozpoznawalne od odpowiedzi człowieka (nagroda Loebnera, kontestowana przez niektóre autorytety w branży *SI*).

Załóżmy jednak, że nawet gdyby komputery potrafiły choćby tylko udawać myślenie, to czy nie można wykorzystać ich możliwości technicznych (szybkości działania, braku zmęczenia) do rozwiązywania typowo ludzkich problemów?

Praktyczne cele *SI* są właśnie związane z próbą zaangażowania komputerów do rozwiązywania zagadnień, co do których nie istnieją proste, klasyczne algorytmy, np.:

- ◆ automatyczne dowodzenie twierdzeń,
- ◆ podejmowanie decyzji na podstawie zbioru kryteriów,
- ◆ inteligentne wyszukiwanie i kojarzenie informacji w dużych zbiorach danych,
- ◆ rozpoznawanie obrazu (a także dźwięków, kojarzenie bodźców dotykowych),
- ◆ budowanie samodzielnych automatów (np. projekt automatycznego kierowcy, który miałby zastąpić człowieka za kierownicą)³,
- ◆ gry (np. szachy, warcaby, reversi, go itd.),
- ◆ planowanie.

Komputery mają zatem wiele zalet, które chcemy wykorzystać, jednak ciągle daleko im do skuteczności ludzkiego mózgu, jego olbrzymiej pojemności (a nie tylko ona się liczy, ale i szybkość dostępu do zawartych w niej informacji), a przede wszystkim do jego możliwości uczenia się, nie wspominając już o możliwości regeneracji⁴!

O grach szczegółowo opowiem dalej, jest to bardzo ciekawy obszar, w którym możliwe są spektakularne wyniki. W dalszych punktach omówię dwa obszary *SI*, które z uwagi na ogrom tematu nie weszły w skład książki, ale Czytelnikowi należy się co najmniej wzmianka o nich. Są to: systemy eksperckie i sieci neuronowe.

² „Computing Machinery and Intelligence” opublikowany w magazynie *Mind*.

³ DARPA, agencja departamentu obrony USA, ogłosiła w 2004 roku konkurs „DARPA Grand Challenge” na wyścigi pojazdów poruszających się bez udziału człowieka. Trzecia odsłona tego konkursu w 2007 roku ujawniła pojazdy, które potrafią poruszać się w mieście, stosując się do zasad ruchu drogowego (w Polsce takie pojazdy nie miałyby zbyt wielkich szans na przebycie wymaganych 60 mil bez wypadku).

⁴ Uszkodzony mózg ludzki nie jest całkowicie bezużyteczny, a nawet potrafi w wielu przypadkach odtworzyć swoje pierwotne funkcje, co w przypadku komputerów na razie jest realizowane wyłącznie przez układy awaryjne. Być może wraz z rozwojem technologii przyjdzie czas na komputery działające na nieco innych zasadach niż obecnie (logika układów dyskretnych), gdzie regeneracja części sytemu na podstawie wiedzy zawartej w jego sprawnej części będzie łatwiejsza.

Systemy eksperckie

System *eksperski* jest to oprogramowanie komputerowe, które umożliwia podejmowanie złożonych decyzji lub odpowiadanie na złożone pytania, potrafiący przy tym uzasadnić udzielane odpowiedzi. System opiera się zwykle na tzw. bazach wiedzy⁵, czyli zbiorach reguł zapisanych w formie implikacji. Bazy wiedzy są tworzone na podstawie wiedzy ludzkich ekspertów, choć zdarza się, iż mogą być generowane automatycznie. Odpowiedzi systemu i ich uzasadnienia są tworzone zgodnie ze znanymi systemowi regułami manipulacji logicznych.

Przykład reguł w systemie eksperckim:

- ♦ if A then B
- ♦ if B then C
- ♦ if C then D
- ♦ if (A i F) then K
- ♦ itp.

Skonstruowane w ten sposób reguły mogą być reprezentowane np. w postaci drzew lub tablic, wewnętrzna reprezentacja bazy wiedzy mocno zależy od języka programowania. Mając bazę wiedzy, musimy dysponować interpreterem, który pozwala ją przeszukiwać, budować fakty lub wywoływać. Moduł taki zwiemy kontrolerem wywołu⁶. Nowatorskie podejście zastosowane w systemach eksperckich polega właśnie na ścisłym oddzieleniu kodu systemu (logika, zasady przetwarzania reguł) od bazy wiedzy, która może być zasilana z zewnątrz w dowolnym momencie od zbudowania samego systemu. System ekspercki jest zatem także techniką konstruowania systemów informatycznych.

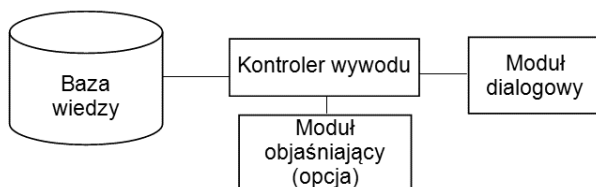
Cechą systemu eksperckiego powinno być jak najlepsze naśladowanie eksperta-człowieka i przyjazna dla odbiorcy interakcja. Twórcom takich systemów przyświeca wiara, iż wiedzę wysoko wykwalifikowanego eksperta można zapisać w programie w takiej postaci, że będzie on mógł przeprowadzać na niej wnioskowanie i generować wyniki zgodne z tymi, jakie podałby ekspert. Ludzki ekspert bywa bardzo drogi, jego czas jest cenny i zbudowanie systemu, który zachowa się tak, jak on, oznacza znaczące obniżenie kosztów konsultacji i łatwiejszy dostęp do wiedzy.

Przykładem systemu eksperckiego może być „sztuczny lekarz”, tzn. program, który naładowany jest wiedzą lekarza specjalisty (np. objawy chorób) i potrafi wydać diagnozę na podstawie zadanych pytań i przedstawionych faktów. Powszechnie za pierwszy system ekspercki uważa się *MYCIN* (Shortliffe, 1972), program analizujący wady krwi i przepisujący odpowiednie antybiotyki. Podobnej klasy są zresztą dowolne systemy diagnostyczne, dla komputera nie ma za bardzo różnicy, czy zachorował człowiek, czy zepsuła się część w samochodzie.

Inne przykłady: system doskonalenia zawodowego, systemy nauczające dzieci upośledzone (program stara się dopasować metodę nauki do trudności przyswajania wiedzy przez dziecko), przewidywanie pogody, prognozowanie rozwoju choroby pacjenta, planowanie działań (np. ruchy robota).

Jakość systemu eksperckiego zależy ściśle od elementów takiego systemu (patrz rysunek 12.1).

Rysunek 12.1.
Schemat systemu
eksperskiego



⁵ Ang. *Knowledge database*.

⁶ Ang. *Inference engine*.

Wyróżnia się wśród nich:

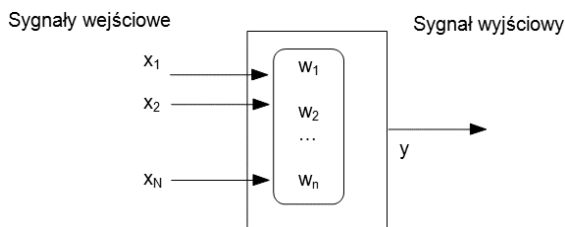
- ♦ *moduł bazy wiedzy* i moduły jej aktualizacji — moduł odpowiedzialny za komputerową reprezentację wiedzy dziedzinowej. Stosuje się zapis stwierdzeń, rachunek zdań, jeśli jest to możliwe — obliczeniowe modele matematyczne. W przypadku systemów branżowych (np. medycyna, prawo, bankowość) pojęcie „ekspert” jest nieco mylące, gdyż nie zawsze wpisywana wiedza pochodzi bezpośrednio od ludzi (nie jest ona efektem ich przepłytywania).
- ♦ *motor* (albo *kontroler*) *wyvodu* — moduł kierujący rozwiązywaniem problemu na podstawie zawartości bazy wiedzy. Jego kluczową cechą powinna być szybkość, a także: umiejętność filtrowania wiedzy, możliwość nawrotów ze ślepych ścieżek wnioskowania, radzenie sobie z konfliktami itp. Jest to najważniejszy element dobrego *SE*!
- ♦ *moduł dialogowy* (interfejs) systemu — moduł odpowiadający za komunikację ze światem zewnętrznym — zadaje pytania i tłumaczy swoje diagnozy. Powinien używać języka zbliżonego do naturalnego.

Systemy eksperckie klasy akademickiej budowano często w językach do tego celu szczególnie się nadających (Lisp, Prolog), jednak w przemyśle informatycznym spotyka się przeróżne hybrydy, wcale nie pomijające znanego nam języka C++. Systemy te dobrze sprawdzają się w zastosowaniach, w których zebrano i spisano olbrzymią wiedzę empiryczną, zwyczajnie trudną do zapamiętania przez pojedynczego człowieka. Ich wadą jest pewna mechaniczność: nie mają one zazwyczaj możliwości uczenia się lub odkrywania wiedzy, modyfikowanie bazy wiedzy nie jest procesem łatwo przewidywalnym w swoich skutkach. Dużym problemem jest budowanie i aktualizacja bazy wiedzy (najlepiej, jeśli istnieją katalogi lub zestawienia, które można bezpośrednio wstawić do bazy wiedzy, oczywiście po pewnej przeróbce, w celu dostosowania do reprezentacji wymaganej przez moduł bazy wiedzy) oraz weryfikacja zawartych w niej informacji, które mogą być zawodne.

Sieci neuronowe

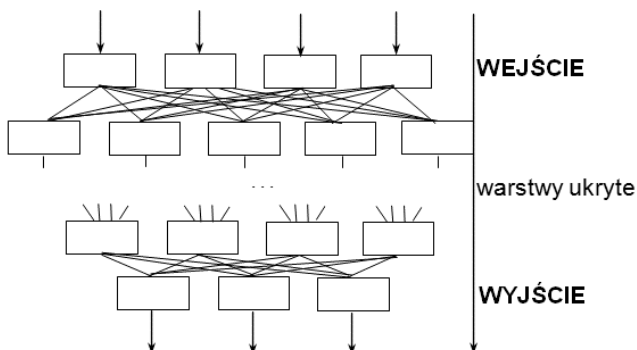
Sieci neuronowe są programowym lub sprzętowym modelem działania ludzkiego mózgu. Bez wnikania w szczegóły biologiczne, mózg ludzki składa się z komórek nerwowych zwanych neuronami, które zbierają dane (sygnały elektrochemiczne z innych komórek i receptorów) za pomocą dendrytów i wyprowadzają je za pomocą aksonów i synaps. Starczy tej biologii? Też tak myślę! Ludziom zamarzyło się odtworzenie fenomenalnych możliwości mózgu poprzez zasyмуляwanie działania neuronu i zbudowanie z ich sieci odpowiednika jeśli nie mózgu, to jego substytutów funkcjonalnych, zdolnych do nauki i rozwiązywania problemów. Rysunek 12.2 przedstawia model informatyczny neuronu jako układu z *wieloma* wejściami i *jednym* wyjściem. Zależność pomiędzy danymi wejściowymi a wyjściowymi może być interpretowana jako rozwiązanie jakiegoś problemu, np. rozpoznanie obrazu lub klasyfikacja danych.

Rysunek 12.2.
Model neuronu



Neuron przetwarza sygnały wejściowe według pewnej funkcji wewnętrznej (może ona być liniowa lub nie), sterowanej wagami w_1, w_2, \dots, w_n . Sieć neuronowa jest połączeniem typu „każdy z każdym” wielu pojedynczych neuronów. Oczywiście z zewnątrz jest to czarna skrzynka z wejściami, na które podajemy sygnały wejściowe i z wyjściami (lub wyjściem). Przykład sieci na rysunku 12.3.

Rysunek 12.3.
Sieć neuronowa



Sieć może mieć w sumie dość dowolną strukturę, posiadać jedną lub wiele warstw oraz ewentualne sprzężenia zwrotne (co najmniej jedno wyjście połączone z wejściem). Do czego może służyć sieć neuronowa? Oto typowe obszary zastosowań:

- ♦ klasyfikacja sygnałów (czyli de facto danych),
- ♦ rozpoznawanie, przewidywanie lub filtrowanie sygnałów,
- ♦ symulowanie działania zmysłów (oko, ucho).

Jak jednak może działać sieć? Wspomnieliśmy, że wynik przetwarzania neuronu sterowany jest wagami w_1, w_2, \dots, w_n . Aby nauczyć neuron określonej reakcji (np. na bodźce optyczne), należy odpowiednio dobrać wagi, a neuron będzie wykonywał wymaganą od niego czynność.

Pojawia się tu jednak problem struktury sieci i jej skomplikowania: neuronów mogą być tysiące i na dodatek są one wzajemnie połączone ze sobą. Jak więc w takich koszmarnych warunkach zajmować się dobieraniem wag?!

Na zadane powyżej pytanie nie odpowiem szczegółowo w ramach tego podpunktu, gdyż o sieciach neuronowych można napisać osobne publikacje (a na dodatek wymagany jest spory ładunek matematyki, czego staram się raczej w tej książce unikać). W ramach naszego siłą rzeczy uproszczonego opisu wspomnę tylko o podstawowej metodzie nauczania sieci:

- ♦ Losujemy wagi podczas fazy inicjacji sieci neuronowej.
- ♦ W kolejnych iteracjach korygujemy wagi na podstawie porównań spodziewanego wyniku działania sieci z wynikiem realnym.
- ♦ Wagi zwiększamy tym bardziej, im większy wystąpił błąd odpowiedzi (odchylenie od spodziewanego wyniku).

W programach do sieci neuronowych rolę nauczyciela pełni oczywiście zbiór algorytmów uczących, wspomagany przez bazę danych (wagi). W procesie nauczania sieć samoczynnie nabywa zdolności generalizacji, czyli oczekiwanego reagowania na dane wejściowe, które nie były zawarte w zbiorze uczącym. Z tego powodu mówi się o uczeniu się sieci neuronowej, choć dla człowieka w pierwszym momencie może to być zaskakujące (jak może uczyć się coś, co nie ma uszu i oczu oraz jest pozbawione wiedzy?).

Reprezentacja problemów

Najważniejszym bodajże zagadnieniem przy pisaniu programów „inteligentnych” jest właściwe *modelowanie rozwiązywanego zagadnienia*. Przykładowo, pisząc program do gry w szachy, musimy sobie zadać następujące pytania:

- ♦ Jaki język najlepiej się nadaje do naszego zadania?
- ♦ Jakich struktur danych należy użyć do reprezentowania szachownicy i pionków?
- ♦ Jakich struktur danych należy użyć do reprezentowania toku myślenia gracza?

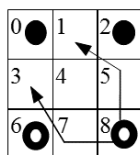
Są to pytania niebagatelne i czasami od odpowiedzi na nie zależy możliwość rozwiązania danego zadania!

Przykład:

Dysponujemy szachownicą 3×3, na której chcemy zamienić miejscami koniki białe z czarnymi (patrz rysunek 12.4). Możliwe ruchy konika znajdującego się na pozycji o numerze 8 są przedstawione za pomocą strzałek.

Rysunek 12.4.

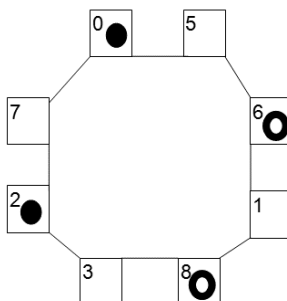
Problem konika szachowego (1)



Reprezentacja zadania w tej postaci, w jakiej jest przedstawiona na rysunku, wcale nie ułatwia nam rozwiązania: nie widać klarownie, jakie ruchy są dozwolone ani celu, jaki należy osiągnąć. Popatrzmy jednak na inne przedstawienie tej samej sytuacji (patrz rysunek 12.5).

Rysunek 12.5.

Problem konika szachowego (2)



Jeśli założymy, że dany konik może poruszać się tylko o dwa pola (w przód i w tył po wyznaczonej ścieżce 0-5-6-1-8-3-2-7-0), to zauważymy, że modelujemy w ten sposób bardzo łatwo ruchy dozwolone i umiemy napisać funkcję, która stworzy listę takich ruchów dla danego konika. Z rysunku została usunięta również pozycja martwa (4), całkowicie niedostępna i w związku z tym w ogóle nam niepotrzebna.

Ćwiczenie 1.

Proszę się zastanowić, jak rozwiązać postawione zadanie, pozwalając być może na *jednoczesne* ruchy kilku pionów?

Ważną reprezentacją zagadnień sztucznej inteligencji są tzw. *grafy stanów*, ilustrujące w węzłach stany problemu (np. planszę z zestawem pionów), a poprzez krawędzie — możliwości zmiany jednego stanu na inny (np. poprzez wykonanie ruchu). W przypadku gry w szachy należałoby zatem zapamiętywać w węźle aktualne stany szachownicy, co czyniłoby reprezentację dość kosztowną, zważywszy na liczbę możliwych sytuacji i — co za tym idzie — na rozmiar grafu!

Gry dwuosobowe i drzewa gier

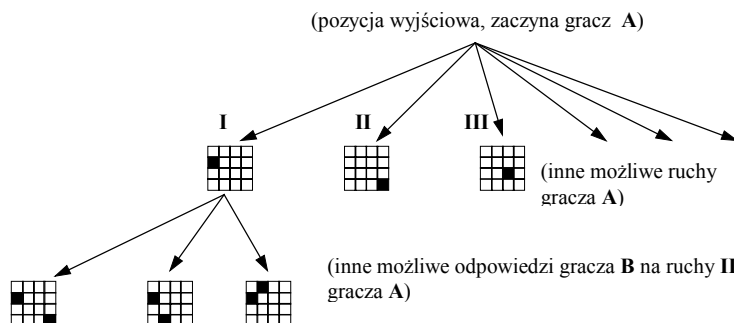
Zaletą typowych gier dwuosobowych jest względna łatwość ich programowej implementacji. Decydują o tym następujące cechy:

- ♦ Na danym etapie mamy komplet wiedzy o sytuacji, w jakiej znajduje się gra (stan planszy).
- ♦ Role graczy są *symetryczne*.
- ♦ Reguły gry są znane z wyprzedzeniem.

W przypadku gier dwuosobowych bardzo wygodną strukturą danych ułatwiających reprezentację stanu i przebiegu gry jest *drzewo*. Ruchy kolejnych graczy są przedstawiane za pomocą węzłów drzewa, w którym poszczególne piętra (poziomy zagłębienia) odpowiadają wszystkim możliwym do wykonania ruchom danego gracza. Przykład drzewa gry jest podany na rysunku 12.6.

Rysunek 12.6.

Przykład
drzewa pewnej
wymyślonej gry



Poszczególne węzły są dość skomplikowanymi strukturami danych, pozwalającymi zapisać kompletny stan pola gry (w naszym przykładzie jest to kratownica 4x4, omawiana gra jest całkowicie fikcyjna). Gracz A, jako zaczynający grę, ma największą swobodę ruchów. Jeśli wybierze on ruch I, to gracz B powinien się dostosować do jego wyboru według dwóch kryteriów:

- ♦ Wybór musi być najkorzystniejszy dla B (kryterium *zdrowego rozsądku*).
- ♦ Wybór musi być zgodny z regułami gry (kryterium *poprawności*).

Drzewo gry jest tym prostsze, im mniej skomplikowana jest gra pod względem możliwości ruchów. Tak więc nietrudno sobie wyobrazić, że drzewo gry „kółko i krzyżyk” jest o wiele prostsze od drzewa gry w warcaby lub szachy.

Drzewa gry w pewnym momencie się kończą (nawet jeśli są bardzo duże): każda sensowna gra prowadzi przecież prędzej czy później do wygranej albo przegranej jednej ze stron lub do remisu! Powstaje zatem praktyczne pytanie: czy da się tak poprowadzić przebieg partii, aby zaproponować jednemu z graczy *strategię wygrywającą*? Aby komputer mógł „rozumować” w kategoriach strategii wygrywającej lub przegrywającej, musi być wyposażony w algorytm skutecznie symulujący w nim zdolność inteligentnego podejmowania decyzji. W praktyce oznacza to budowanie w program dwóch typów funkcji:

- ♦ Ewaluacja: bieżący stan gry jest szacowany pod kątem przewagi jednej ze stron i na tej podstawie jest generowana liczba rzeczywista. Porównanie dwóch stanów gry sprowadzi się zatem do porównania dwóch liczb!
- ♦ Decyzja: na podstawie ewaluacji bieżącego stanu gry i ewentualnie kilku stanów następnych (znanych na podstawie wygenerowanego w całości lub częściowo drzewa gry) podejmowana jest decyzja, który ruch wybrać na danym etapie gry.

Pierwsza funkcja jest ideowo dość prosta do stworzenia, pozwala ona bowiem ocenić siłę rażenia jednej ze stron. O ile jednak sama idea nie jest skomplikowana, to matematyczne uzasadnienie wyboru tej, a nie innej funkcji bywa czasami bardzo trudne. Programy często wykorzystują

pewne intuicyjne obserwacje trudno przekładalne na język matematyki, a jednak w praktyce skuteczne!

Funkcja decyzja próbuje ująć w postaci programu komputerowego coś, co nazywamy po prostu strategią gry. Funkcja decyzja staje się trywialna, jeśli możemy szybko wygenerować *całe* drzewo gry i oszacować drogi, po których może się potoczyć dana partia. Niestety dla większości gier powszechnie uznawanych za godne uwagi rozrywki intelektualne (np. szachy, REVERSI, GO itd.) jest to jeszcze niewykonalne. Komputery są ciągle zbyt wolne do pewnych zastosowań, mimo że zdarza nam się o tym zapominać podczas oglądania oszałamiających animacji czy też fascynujących i intrygujących złożonością gier komputerowych. To co pozostaje, to wygenerowanie fragmentu drzewa gry (do jakiejś sensownej głębokości) i na tej podstawie podjęcie odpowiedniej decyzji.

Okazuje się, że dziedzina sztucznej inteligencji odniosła duże sukcesy w poszukiwaniu takich algorytmów, zwanych często zwyczajnie *strategiami przeszukiwania*. Najbardziej znanymi z nich są: A^* , *mini-max*, algorytm *cięć α - β* czy *SSS**. Szczegółowe przedstawienie każdego z tych algorytmów byłoby dość trudne w tej książce, gdzie została przyjęta zasada prezentacji w C++ gotowych programów, ilustrujących omawiane zagadnienia. Niestety listingi zajęłyby zbyt dużo miejsca, aby to miało w ogóle sens. Zdecydowałem się zatem na szersze omówienie tylko i wyłącznie algorytmu *mini-max* na prostym do kodowania przykładzie gry w „kółko i krzyżyk”. Nawet tak prosta gra wymaga jednak dość sporo linii kodu i — aby nie wypełniać niepotrzebnie stron książki listingami — zdecydowałem się na szersze omówienie kluczowych punktów programu gry w „kółko i krzyżyk”. Czytelnik dysponujący dużą ilością wolnego czasu powinien być w stanie napisać na tej podstawie program dowolnej innej gry dwuosobowej, w naszej prezentacji chodzi wyłącznie o zrozumienie stosowanych mechanizmów. (Pod adresem <ftp://ftp.helion.pl/przyklady/algo4.zip> znajduje się pełna wersja gry, patrz plik *tictac.cpp*).

Pełniejsze omówienie pominiętych (ale bardzo ważnych w praktyce) algorytmów Czytelnik znajdzie np. w [BC89]. Książka ta nie prezentuje co prawda algorytmów w jakimś konkretnym języku programowania, ale dla wprawnego programisty nie powinno to stanowić żadnej przeszkody.

Algorytm mini-max

Wychodzimy z pozycji startowej (stan gry) i szukamy najlepszego możliwego ruchu. Mamy dwa typy węzłów: „max” i „min”. Algorytm przypuszcza, iż przeciwnik skonfrontowany z wieloma wyborami wykonałby najlepszy ruch dla niego (czyli najgorszy dla nas). Naszym celem będzie zatem wykonanie ruchu, który maksymalizuje dla nas wartość pozycji, po której przeciwnik wykonał swój najlepszy ruch (taki, który minimalizuje wartość dla niego). Badamy w ten sposób pewną liczbę poziomów⁷ (nietrudno zauważyć formę drzewa analizy), wartości z ostatnich poziomów są „wnoszone” do góry, wedle reguły *mini-maxa*.

Prosty przykład prezentowany jest na rysunku 12.7 i ilustruje sposób wybrania pierwszego najlepszego ruchu. Wartości liczbowe reprezentują siłę rażenia danej pozycji.

Idea *mini-maxa* polega na systematycznej propagacji wartości danych pozycji, poczynając od samego dołu aż do wierzchołka. Jeśli bieżący wierzchołek ojca reprezentuje ruchy gracza *A*, to z wierzchołkiem tym wiąże się maksimum z wartości jego wierzchołków potomnych.

W przypadku gdy węzeł reprezentuje przeciwnika (gracza *B*), to bierze się minimum tych wartości. Dlaczego tak, a nie na przykład odwrotnie? Jest to związane z istotnym założeniem zdrowego rozsądku obu graczy: *B* będzie się starał maksymalizować swoje szanse zwycięstwa, czyli inaczej

⁷ Ich liczba jest wybieralna i determinuje głębokość zagłębienia procedury *mini-max*. W przypadku niektórych gier zbyt głębokie przeszukiwanie nie ma, oczywiście, większego sensu: są one zbyt „płytkie”!



Rysunek 12.7. Reguła mini-max

mówiąc: zminimalizować szanse *A* na zwycięstwo. Jeśli analiza całego drzewa nie jest praktycznie możliwa, algorytmy poprzestają na pewnej arbitralnej głębokości — w naszym przykładzie jest to $h = 2$.

Założmy również, że wartości liczbowe węzłów z ostatniego poziomu zostały nam dostarczone przez pewną znaną funkcję ewaluacja. W analizowanym przykładzie został wybrany węzeł z wartością $1 = \max(-1, 2, 1)$. Pamiętajmy, że ten wybór zależy od głębokości analizy drzewa gry i przy innej wartości h pierwszy ruch mógłby być zupełnie inny!

Istnieje poprawiona wersja algorytmu *mini-max*, która pozwala znacznie skrócić czas analizy, eliminując zbędne porównania wartości pochodzących z poddrzew i tak niemających szansy na wyniesienie podczas propagacji wartości wg reguły *mini-maxa*. Jest ona powszechnie znana jako *algorytm cięć $\alpha\beta$* . Przykładowo wartość -1 wyniesiona do góry na rysunku 12.7 (szacujemy węzły terminalne od lewej do prawej) sugeruje, iż nie ma sensu analizować tych części drzewa, które wyniosłyby wartość mniejszą niż -1 . Jest to oczywiste wykorzystanie matematycznych własności funkcji *min* i *max*.

Przedstawmy wreszcie tajemniczą procedurę *mini-max*. W celu ułatwienia jej implementacji programowej zostanie ona zaprezentowana w pseudokodzie⁸.

Algorytm przeszukiwania drzewa gry, z wykorzystaniem reguły *mini-max*, ma następującą postać⁹:

```

MiniMax(węzeł w)
{
    jeśli w jest typu MAX to  $v = -\infty$ 
    jeśli w jest typu MIN to  $v = +\infty$ 
    jeśli w jest węzłem terminalnym to
        zwróć ewaluacja(w)
     $p1, p2, \dots pk = \text{generuj}(w)$  //potomkowie węzła w
    dla  $j=1 \dots k$  wykonuj
    {
        jeśli w jest typu MAX to
             $v = \max(v, \text{MiniMax}(pk))$ 
        w przeciwnym przypadku
             $v = \min(v, (\text{MiniMax}(pk)))$ 
    }
    zwróć  $v$ 
}

```

Dotychczas unikaliśmy dyskusji na temat funkcji ewaluacja. Powód jest dość prozaiczny: funkcja ta jest silnie związana z rozpatrywaną grą i nie ma sensu jej omawiać poza jej kontekstem.

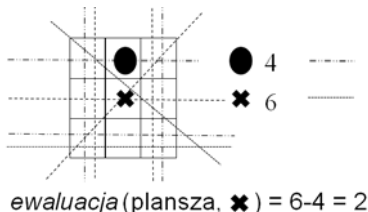
⁸ Warto sobie zdać sprawę, że konkretna implementacja procedury *mini-max* może być zmieniona nie do poznania przez grę i sposób jej reprezentacji.

⁹ Ta wersja jest nastawiona na zwycięstwo gracza MAX.

Po czym poznajemy siłę naszej pozycji w danym etapie gry w „kółko i krzyżyk”? Można wymyślać dość sporo dziwnych kryteriów, mnie jednak przekonało jedno, które notabene dość często pojawia się w literaturze. Wykorzystujemy pojęcie liczby *linii otwartych* dla danego gracza, tzn. takich, które nie są blokowane przez przeciwnika i w związku z tym roją nadzieję na skonstruowanie pełnej linii dającej nam zwycięstwo. Omawianą zasadę ilustruje rysunek 12.8. Wartość tej liczby jest pomniejszana o liczbę linii otwartych dla przeciwnika.

Rysunek 12.8.

Pojęcie linii otwartych
w grze w „kółko i krzyżyk”



Rysunek sugeruje przy okazji strukturę danych, która może być wykorzystywana do zapamiętania stanu gry. Jest to zwykła tablica `int t[9]`, której indeksy odpowiadają pozycjom planszy z rysunku 12.8. Oprócz wartości typu `int` możliwe jest pewne wzbogacenie stosowanej semantyki poprzez użycie typu wyliczeniowego¹⁰:

```
enum KTO{nic, komputer, czlowiek};
```



Uwaga

Z powodu znacznego rozmiaru listingi podane w tym rozdziale są niekompletne, pełny kod znajduje się w pliku `tictac.cpp` dostępnym na [ftp](#).

Wartościami danego pola planszy byłyby wówczas zmienne nie typu `int`, ale typu `KTO`, choć znawcy języków C/C++ wiedzą, że wewnętrznie jest to również `int`.

Funkcja `ewaluacja` otrzymuje w parametrze planszę i informację o tym, dla kogo wyliczenie ma zostać przeprowadzone.

Problem wartości typu plus (minus) nieskończoność można w programie C++ rozwiązać, wybierając liczby, które są znacznie większe od tych zwracanych przez funkcję `ewaluacja`, np.:

```
const plus_niesk = 1000;
const minus_niesk = -1000;
```

Podczas gry następuje zmiana gracza, w związku z tym przydatna będzie funkcja mówiąca nam o tym, kto ma zagrać:

```
KTO Nastepny_Gracz(KTO gracz)
{
    if (gracz==komputer)
        return czlowiek;
    else
        return komputer;
}
```

Przydadzą się również funkcje pomocnicze (niezbyt złożony kod pominąłem):

```
void WyswietlPlansze(plansza);
void ZerujPlansze(plansza);
int KoniecGry(plansza);
```

Funkcja `KoniecGry` dokonuje sprawdzenia, czy ktoś nie postawił linii złożonej z trzech jednakowych znaków, co — jak pamiętamy — gwarantuje zwycięstwo w tej grze, lub czy nie doszło do remisu.

¹⁰ Stałym *komputer* i *czlowiek* odpowiadają na planszy znaki: kółko i krzyżyk.

Sama gra jest zwykłą pętlą, która prowokuje wykonanie ruchów. Załóżmy, że pętla ta została zamknięta w funkcji `Graj`:



tictac.cpp

```
void Graj(plansza, gracz)
{
    gracz_tmp=gracz;
    while(!SprawdZczyKoniecGry(plansza,gracz_tmp))
    {
        WyswietlPlansze(plansza);
        ruch=WybierzRuch(gracz_tmp, plansza);
        WykonajRuch(gracz_tmp, plansza, ruch);
        gracz_tmp=Nastepny_Gracz(gracz_tmp);
    }
}
```

Powyższy schemat jest identyczny dla większości gier dwuosobowych. Na samym początku musimy określić, kto zaczyna (komputer, człowiek?), np. poprzez losowy wybór. Losowanie to powinno się dokonać raz w funkcji `main`, która po wyzerowaniu planszy powinna wywołać procedurę `Graj`. Warunkiem progresji pętli jest stan, w którym nikt jeszcze nie wygrał lub nie zremisował gry. Procedura `WykonajRuch` ściśle zależy od zastosowanych struktur danych. W naszym przypadku może to być po prostu:

```
plansza[ruch]=gracz_tmp;
```

Nieco trudniejsze jest wykonanie ruchu w tak skomplikowanej grze, jak szachy czy też Reversi, mamy bowiem do uwzględnienia efekty uboczne, takie jak bicie pionów, roszady itp.

Skąd mamy jednak wiedzieć, jaki ruch powinien zostać wykonany? Odpowiedzi dostarczyć nam powinna funkcja `WybierzRuch`, która używa poznanej wcześniej funkcji `mini-max`:

```
int WybierzRuch(gracz, plansza)
// wybór ruchu zależy od tego, kto gra
if (gracz==człowiek)
do{
    cout << "Twój wybór(0..8): ";
    cin >> ruch;
}while(!Zajete(plansza, ruch));
else
{
    cout << "Ruch komputera:\n";
    ruch=MiniMax(plansza,gracz);
    return ruch;
}
```

Treść procedury `MiniMax`, którą można odnaleźć w pliku *tictac.cpp*, jest dokładnym tłumaczeniem podanego wcześniej algorytmu, oczywiście z uwzględnieniem struktur danych właściwych dla danej gry.

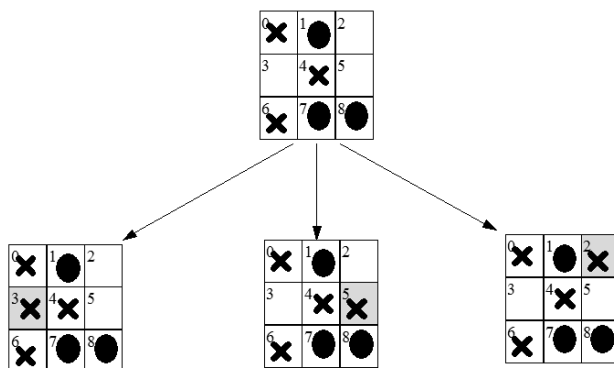
Pozorną trudność może sprawić generowanie węzłów-potomków danego węzła. Rysunek 12.9 ukazuje wynik funkcji `generuj` dla pewnego węzła w (zakładamy, że ruch należał do gracza stawiającego krzyżyki).

Nasuwać się tutaj na myśl jakieś listy, drzewa, zbiory itp. Popatrzmy jednak, jak sprytnie można zakodować listę potomków danego węzła tylko z użyciem jednej pomocniczej planszy (patrz rysunek 12.10). Wystarczy się umówić, że wpisanie wartości innej niż minus jeden oznacza jeden wygenerowany węzeł: pozwala to nam upakować w jednej planszy całą listę możliwych ruchów!

Na tym zakończymy omawianie zagadnień technicznych związanych z programowaniem gier dwuosobowych. Czytelnikowi głębiej zainteresowanemu tą tematyką polecam jednak poszerzenie swojej wiedzy za pomocą literatury specjalistycznej przed przystąpieniem do kodowania np. gry w szachy itp.

Rysunek 12.9.

Generowanie listy
możliwych ruchów
gracza na podstawie
danego węzła

**Rysunek 12.10.**

Kodowanie listy węzłów
potomnych przy użyciu
tylko jednego węzła

0	-1	1	-1	2	1
3	1	4	-1	5	1
6	-1	7	-1	8	-1

Algorytm *mini-max* w swojej podstawowej formie jest dość wolny i w praktyce bywa często zastępowany procedurą *cięć α - β* . Z kolei nie każdy algorytm przeszukiwania dobrze nadaje się do programowania określonych gier z uwagi na skomplikowaną obsługę struktur danych. Dobre algorytmy odszukiwania właściwej strategii gry są, niestety, bardzo złożone. Programiści zaczynają coraz częściej wykorzystywać szybkość współczesnych komputerów, co pozwala uprościć sam proces programowania poprzez stosowanie najprostszych algorytmów przeszukiwania typu *brute-force*. Tak postąpiono w 1996 roku podczas oprogramowywania komputera mającego pokonać w grze w szachy samego mistrza Kasparowa¹¹. We wspomnianym pojedynku górą znowu okazał się człowiek, ale rok później (11 maja 1997) konstruktorzy wystawili jeszcze silniejszy komputer (256 procesorów¹²) i tym razem Kasparow przegrał.

¹¹ Komputer generował w zadanym czasie jak największą liczbę możliwych strategii, obliczał ich siłę (funkcja ewaluacja!) i wybierał tę lokalnie najlepszą.

¹² W chwili obecnej działają superkomputery zbudowane z ponad 200 000 procesorów (projekt Blue Gene/L firmy IBM).

Rozdział 13.

Kodowanie i kompresja danych

Podczas pisania pierwszego wydania tej książki Internet — najpopularniejsza globalna sieć komputerowa — zyskiwał popularność głównie w środowiskach akademickich, gdyż umożliwiał efektywny dostęp do informacji na całym świecie i wzbogacał kontakty naukowe. Podczas pracy nad wydaniem trzecim tej książki można bez wahania powiedzieć, że Internet zszedł już pod strzechy: coraz więcej osób nie wyobraża sobie zakupu komputera bez karty sieciowej, tradycyjny dostęp do Internetu przez wydzwanianie w sieci TPSA został powoli zastąpiony przez numery alternatywne (np. Onet Konekt) lub łącza stałe, instalowane w domach (szerokopasmowy dostęp w technologii ADSL¹ lub poprzez sieci kablowe).

Z dobrodziejstw Internetu korzysta coraz więcej osób, również niezwiązanych z informatyką. Możliwość przechadzania się po sieci za pomocą łatwych w użyciu graficznych przeglądarek (np. Internet Explorer, Opera, Firefox) czy to w poszukiwaniu jakichś istotnych danych, czy też zwyczajnie dla rozrywki, fascynuje wiele osób, stając się nieraz czymś w rodzaju nałogu.

Prostota oprogramowania, które służy do korzystania z zasobów sieciowych, skutecznie odseparowuje zwykłego użytkownika komputera od problemów, z którymi musi sobie radzić oprogramowanie komunikacyjne. Dawniej, gdy główny problem stanowiła niska przepustowość łącz, kluczowym zagadnieniem była *kompresja* przesyłanych danych, czyli takie ich zakodowanie², które — nie zmniejszając ilości przesyłanej informacji — zmniejszy liczbę bitów krążących po kablach. Później punkt ciężkości przesunął się na bezpieczeństwo danych, tzn. ich ochronę przed niepowołanym dostępem z zewnątrz (transmisje w Internecie bardzo łatwo podsłuchać odpowiednimi programami). Obecnie temat kompresji znowu zyskał na popularności, gdyż ilość danych multimedialnych (obraz, dźwięk) transmitowanych przez Internet rośnie wykładniczo — ludzie masowo przesyłają sobie pocztą elektroniczną załączniki z dokumentami pełnymi grafiki, ściągają — często nielegalnie — pliki muzyczne na swoje komputery. Gdyby nie kodowanie danych, to praca w Internecie byłaby praktycznie niemożliwa, nawet przy posiadaniu szybkich łącz.

Czy kompresja dotyczy tylko świata komputerów? Oczywiście nie, inne znane przykłady zastosowań to telefonia komórkowa, cyfrowa telewizja satelitarna (w trakcie pisania tej książki analogowe przekazy satelitarne znikają z dnia na dzień, zastępowane przez tańsze dla nadawców kodowane przekazy cyfrowe) czy choćby transmisje modemowe i faksowe.

¹ Ang. *Asymmetric Digital Subscriber Line*.

² Kodowanie — przedstawienie informacji w postaci dogodnej do przesyłania, np. w postaci ciągów zer i jedynek, czyli po prostu dwóch sygnałów elektrycznych dających się łatwo odróżnić od siebie, np. poprzez pomiar ich amplitudy lub częstotliwości.

Jak w ogóle możliwa jest kompresja danych? Dla laika proces kompresji danych wydaje się magiczny, jednak po powierzchownym nawet wejrzeniu okazuje się, że nie ma w nim niczego tajemniczego. Weźmy dla przykładu 50-znakową wiadomość: „SPOTKANIE JUTRO O PIĘTNASTEJ NA ŁAWCE POD RATUSZEM”. Przyjmując najprostsze kodowanie 8-bitowym kodem ASCII (w którym na każdy z 256 znaków tego kodu przypada pewien 8-bitowy ciąg zerojedynkowy), długość powyższej wiadomości możemy oszacować na $50 \cdot 8 = 400$ bitów³. Czy jednak w przypadku zwykłych tekstów, zawierających komunikaty w języku polskim, musimy koniecznie używać kosztownego kodowania 8-bitowego? Język polski nie zawiera przecież aż $2^8 = 256$ znaków! Załóżmy, że dla typowych tekstów ograniczymy się do następującego alfabetu pokazanego w tabeli 13.1:

Tabela 13.1. Uproszczony alfabet zdefiniowany dla celów kodowania

Znaki	Komentarz	Liczba znaków
'A', ..., 'Z'	Znaki podstawowe	26
' '	Odstęp (spacja)	1
'.'		1
':'		1
'-'		1
'_'	kreska (łącznik)	1
'Ą', 'Ć', 'Ę', 'Ł', 'Ń', 'Ó', 'Ż', 'ż'		8
	Razem:	39

Do zakodowania 39 znaków w zupełności wystarczy 6 bitów ($A = 00\ 0000$, $B = 00\ 0001$, $C = \dots$), czyli komunikat kurczy nam się z 400 do 300 bitów⁴!

Kody tabelkowe, może i prymitywne, mają jednak szerokie zastosowanie w branży komputerowej. Przykładowo terminale znakowe używają dość prostego, aczkolwiek skutecznego kodu ASCII (ang. *American Standard Code for Information Interchange*), w którym pojedyncze znaki są zapisane w jednym bajcie, co pozwala zakodować 256 różnych znaków:

- ♦ 26 dużych i małych liter alfabetu łacińskiego,
- ♦ cyfry od 0 do 9, spację,
- ♦ znaki specjalne, np. %, !, *,
- ♦ znaki sterujące (kody ASCII od 0 do 31), np. polecenie „przejdź do nowego wiersza”, czyli LF — od ang. *Line Feed*), znak końca tekstu itp.,
- ♦ kody ASCII powyżej 127 to tzw. zestaw rozszerzony; zapisuje się w nim znaki narodowe i znaki semigrafiki (symbole pozwalające tworzyć na ekranie ramki itp.).

Urządzenie elektroniczne — tutaj terminal alfanumeryczny — odbiera ciągi bajtów i dzięki wbudowanej elektronice wie, jaki znak rozpoznawalny dla człowieka wyświetlić.

Łatwo zauważyć, że znajomość przesyłanego alfabetu pozwala, przy umiejętnym doborze kodu, znacznie zmniejszyć długość przesyłanego komunikatu, bez utraty informacji w nim zawartej. Istnieje mnogość kodów, bardziej skomplikowanych niż prymitywne kody tabelkowe typu ASCII, nie jest jednakże moim zamiarem zamienienie tego rozdziału w minipodręcznik teorii kodowania i informacji. Bez wnikania w szczegóły, warto być może wspomnieć, że istnieją dwie podstawowe grupy kodów: *równomierne* (o stałej długości słowa kodowego) i *nierównomierne* (o zmiennej długości słowa kodowego). W obu przypadkach można do zakodowanej informacji dołączyć pewne dodatkowe bity kontrolne, ułatwiające odtworzenie informacji, nawet w przypadku częściowego uszkodzenia przesyłanego komunikatu (uzyskujemy wówczas tzw. *kody nadmiarowe*).

³ W celu uproszczenia nie uwzględniamy tutaj żadnych dodatkowych bitów związanych z kontrolą poprawności transmisji danych ani ze szczegółami technicznymi konkretnego protokołu telekomunikacyjnego — inaczej mówiąc, znajdujemy się na poziomie *aplikacji*.

⁴ Zyskujemy 25% pierwotnej długości tekstu!

Nie chciałbym jednak zbyt szeroko omawiać tych zagadnień, gdyż są one bardziej związane z transmisją sygnałów (fizyczna transmisja danych; sens przesyłanej informacji nie jest istotny) niż z informatyką w czystszej postaci (aplikacje użytkownika; sens przesyłanej informacji ma kluczowe znaczenie).

W dalszej części rozdziału omówimy szczegółowo popularny system kodowania z tzw. *kluczem publicznym* oraz kod *Huffmana*, który jest znakomitym i nieskomplikowanym przykładem uniwersalnego algorytmu kompresji danych. Ponieważ wspomnieliśmy o sieci Internet, to zaprezentujemy kodowanie *LZW*, używane w kompresji plików *GIF*. A całość zaczniemy jak zwykle od prostych przykładów!

Czytelników zainteresowanych szczególnie pogłębieniem wiedzy na temat zagadnień kompresji danych zachęcam do lektury [Say02], bardzo kompletnego podręcznika, na dodatek napisanego bardzo prostym językiem. O ogromie dziedziny niech świadczy grubość tej publikacji: ponad 600 stron teorii, a sam autor określa swój podręcznik jako wprowadzenie do tematu.

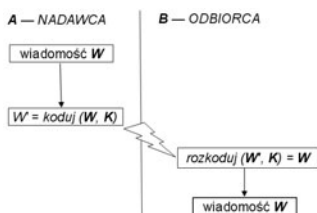
Kodowanie danych i arytmetyka dużych liczb

Kodowanie danych (lub jak kto woli: szyfrowanie wiadomości) ma miejsce wszędzie tam, gdzie z pewnych względów chcemy utajnić zawartość przesyłanej informacji, tak aby jej treść nie dostała się w niepowołane ręce i nie mogła być wykorzystana w niemyślach nam celach. Może ono dotyczyć prywatnej korespondencji, jednak w praktyce najczęstsze zastosowanie znajduje we wszelkiego rodzaju transakcjach gospodarczych ze względu na dobro kontrahentów.

Kodowanie symetryczne

Kodowanie pasjonowało ludzi od wieków i czyniono wielkie starania, aby wymyślać takie algorytmy kodujące, które byłyby trudne do złamania w rozsądnym czasie. Proces kodowania i dekodowania można przedstawić w postaci prostego schematu, zaprezentowanego na rysunku 13.1.

Rysunek 13.1.
Kodowanie symetryczne



Pewna wiadomość W jest szyfrowana przez nadawcę A za pomocą procedury szyfrującej koduj , która przyjmuje dwa parametry: tekst do zaszyfrowania i pewien dodatkowy parametr K , zwany kluczem. Klucz K pełni rolę elementu komplikującego powszechnie znany algorytm kodowania i ma na celu utrudnienie osobom niepowołanym odczytanie wiadomości. Odbiorca B otrzymuje zaszyfrowaną (nieczytelną) wiadomość W' , ale mając do dyspozycji procedurę odkodowującą i dysponując kluczem K , bez trudu poradzi sobie z odtworzeniem wiadomości W .

Przykładem najprostszego kodu jest przypisanie literze alfabetu numeru (załóżmy, że nasz alfabet składa się z 39 znaków). Jest to zwykłe kodowanie tabelkowe, bardzo łatwe zresztą do złamania przez językoznawców uzbrojonych w komputerowe „liczydło” i swoją wiedzę. Jak skomplikować ten powszechnie znany algorytm kodowania? Można na przykład dodać do przesyłanej liczby kodowej pewną wartość K , co spowoduje, że niemożliwe stanie się odczytanie wiadomości poprzez zwykłe porównywanie pozycji tabelki kodującej. Odbiorca B , zanim rozpocznie dekode-

wanie, powinien odjąć od otrzymanych liczb liczbę K , tak aby otrzymać kanoniczny kod tabelkowy⁵. Uważny Czytelnik dostrzeże zasadniczą niedogodność takiego systemu kodującego, przyglądając się rysunkowi 13.1: nadawca i odbiorca muszą znać wartość klucza K . Przesyłanie konwencjonalnymi metodami klucza, np. poprzez kuriera, jest bardzo niepraktyczne i na dodatek naraża na niebezpieczeństwo zarówno poufność danych, jak i... samego kuriera!

Metoda opisana w tym punkcie jest zwana *kodowaniem symetrycznym* — tym samym kluczem szyfrujemy i odszyfrowujemy dane. Nie jest trudno dojść do wniosku, że jest to słaba metoda, do jej złamania wystarczy kradzież klucza lub metody *reverse-engineeringu*. Mimo to algorytmy symetryczne mają dużą zaletę: są szybkie! Niektóre ciekawe algorytmy szyfrujące symetryczne zostały opublikowane i stały się częścią protokołów i znanych programów. Przykładem może tutaj być algorytm *DES* (ang. *Data Encryption Standard*), szyfr blokowy, w którym dane są szyfrowane blokami o długości 64 bitów (tj. 8 znaków ASCII wyposażonych w bit parzystości).

W *DES* klucz ma długość 56 bitów, choć jest zapisywany za pomocą 64 bitów (co ósmy bit jest bitem parzystości). W ciele algorytmu dokonywane są cykliczne permutacje i mieszania bloków danych. Modyfikacja oryginalnego ciągu zależy od wartości podkluczy K_1, K_2, \dots, K_{16} wygenerowanych na podstawie klucza podawanego przez użytkownika (K_0). Podklucze są używane do permutacji i mieszania bloków danych w kolejnych turach przetwarzania algorytmu. Proces odkodowania używa tych samych podkluczy, ale w odwrotnej kolejności.

Algorytm *DES* jest stosowany do kodowania załączników w poczcie elektronicznej, był niegdyś szeroko wykorzystywany w przemyśle, (np. w branży finansowej), obecnie stosuje się w nich raczej jego silniejszą odmianę o nazwie *3DES* z kluczem 168 bitów, trudniejszą do złamania.

W *DES* dla każdej wiadomości klucz wybierany jest losowo spośród 72 000 000 000 000 000 (72 kwadrylionów) możliwych, wiadomości szyfrowane za pomocą algorytmu *DES* uważało się za niemożliwe do złamania. Niestety w 1997 roku firma *RSA* (firma będąca właścicielem opisywanej dalej w tym rozdziale innej metody kryptograficznej) ustanowiła nagrodę w wysokości 10 000 USD za sforsowanie algorytmu *DES* i dzięki wysiłkom powołanej do tego celu grupy i wolnych mocy kilkunastu tysięcy komputerów użytkowników Internetu, zaowocowało to złamaniem kodu w niecałe 3 miesiące. Dziesięć lat później *Frontier Foundation* polepszyła ten wynik, łamiąc zakodowany w *DES* tekst w 9 dni.

Na zakończenie dodam tylko, że istnieją kodery sprzętowe (oparte na układach scalonych) *DES*, które są średnio 1 000 razy szybsze od wersji software'owych.

Kodowanie asymetryczne

Kodowanie asymetryczne eliminuje wadę związaną z kłopotliwą logistyką i pilnowaniem kluczy napotkaną w algorytmach symetrycznych. Rozwiązuje ono efektywnie *problem transmisji klucza* w świecie, gdzie ważne jest, aby wiadomość dotarła w ułamku sekundy do odbiorcy, bez obarczania go dodatkową troską o wiarygodność otrzymanego klucza K .

Zakłada się, że odbiorca może wysłać nadawcy klucz szyfrujący w sposób jawny. Nadawca zaszyfruje nim swoją wiadomość i wysyła ją odbiorcy, który odszyfruje ją drugim kluczem, będącym wyłącznie w jego posiadaniu. Przechwycenie klucza szyfrującego, bez drugiego klucza — odszyfrowującego, nic nie daje!

Pierwszy klucz szyfrujący nazywa się *kluczem publicznym*, a drugi odszyfrowujący — *kluczem prywatnym*. Klucz prywatnego należy strzec jak oka w głowie, a klucz publiczny można swobodnie rozdawać tym, z którymi chcemy się komunikować. Każdy, kto otrzymał klucz publiczny,

⁵ Zarówno przykład kodu, jak i klucza są najprostszy z możliwych i żadna armia na świecie nie zakodowałaby za ich pomocą nawet jadłospisu dziennego, aby nie ośmieszyć się przed przeciwnikiem!

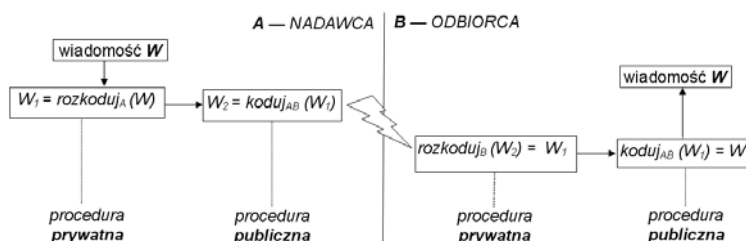
będzie mógł wysłać do nas zaszyfrowaną nim wiadomość, którą odczytamy tylko my jako jedyni posiadacze klucza prywatnego.

Metodę *kodowania z kluczem publicznym*, która eliminowała całkowicie dystrybucję klucza, wynaleźiono w 1976 r. Jej wynalzcami byli W. Diffie i M. Hellman, jednak jej pierwsza praktyczna realizacja została opracowana przez R. Rivesta, A. Shamira i L. Adlemana, stając się znaną jako tzw. kryptosystem *RSA*. Metoda *RSA* gwarantuje bardzo duży stopień bezpieczeństwa przesyłanej informacji. Ponieważ została ona uznana przez matematyków za niemożliwą do złamania, stała się momentalnie obiektem zainteresowania komputerowych maniaków na całym świecie, którzy za punkt honoru przyjęli jej złamanie⁶.

Zanim przeanalizujemy system *RSA* na konkretnym przykładzie liczbowym, spróbujmy zrozumieć samą ideę kryptografii z kluczem publicznym.

System kryptograficzny z kluczem publicznym jest przedstawiony na rysunku 13.2. Składa się on z trzech procedur: *prywatnych*: rozkoduj_A i rozkoduj_B i *publicznej*: koduj_{AB} .

Rysunek 13.2.
System kodujący
z kluczem publicznym



Nadawca A, chcąc wysłać do odbiorcy B wiadomość W , w pierwszym momencie czyni rzecz dość dziwną: zamiast zwyczajnie zakodować ją i wysłać poprzez kanał transmisyjny do odbiorcy, dodatkowo używa funkcji rozkoduj_A na niezaszyfrowanej wiadomości! Czynność ta, na pierwszy rzut oka dość absurdalna, ma swoje uzasadnienie praktyczne: na wiadomości W jest odciskany niepowtarzalny *podpis cyfrowy* nadawcy A, co w wielu systemach (np. bankowych) ma znaczenie wręcz strategiczne! Następnie podpisana wiadomość (W_1) jest szyfrowana przez powszechnie znaną procedurę szyfrującą koduj_{AB} i dopiero w tym momencie wysyłana do B.

Odbiorca B otrzymuje zakodowaną sekwencję kodową i używa swojej prywatnej funkcji rozkoduj_B , która jest tak skonstruowana, że na wyjściu odtworzy podpisaną wiadomość W_1 . Podobnie specjalna musi być funkcja koduj_{AB} , która z cyfrowo podpisanej wiadomości W_1 powinna odtworzyć oryginalny komunikat W .



Ostrzeżenie

Wymogi bezpieczeństwa zakładają praktyczną *niemożność odtworzenia* tajnych procedur rozkodowujących na podstawie jawnych procedur kodujących.

Idea jest zatem urzekająca, wszakże pod warunkiem dysponowania trzema tajemniczymi procedurami, które na dodatek są powiązane ze sobą dość ostrymi wymaganiami! Dopiero po roku od pojawienia się idei systemu z kluczem publicznym powstała pierwsza (i jak do tej pory najlepsza) realizacja praktyczna: system kryptograficzny *RSA*. System ten zakłada, że odbiorca B wybiera losowo trzy bardzo duże liczby pierwsze S , $N1$ i $N2$ (typowo 100 cyfrowe) i udostępnia publicznie tylko ich iloczyn⁷ $N = N1 \cdot N2$ oraz pewną liczbę P , spełniającą warunek:

⁶ Prawa do algorytmu *RSA* posiada *RSA Data Security Inc.*, udzielająca płatnej licencji na jego używanie w programach innych producentów (skorzystały z niej takie produkty, jak Internet Explorer i Netscape Nawigator). *RSA* wchodzi w skład wielu standardów i protokołów sieciowych (np. S/MIME, SSL) oraz programów (*PGP*, warianty bezpiecznej poczty).

⁷ Ponieważ nie są aktualnie znane szybkie metody rozkładu na czynniki pierwsze, dokonanie takiego rozkładu przez osobę postronną jest wysoce nieprawdopodobne.

$$P \cdot S \bmod (N1-1) \cdot (N2-1) = 1.$$

Zostało udowodnione, że dla każdego ciągu kodowego M (tekst zostaje zamieniony na odpowiadający mu ciąg liczbowy o pewnej skończonej długości) spełniona jest równość: $M^{PS} \bmod N = M$.

Kodowanie sprowadzi się zatem do obliczenia równości:

$$\{\text{ciąg kodowy}\} = \text{koduj}(M) = M^P \bmod N,$$

natomiast dekodowanie jest równoważne obliczeniu:

$$M = \text{dekoduj}(\{\text{ciąg kodowy}\}) = \{\text{ciąg kodowy}\}^S \bmod N.$$

Pomimo pozornej trudności wykonania operacji na bardzo dużych liczbach okazuje się, że własności funkcji modulo powodują, iż zarówno ciąg kodowy, jak i jego zaszyfrowana postać należą do tego samego zakresu liczb. Złamanie systemu *RSA* byłoby możliwe, jeśli umielibyśmy na podstawie znanych wartości N i P odtworzyć utajnione S , potrzebne do rozkodowania wiadomości! Nie znaleziono do tej pory algorytmu, który potrafiłby wykonać to zadanie w rozsądnym czasie.

Wszelkie algorytmy kryptograficzne napotykają na problem wykonywania obliczeń na bardzo dużych liczbach całkowitych. Okazuje się, że obliczenia te mogą zostać znacznie uproszczone, ale warunkiem jest traktowanie tych liczb jako współczynników wielomianów. Weźmy dla przykładu liczbę:

$$12\ 9876\ 0002\ 6000\ 0000\ 0054$$

W systemie o podstawie $x = 10$ powyższa liczba może zostać przedstawiona jako:

$$x^{21} + 2x^{20} + (9x^{19} + 8x^{18} + 7x^{17} + 6x^{16}) + (2x^{12}) + (6x^{11}) + (5x^1 + 4).$$

Jeśli $x = 10$ wydaje nam się za małe, to identyczną liczbę otrzymamy, podstawiając np. $x = 10\ 000$:

$$(12x^5) + (9876x^4) + (2x^3) + (6000x^2) + 54.$$

W konsekwencji: jeśli będziemy interpretować duże liczby jako wielomiany, to wszelkie operacje na tych liczbach mogą zostać zastąpione algorytmami działającymi na wielomianach.

Aby dodawać i mnożyć duże liczby całkowite, musimy zatem nauczyć się dodawać i mnożyć... wielomiany!

Reprezentacja wielomianu w C++ jest najprostsza przy użyciu tablic, służących do zapamiętywania współczynników. Wielomian stopnia n i zmiennej x jest ogólnie definiowany następująco:

$$W(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

Obliczanie wartości $W(b)$ dla pewnego b wydaje się dość kosztowne z uwagi na konieczność wielokrotnego mnożenia i dodawania:

```
int oblicz_wielomian(int a, int w[], int rozm)
{ // metoda klasyczna
  int res=0, pot=1;
  for(int j=rozm-1; j>=0; j--)
  {
    res+=pot*w[j]; // sumy cząstkowe
    pot*=a;        // następna potęga a
  }
  return res;
}
```

(W przypadku wielomianów o współczynnikach niecałkowitych należy wszędzie zamienić typ `int` na `double`).

Istnieje jednak tzw. *schemat Hornera*⁸ pozwalający na znacznie prostsze obliczenie $W(b)$:

$$W(b) = (\dots((a_n b + a_{n-1})b + a_{n-2})b + \dots + a_1)b + a_0.$$

Realizacja *schematu Hornera* może być następująca:



horner.cpp

```
const int n=5; // stopień wielomianu

int oblicz_wielomian_H(int a,int w[],int rozm)
{ // schemat Hornera
  int res=w[0];
  for(int j=1; j<rozm; res=res*a+w[j++]);
  return res;
}

int main()
{
  int w[n]={1,4,-2,0,7}; // współczynniki wielomianu
  cout << oblicz_wielomian(2,w,n) << endl; // metoda klasyczna
  cout << oblicz_wielomian_H(2,w,n) << endl; // schemat Hornera
}
```

Przy użyciu reprezentacji tablicowej nieskomplikowane staje się również dodawanie i mnożenie wielomianów:



wielom.cpp

```
const int n=3; // stopień wielomianu

void dodaj_wiel(int x[],int y[],int z[], int rozm)
{
  for(int i=0; i<rozm; i++)
    z[i]=x[i]+y[i]; // wielomian z = x+y
}

void mnoz_wiel(int x[],int y[],int z[], int rozm)
{
  int i,j;
  for(i=0; i<2*rozm-1; i++)
    z[i]=0; // zerowanie rezultatu
  for(i=0; i<rozm; i++)
    for(j=0; j<rozm; j++)
      z[i+j]=z[i+j] + x[i]*y[j];
}
```

Zacytowane powyżej algorytmy są bezpośrednim tłumaczeniem praktycznych sposobów znanych nam ze szkoły podstawowej lub średniej. Co jest ich podstawową wadą? Otóż to co wydawało nam się zaletą: reprezentacja tablicowa (a więc prosty dostęp do współczynników)! Jest ona mało ekonomiczna, jeśli chodzi o zużycie pamięci, o czym najlepiej niech świadczy próba pomnożenia następujących wielomianów:

$$(2x^{1600} + 3x^{900}) \cdot (3x^{85} + 1).$$

⁸ Wynalazcą tej procedury był tak naprawdę Isaac Newton, ale historia przypisała ją Hornerowi.

Owszem, można zarezerwować tablice o rozmiarach $1\ 600$, 85 i $1\ 600+85$ (na wynik), ale biorąc pod uwagę, że będą się one składały głównie z zer, nie jest to najrozsądniejszym pomysłem.

Na pomoc przychodzi tutaj reprezentacja wielomianu za pomocą listy jednokierunkowej; wybieramy najprostsze rozwiązanie, w którym nowe składniki wielomianu są dokładane na początek listy (użytkownik musi jednak pamiętać o wstawianiu nowych składników w określonej kolejności: od potęg najwyższych do najniższych lub odwrotnie). Nie będą zapamiętywane składniki zerowe:



wielom2.cpp

```
typedef struct wsp
{
    int c;
    int j;
    struct wsp *nastepny;
} WSPOLCZYNNIKI, *WSPOLCZYNNIKI_PTR;

WSPOLCZYNNIKI_PTR wstaw(WSPOLCZYNNIKI_PTR p, int c, int j)
{ // dodaje nowy węzeł (współczynnik) do wielomianu
    if(c!=0) // tylko elementy c*(x^j), dla c!=0
    {
        WSPOLCZYNNIKI_PTR q=new WSPOLCZYNNIKI;
        q->c=c;
        q->j=j;
        q->nastepny=p;
        return q;
    }else
    return p; // lista nie została zmieniona
}
```

Funkcje obsługujące taką reprezentację komplikują się nieco, ale algorytmy zyskują znacznie na efektywności i są oszczędne w kwestii zajmowania pamięci. Popatrzmy na funkcję, która doda do siebie dwa wielomiany:

```
WSPOLCZYNNIKI_PTR dodaj(WSPOLCZYNNIKI_PTR x, WSPOLCZYNNIKI_PTR y)
{ //zwraca wielomian x+y
    WSPOLCZYNNIKI_PTR res=NULL;
    while((x!=NULL) && (y!=NULL))
    if(x->j==y->j)
    {
        res=wstaw(res, x->c+y->c, x->j);
        x=x->nastepny;
        y=y->nastepny;
    }
    else
    if(x->j<y->j)
    {
        res=wstaw(res, x->c, x->j);
        x=x->nastepny;
    }
    else
    if(y->j<x->j)
    {
        res=wstaw(res, y->c, y->j);
        y=y->nastepny;
    }
    // W tym momencie x lub y może jeszcze zawierać elementy, które nie zostały obsłużone w pętli while
    // z uwagi na jej warunek. Wstawiamy zatem resztę czynników (jeśli istnieją):
    while (x!=NULL)
    {
        res=wstaw(res, x->c, x->j);
        x=x->nastepny;
    }
```



```

    }
    while (y!=NULL)
    {
        res=wstaw(res,y->c,y->j);
        y=y->nastepny;
    }
    return res;
}

```

Algorytm funkcji `dodaj` został pozostawiony w możliwie najprostszej i łatwej do analizy postaci (Czytelnik dysponujący wolnym czasem może się pokusić o wprowadzenie w nim szeregu drobnych ulepszeń).

Popatrzmy jeszcze na sposób korzystania z powyższych funkcji⁹:

```

int main()
{
    WSPOLCZYNNIKI_PTR pw1, pw2, pw3, pwtemp;
    pw1=pw2=pw3=pwtemp=NULL;
    pw1=wstaw(pw1,5,1700);
    pw1=wstaw(pw1,6,700);
    pw1=wstaw(pw1,10,50);
    pw1=wstaw(pw1,5,0);

    pw2=wstaw(pw2,6,1800);
    pw2=wstaw(pw2,-6,700);
    pw2=wstaw(pw2,5,50);
    pw2=wstaw(pw2,15,0);

    pw3=dodaj(pw1,pw2);
}

```

Kod implementuje oczywiście następujące wielomiany:

- ♦ $pw1(x) = 5x^{1700} + 6x^{700} + 10x^{50} + 5$,
- ♦ $pw2(x) = 6x^{1800} - 6x^{700} + 5x^{50} + 15$,
- ♦ $pw3(x) = pw1(x) + pw2(x) = 6x^{1800} + 5x^{1700} + 15x^{50} + 20$.

Omawiając system kodowania danych *RSA*, napotkaliśmy na niedogodność związaną z operacjami na bardzo dużych liczbach całkowitych. Aby otrzymać ciąg kodowy, powstały na podstawie pewnego tekstu M^{10} , musimy obliczyć wyrażenie:

$$\{\text{ciąg kodowy}\} = M^P \bmod N.$$

Podnoszenie do potęgi może być zrealizowane poprzez zwykłe mnożenie, ale co zrobić z obliczaniem funkcji modulo? Jak na przykład poradzić sobie z wyliczeniem:

$$12\,9876\,0002\,6000\,0000\,0054 \bmod N?$$

Jeśli wszakże przedstawimy powyższą liczbę jako wielomian o podstawie $x = 10\,000$, to otrzymamy znacznie prostsze wyrażenie:

$$12(x^5 \bmod N) + 9876(x^4 \bmod N) + 2(x^3 \bmod N) + 6000(x^2 \bmod N) + 54.$$

Wartości w nawiasach są stałymi, które można wyliczyć tylko raz i „na sztywno” wpisać do programu kodującego, eliminując nieco pierwotną wadę *RSA*, jaką jest jego wolne działanie.

Ta wada algorytmu *RSA* doprowadziła do powstania ciekawej wariacji metody, użytej po raz pierwszy bodajże w PGP, polegającej na szyfrowaniu danych szybkim szyfrem symetrycznym (np. *3DES*) przy użyciu całkowicie losowego klucza; sam klucz szyfrujemy wolniejszym szyfrem sy-

⁹ Wersja na *ftp* zawiera funkcję do wyświetlania zawartości, sprawdź ją w praktyce!

¹⁰ Pamiętajmy, że po zamianie każdej litery tego tekstu na pewną liczbę (np. w kodzie ASCII) całość możemy traktować jako jedną, bardzo dużą liczbę M .

metrycznym i dołączamy do wiadomości. Odbiorca najpierw rozszyfrowuje klucz, a potem używa go do rozszyfrowania właściwych danych. W ten sposób osiągamy pożądaný cel — to znaczy szybkie szyfrowanie dużych danych i bezpieczne przekazanie klucza do ich odtworzenia.

Metody prymitywne

Informacje podane w tym punkcie Czytelnik powinien traktować raczej jako ciekawostkę niż metodę bezpiecznego szyfrowania, choć można je wykorzystać w prostych programach co najmniej w celu utrudniania życia przeciwnikowi (włamywaczowi chcącemu pozyskać nasze cenne dane).

Metoda pierwsza wykorzystuje właściwości funkcji XOR (różnica symetryczna, patrz dodatek B) do odwracania bitów. Pierwsze użycie XOR na pewnej wartości binarnej i znaku umownie określonego kluczem (lub znaku, który jest przeciwieństwem ciągu bitów) wykonuje operacje szyfrowania, drugie — odkodowania.

Proszę spojrzeć na przykładowy program w C++:



xor.cpp

```
void Xor(char *s, char xor_key)
{
    for (int i=0; s[i] != '\0'; i++)
        s[i]=s[i]^xor_key;
}

int main()
{
    char s[]="ała ma kota";
    cout << "Oryginalny ciąg znaków:\t" << s << endl;
    Xor(s,12);
    cout << "Ciąg zakodowany:\t" << s << endl;
    Xor(s,12);
    cout << "Ciąg odkodowany:\t" << s << endl;
}
```

Wynik działania:

```
Oryginalny ciąg znaków: ała ma kota
Ciąg zakodowany:      m`m.am.gcxm
Ciąg odkodowany:      ała ma kota
```

Podobna w swojej prostocie metoda wykorzystuje własność kodu ASCII, w którym jest znana maksymalna wartość kodu przyjmowanego przez znak:



255.cpp

```
void odejmij(char *s)
{
    for (int i=0; s[i] != '\0'; i++)
        s[i]=255-s[i];
}
```

Funkcja main jest niemal identyczna (zamiast xor wywołujemy odkoduj(s) bez parametru):

```
int main()
{
    char s[]="ała ma kota";
    cout << "Oryginalny ciąg znaków:\t" << s << endl;
```

```
odejmij(s):
cout << "Ciąg zakodowany:\t"      << s << endl;
odejmij(s):
cout << "Ciąg odkodowany:\t"      << s << endl;
}
```

Wynik działania:

```
Oryginalny ciąg znaków:  ala ma kota
Ciąg zakodowany:       x&xd1xd0&xf6x
Ciąg odkodowany:       ala ma kota
```

Łamanie szyfrów

W poprzednich punktach omówiliśmy dwie podstawowe klasy algorytmów kodujących, dla odprężenia podam kilka podstawowych informacji na temat... ich łamania. Ponieważ wyczerpujące omówienie tematu wymagałoby napisania odrębnej książki, postanowiłem tylko wymienić pobieżnie kilka najpopularniejszych metod stosowanych do łamania szyfrów:

- ♦ *Kradzież klucza* — niekoniecznie przy użyciu tzw. czynnika ludzkiego, ale np. przechwycenie transmisji klucza pomiędzy odbiorcą i nadawcą informacji. Przy algorytmach symetrycznych może przybrać perfidną formę podmiany kluczy, podczas gdy obie strony wymieniają klucze dla późniejszej bezpiecznej komunikacji. Atakujący ustawia się pomiędzy nadawcą a odbiorcą na linii komunikacyjnej i przeprowadza oddzielną wymianę kluczy z każdą ze stron!
- ♦ *Metoda „na siłę”* (ang. *brute-force*) — atakujący próbuje wszystkich możliwych kombinacji kluczy po kolei. W metodzie tej wymagana moc obliczeniowa wzrasta wykładniczo wraz z długością klucza, np. klucz 40-bitowy wymaga 2^{32} kroków, co jest wykonywalne na komputerach klasy domowej niemal w czasie rzeczywistym. Uznaje się obecnie, że klucze 128-bitowe są nie do złamania w ciągu najbliższych lat, ale zakłada to wyłącznie podejście *brute-force*, a co się stanie, jeśli urodzi się jakiś geniusz matematyczny, który przyspieszy tę metodę? Nie ma co jednak popadać w pesymizm, gdyż wiele systemów jest obecnie tak projektowanych, aby klucze miały charakter dynamiczny i dość szybko się dezaktualizowały. Mimo to postęp technologii może faktycznie zakwestionować obecnie stosowane metody, gdyż w tzw. komputerach kwantowych (w trakcie pisania tej książki istniejących wyłącznie na papierze) przewiduje się, że wzrost mocy obliczeniowej będzie rewolucyjny (o rzędy wielkości w stosunku do obecnych osiągnięć) i wówczas metoda *brute-force* może stać się użyteczna nawet przy bardzo długich kluczach!
- ♦ *Dedukcja na podstawie fragmentu lub całości treści* — atakujący może założyć, że wymieniana korespondencja zawiera znane elementy, np. adresy i nazwy firm w nagłówkach, i pod tym kątem może próbować przyspieszyć rozpracowanie algorytmu lub złamać jego strukturę.



Czytelnikom zainteresowanym algorytmami kryptograficznymi i eksperymentowaniem na nich polecam przejrzanie biblioteki *CryptPak406.zip* autorstwa Markusa Hahna, załączonej w archiwum ZIP dostępnym na [ftp](#). Jest ona dystrybuowana na zasadach open source, przygotowanej do kompilacji w środowisku Win32 i Linux.

Techniki kompresji danych

Jednym z historycznie pierwszych przykładów kompresji danych był alfabet *Morse'a*. Samuel Morse zauważył, że pewne litery alfabetu występują częściej niż inne, i wykorzystał to w swoim systemie kodowania, opartym na dwóch znakach: kreska (–) i kropka (.), łatwych do przesyłania za pomocą telegrafu. Zmniejszenie długości przesyłanych tekstów uzyskane zostało poprzez za-

kodowanie częściej występujących znaków krótszym ciągiem kodowym, a znaków występujących rzadziej — dłuższym¹¹ (tabela 13.2).

Tabela 13.2. *Alfabet Morse’a*

Znak	Kod
A	· —
B	— · · ·
C	— · — ·
D	— · · ·
E	·
F	· — · — ·
G	— — ·
H	· · · ·
I	· ·
J	· — — —
K	— · —
L	· — · ·
M	— —
N	— ·
O	— — —
P	· — — ·
R	· — · ·
S	· · ·
T	—
U	· · —
W	· — —
Y	— · — —
Z	— — · ·

Koncepcyjnie podobny system kodowania jest zastosowany w alfabecie Braille’a, gdzie znak (wyraz) reprezentowany jest przez dwuwymiarową liczkę o rozmiarze 2×3 , zawierającą punkty wypukłe lub płaskie. Ponieważ poszczególnych wartości jest $2^6 = 64$, co przekracza liczbę znaków alfabetu (26), pozostałe kody można użyć do reprezentowania słów często występujących — w tym celu jeden znak jest przeznaczony do poinformowania odbiorcy, że za nim wystąpi całe słowo, a nie znak¹².

W ogólnym modelu matematycznym kompresji danych mamy — podobnie, jak dla kodowania i dekodowania — do czynienia z dwoma algorytmami: kompresującym i rekonstruuującym. Co się tyczy samych algorytmów, to w zasadzie dwie najpopularniejsze metody klasyfikacji dzielą algorytmy kompresji na:

- ◆ *bezzratne* (po zakodowaniu i zrekonstruowaniu uzyskujemy wynik zgodny w 100% z oryginałem),
- ◆ *stratne* (po za kodowaniu i zrekonstruowaniu wynik może się nieco różnić od oryginału).

W kompresji stratnej dopuszczamy pewną utratę informacji, oczywiście w zastosowaniach, które to umożliwiają, np. kompresja mowy lub obrazu. Na pewno kompresji stratnej nie można użyć w systemach, w których liczy się 100-procentowa wiarygodność, np. bankowych lub innych systemach tej klasy odpowiedzialności biznesowej.

¹¹ Pretendentów przynających się do wynalezienia tego systemu kodowania było wielu, ale do historii przeszedł Samuel Morse, który pierwszą depeszę napisaną tym alfabetem przesłał z Waszyngtonu do Baltimore w 1844 r.

¹² Alfabet został opublikowany w 1837 przez Louisa Braille’a, francuskiego pedagoga, niewidomego od dziecka, który — pracując jako nauczyciel w zakładzie dla niewidomych — chciał umożliwić niewidomym dzieciom czytanie.

Algorytmów kompresji jest bardzo dużo i na dodatek dziedzina ta jest w trakcie ciągłego rozwoju, wymuszonego potrzebami współczesnej technologii. Bez wnikania w poboczne klasyfikacje można powiedzieć, że głównymi kryteriami oceny algorytmów kompresji są:

- ♦ szybkość działania,
- ♦ stopień kompresji, tj. współczynnik, w jakim uległ zmniejszeniu rozmiar danych wejściowych po przetworzeniu ich przez algorytm kompresji.

Wszelkie algorytmy kompresji wykorzystują fakt, iż w plikach z danymi znajduje się więcej danych, niż jest to faktycznie potrzebne do przekazania tej samej zawartości informacyjnej. Taką właściwość zbioru danych nazywamy fachowo redundancją — pełną definicję matematyczną darujemy sobie, gdyż nie chciałbym zamieniać tej książki w zbiór teorii informacji. Zresztą samo pojęcie nie jest egzotyczne, np. w językoznawstwie jest to występowanie w języku elementów funkcjonalnie zbędnych.

Kompresja poprzez modelowanie matematyczne

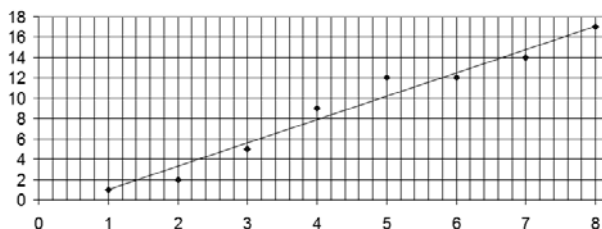
Kompresja przez modelowanie matematyczne polega na próbie odtworzenia redundancji zbioru danych. Metoda ta daje się dość łatwo wytłumaczyć na prostym przykładzie, który niestety ma to do siebie, że nie jest w praktyce zbyt użyteczny. Niemniej ma on swoje walory edukacyjne i pozwala opisać typowy schemat działania algorytmów tego rodzaju.

Załóżmy, że podczas transmisji danych przekazaliśmy następujący ciąg liczb: 1, 2, 5, 9, 12, 12, 14 i 17. Jest to fragment przekazu i naszym zadaniem jest sprawdzenie, czy w celu przekazania tego ciągu trzeba koniecznie przekazać aż $8 \cdot 5 = 40$ bitów (każda z liczb od 1 do 17 może zostać zakodowana binarnie na 5 bitach) — być może ta sama informacja mogłaby zostać przekazana nieco zwięźlej?

Jednym ze sposobów jest wydedukowanie funkcji matematycznej, która na podstawie numeru sekwencyjnego próbki pozwoli nam wyliczyć wartość przekazywanego znaku. Aby odtworzyć taką funkcję, spróbujemy narysować prosty wykres danych (rysunek 13.3).

Rysunek 13.3.

Kompresja wykorzystująca matematyczne modelowanie zbioru danych



Już na pierwszy rzut oka widać, że dane układają się z dobrym przybliżeniem w kształt prostej (jest nałożona na wykres). Załóżmy, że zamodelujemy ten ciąg danych funkcją $(7n-4)/3$, gdzie n jest numerem próbki. Porównajmy teraz ciąg kodowy z naszym modelem matematycznym:

n	1	2	3	4	5	6	7	8
Wartość	1	2	5	9	12	12	14	17
$(7n-4)/3$	1	3	6	8	10	13	15	17
Odchyłka	0	1	1	-1	-2	1	1	0

Funkcja modelująca może zostać zakodowana na stałe (albo też jej parametry mogą zostać przekazane na początku przekazu) i w efekcie — znając ją — aby przesłać identyczną informację, trzeba przekazać wyłącznie informację o odchyleniach: 0, 1, 1, -1, -2, 1, 1, 0. W ciągu odchyłeń pojawiają się tylko cztery wartości odchyłeń:

Odchylenie	Kod
-2	00
-1	01
0	10
1	11

Można więc zakodować je za pomocą 2 bitów! Abstrahując w tym momencie od funkcji modelującej, sama informacja użyteczna może w tym momencie zająć nie 40 bitów, ale $8 \cdot 2 = 16$.

Kompresja metodą RLE

Kompresja metodą *RLE* (ang. *Run Length Encoding*) jest prostym, bezstratnym algorytmem znakomicie sprawdzającym się przy kompresji bloków danych, w których powtarza się wiele takich samych znaków pod rząd. Najprostszym przykładem mogą być pliki graficzne zawierające obrazy o jednolitych powierzchniach, pliki tekstowe natomiast kompresują się już znacznie gorzej.

Załóżmy, że w pliku graficznym kodujemy dane o kolorach (i ewentualne inne informacje) w pojedynczej wartości 8-bitowej. W naszych przykładach taką wartość będziemy symbolizować literkami A, B, C itd. W ramach przykładu popatrzymy na fragment pliku graficznego:

AABBBBCCDDAABBBDDDDDDDDAAAAA (30 znaków)

Blok danych ma 30 bajtów, ale informacja w nim przekazana może równie dobrze zostać zakodowana jako:

AA*4BCCDDA*3B*9D*6A (17 znaków),

gdzie sekwencja $\{ * N Z \}$ oznacza N-krotne wystąpienie znaku Z. Znacznik * jest separatorem pozwalającym odróżnić dane niekodowane od danych, które zdecydowaliśmy się zastąpić sekwencją kodową.

Zastanówmy się zatem, jakie powinno być kryterium wyboru, czy dane kodować czy nie?

Wbrew pozorom nie jest to specjalnie skomplikowane, jeśli uświadomimy sobie, czemu służy w ogóle kompresja? W nawale prezentowanych algorytmów mogło nam to gdzieś umknąć, zatem pozwól, Czytelniku, że przypomnę: naszym celem jest zmniejszenie rozmiaru bloku danych bez utraty jego zawartości informacyjnej.

Czy warto zakodować jeden znak Z? Oczywiście że nie, zamiast efektu kompresji doprowadzamy nawet do wzrostu objętości pliku danych:

$Z \rightarrow *1Z$ (3 znaki w miejsce jednego)

Podobnie ma się sprawa z dwoma znakami, a nawet trzema, gdzie nie ma ani kompresji, ani wzrostu objętości pliku danych:

$ZZZ \rightarrow *3Z$ (3 znaki w miejsce... trzech).

Kompresja LRE staje się zatem efektywna, gdy przepuszczamy bez zmian sekwencje od 1 do 3 znaków, a koncentrujemy się na kompresji powtarzalnych bloków o rozmiarze od 4 wzwyż.



W archiwum ZIP dostępnym na *ftp* w pliku *rle8_sc.zip* znajdują się przykłady procedur kodujących i dekodujących RLE autorstwa Shaun Case. Są to programy na licencji *public domain*, ściągnięte przez *ftp* z Internetu. Pliki są dostarczone w nietkniętej postaci i mogą wymagać dostosowania do konkretnej wersji kompilatora C++.

Oryginalnie są napisane w języku C dla kompilatora Borland C++ 2.0; program daje się bez problemu kompilować w Borland C++ Builder 5.0; aby skompilować w Visual C++, przeczytaj plik *rle.txt*.

Kompresja danych metodą Huffmana

Pierwszym algorytmem, który opiszę szczegółowo, jest algorytm *Huffmana*, który jest wręcz akademickim przykładem, jak można stworzyć dobry algorytm dający się łatwo implementować za pomocą współczesnych języków programowania. Sam algorytm należy do licznej klasy tzw. *algorytmów prefiksowych*, jego opis poprzedzę jednak wstępem, który ma na celu wytłumaczenie głównej idei, na której ów algorytm bazuje.

Kod, którego zdecydujemy się używać, może się znacznie różnić od znanego kodu ASCII. Jak pamiętamy, kod ASCII jest tabelą 8-bitowych znaków tekstu (nie wszystkie są co prawda używane w języku polskim, ale nie ma to tutaj większego znaczenia). Jego podstawową cechą jest równa długość każdego słowa kodowego odpowiadającego danemu znakowi: 8 bitów. Czy jest to obowiązkowe? Otóż nie, popatrzmy na przykład kodowania znaków pewnego alfabetu 5-znakowego (tabela 13.3).

Tabela 13.3. Przykład kodowania znaków pewnego alfabetu 5-znakowego

Znak	Kod bitowy
☞	000
☝	001
☞	01
☞	10
☞	11

Gdzieś w dalekiej dżungli żyje lud, który potrafi za pomocą kombinacji tych 5 znaków wyrazić wszystko: wypowiedzenie wojny, rozejm, prośbę o żywność, prognozę pogody, etc. Teksty zapisywane są na liściach pewnej odpornej na działanie pogody rośliny. W celu szybkiej komunikacji został wymyślony system szybkiego przesyłania wiadomości za pomocą sygnałów trąb niosących dźwięk na bardzo długie dystanse.

Dwa krótkie sygnały oznaczają znak ☞, krótki i długi oznaczają ☝ itd., zgodnie z przedstawioną poniżej tabelką (0 — sygnał długi, 1 — krótki). Jest godne docenienia, iż mamy przed sobą niewątpliwie kod... binarny! (Nawet jeśli ów tajemniczy lud nie zdaje sobie z tego sprawy).

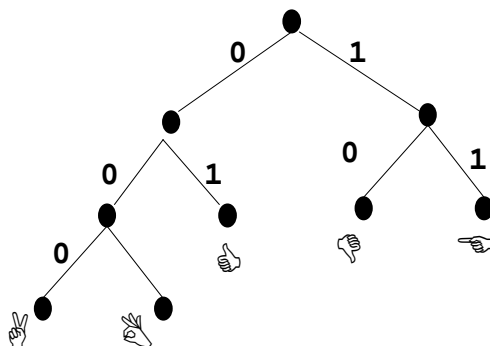
Załóżmy, że pewnego dnia odebrano następujący sygnał: 011110000001 (nadawca wysłał wiadomość: ☞☞☞☞☞, czyli „doślijcie świeże melony”). Czy możliwe jest nieprawidłowe odtworzenie wiadomości, tzn. ewentualne pomylenie jednego znaku z innym? Spróbujmy:

- 0 — znakiem może być: ☝ lub ☞ lub ☞.
- 01 — już wiemy, że jest to ☞!
- ☞+1 — znakiem może być: ☝ lub ☞.
- ☞+11 — już wiemy, że jest to ☞!
- ☞+☞+1 — znakiem może być: ☝ lub ☞
- ...
- (itd.)

Pomyłki są, jak to wyraźnie widać, niemożliwe, gdyż żaden znak kodowy nie jest *przedrostkiem* (prefiksem) innego znaku kodowego. Dotarliśmy do istotnej cechy kodu: ma on być *jednoznaczny*, tzn. nie może być wątpliwości, czy dana sekwencja należy do znaku ☞, czy też może do znaku ☝.

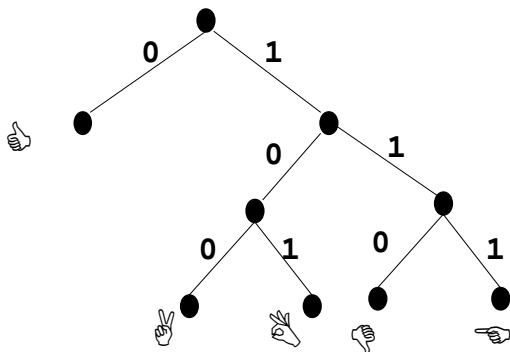
Konstrukcja kodu o powyższej własności jest dość łatwa w przypadku reprezentacji alfabetu w postaci tzw. drzewa kodowego. Dla naszego przykładu wygląda ono tak, jak na rysunku 13.4.

Rysunek 13.4.
Przykład drzewa
kodowego (1)



Przechadzając się po drzewie (poczynając od jego korzenia aż do liści), odwiedzamy gałęzie oznaczone etykietami 0 (lewe) lub 1 (prawe). Po naszym dotarciu do danego liścia ścieżka, po której szliśmy, jest jego binarnym słowem kodowym. Zasadniczym problemem drzew kodowych jest ich... nadmiar. Dla danego alfabetu można skonstruować cały las drzew kodowych, o czym świadczy przykład rysunku 13.5.

Rysunek 13.5.
Przykład drzewa
kodowego (2)



Powstaje więc pytanie: które drzewo jest najlepsze? Oczywiście kryterium jakości drzewa kodowego jest związane z naszym celem głównym: kompresją. Kod, który zapewni nam największy stopień kompresji, będzie uznany za najlepszy. Zwróćmy uwagę, że długość słowa kodowego nie jest stała (w naszym przykładzie wynosiła 2 lub 3 znaki). Jeśli w jakiś magiczny sposób sprawimy, że znaki występujące w kodowanym tekście najczęściej będą miały najkrótsze słowa kodowe, a znaki występujące sporadycznie — odpowiednio — najdłuższe, to uzyskana reprezentacja bitowa będzie miała najmniejszą długość w porównaniu z innymi kodami binarnymi.

Na tym spostrzeżeniu bazuje kod *Huffmana*, który służy do uzyskania optymalnego drzewa kodowego. Jak nietrudno się domyślić, potrzebuje on danych na temat częstotliwości występowania znaków w tekście. Mogą to być wyniki uzyskane od językoznawców, którzy policzyli prawdopodobieństwo występowania określonych znaków w danym języku, lub po prostu nasze własne wyliczenia bazujące na wstępnej analizie tekstu, który ma zostać zakodowany. Sposób postępowania zależy od tego, co zamierzamy kodować (i ewentualnie przesyłać): teksty języka mówionego, dla którego prawdopodobieństwa występowania liter są znane, czy też losowe w swojej treści pliki „binarne” (np. obrazy, programy komputerowe itp.).

Dla zainteresowanych podaję tabelkę zawierającą dane na temat języka polskiego (przytaczam za encyklopedią Wikipedi¹³ — patrz tabela 13.4).

¹³ http://pl.wikipedia.org/wiki/Częstość_występowania_liter_w_języku_polskim

Tabela 13.4. *Prawdopodobieństwa występowania liter w języku polskim*

Litera	Prawdop.	Litera	Prawdop.	Litera	Prawdop.	Litera	Prawdop.
Spacja	0,140	f	0,003	n	0,056	u	0,020
a	0,086	g	0,014	ń	0,001	w	0,042
ą	0,011	h	0,009	o	0,073	y	0,038
b	0,014	i	0,086	ó	0,007	z	0,056
c	0,038	j	0,028	p	0,027	ż	0,001
ć	0,005	k	0,029	r	0,037	ź	0,012
d	0,033	l	0,021	s	0,041		
e	0,087	ł	0,013	ś	0,008		
ę	0,014	m	0,033	t	0,043		

Algorytm *Huffmana* korzysta w bezpośredni sposób z takich właśnie tabel. Wyszukuje on i grupuje rzadziej występujące znaki, tak aby w konsekwencji przypisać im najdłuższe słowa binarne, natomiast znakom występującym częściej — odpowiednio najkrótsze. Może on operować prawdopodobieństwami lub częstotliwościami występowania znaków. Poniżej podam klasyczny algorytm konstrukcji kodu *Huffmana*, który następnie przeanalizujemy na konkretnym przykładzie obliczeniowym.

FAZA REDUKCJI (kierunek: w dół)

1. Uporządkować znaki kodowanego alfabetu wg malejącego prawdopodobieństwa.
2. Zredukować alfabet poprzez połączenie dwóch najmniej prawdopodobnych znaków w jeden znak zastępczy, o prawdopodobieństwie równym sumie prawdopodobieństw łączonych znaków.
3. Jeśli zredukowany alfabet zawiera 2 znaki (zastępcze), skok do punktu 4, w przeciwnym przypadku — powrót do 2.

FAZA KONSTRUKCJI KODU (kierunek: w górę)

4. Przyporządkować dwóm znakom zredukowanego alfabetu słów kodowych 0 i 1.
5. Dopisać na najmłodszych pozycjach słów kodowych odpowiadających dwóm najmniej prawdopodobnym znakom zredukowanego alfabetu 0 i 1.
6. Jeśli powróciliśmy do alfabetu pierwotnego, koniec algorytmu, w przeciwnym wypadku — skok do 5.

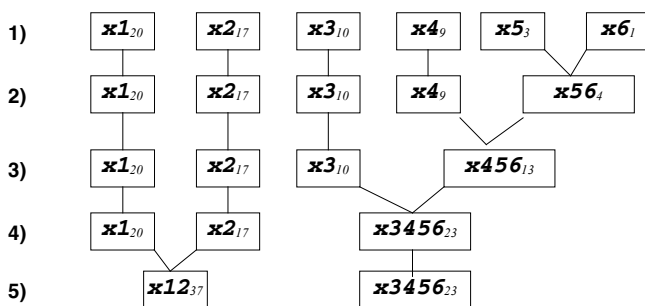
Zdaję sobie sprawę, że algorytm może wydawać się niezbyt zrozumiały, ale wszystkie jego ciemne strony powinien rozjaśnić konkretny przykład obliczeniowy, który zaraz wspólnie przeanalizujemy.

Załóżmy, że dysponujemy alfabetem składającym się z sześciu znaków: x_1, x_2, x_3, x_4, x_5 i x_6 . Otrzymujemy do zakodowania tekst o długości 60 znaków, których częstotliwości występowania są następujące: 20, 17, 10, 9, 3 i 1. Aby zakodować sześć różnych znaków, potrzeba minimum 3 bitów ($6 < 2^3$), zatem zakodowany tekst zająłby $3 \cdot 60 = 180$ bitów. Popatrzmy teraz, jaki będzie efekt zastosowania algorytmu *Huffmana* w celu otrzymania optymalnego kodu binarnego.

Postępując według reguł zacytowanych w powyższym algorytmie, otrzymamy następujące redukcje (patrz rysunek 13.6).

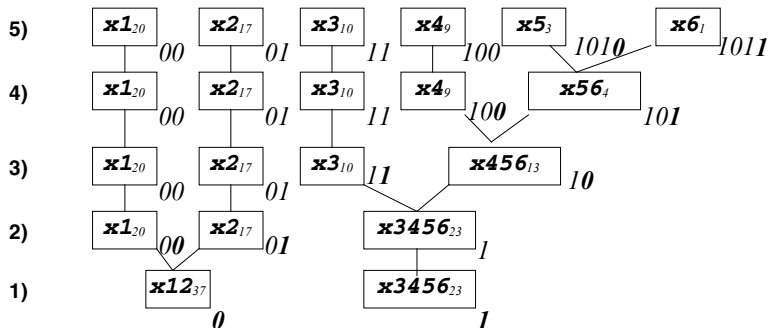
Rysunek przedstawia sześć etapów redukcji kodowanego alfabetu (proszę nie sugerować się postacią rysunku, to jeszcze nie jest drzewo binarne!). Znaki x_5 i x_6 występują najrzadziej, zatem redukujemy je do zastępczego znaku, który oznaczmy jako x_{56} . Podobnie czynimy w każdym kolejnym

Rysunek 13.6.
Konstrukcja
kodu Huffmana
— faza redukcji



etapie, aż dojdziemy do momentu, w którym zostają nam tylko dwa znaki alfabetu (zastępcze). Faza *redukcji* została zakończona i możemy przejść do fazy *konstrukcji kodu*. Popatrzmy w tym celu na rysunek 13.7.

Rysunek 13.7.
Konstrukcja kodu
Huffmana — faza
tworzenia kodu



Bity 0 i 1, które są dokładane na danym etapie do zredukowanych znaków alfabetu, są **wyłuszczone**. Mam nadzieję, że czytelnik nie będzie miał zbyt wielu problemów z odtworzeniem sposobu konstruowania kodu, tym bardziej że nasz przykład bazuje na krótkim alfabecie.

Przy klasycznej metodzie kodowania binarnego komunikat o długości 60 znaków (napisany z użyciem 6-znakowego alfabetu) zakodowalibyśmy za pomocą $60 \cdot 3 = 180$ bitów. Przy użyciu kodu *Huffmana* ten sam komunikat zająłby odpowiednio: $20 \cdot 2 + 17 \cdot 2 + 10 \cdot 2 + 9 \cdot 3 + 3 \cdot 4 + 1 \cdot 4 = 137$ znaków (zysk wynosi ok. 23%).

Wiemy już, jak konstruować kod, warto zastanowić się nad implementacją programową w C++. Nie chciałem prezentować gotowego programu kodującego, gdyż zająłby on zbyt dużo miejsca. Dobrą metodą byłoby skopiowanie struktury graficznej przedstawionej na dwóch ostatnich rysunkach. Jest to przecież tablica 2-wymiarowa, o rozmiarze maksymalnym 6×5 . W jej komórkach trzeba by było zapamiętywać dość złożone informacje: kod znaku, częstotliwość jego występowania, kod bitowy. Z zapamiętaniem tego ostatniego nie byłoby problemu, możliwości w C++ jest dość sporo: tablica 0-1, liczba całkowita, której reprezentacja binarna byłaby tworzonym kodem itp. Podczas kodowania nie należy również zapominać, aby wraz z kodowanym komunikatem posłać jego... kod *Huffmana*! (Inaczej odbiorca miałby pewne problemy z odczytaniem wiadomości).

Problemy techniczne jest zatem dość sporo. Oczywiście zaproponowany powyżej sposób wcale nie jest obowiązkowy. Bardzo interesujące podejście (wraz z gotowym kodem C++) prezentowane jest w [Sed92]. Autor używa tam kolejki priorytetowej do wyszukiwania znaków o najmniejszych prawdopodobieństwach, ale to podejście z kolei komplikuje nieco proces tworzenia kodu binarnego na podstawie zredukowanego alfabetu. Zaletą algorytmów bazujących na „stertopodobnych” strukturach danych jest jednak niewątpliwie ich efektywność: operacje

na stercie są bowiem klasy $\log N$, co ma wymierne znaczenie praktyczne! Popatrzmy zatem, jak można wyrazić algorytm tworzenia drzewa kodowego *Huffmana* właśnie przy użyciu tych struktur danych:

```
Huffman(s, f)
// s — kodowany binarny ciąg znaków
// f — tablica częstotliwości występowania znaków w alfabecie
{
  wstaw wszystkie znaki ciągu s do sterty H
  stosownie do ich częstotliwości;
  dopóki H nie jest pusta wykonuj
  {
    jeżeli H zawiera tylko jeden znak X to
      X staje się korzeniem drzewa Huffmana T;
    w przeciwnym przypadku
    {
      • weź dwa znaki X i Y z najmniejszymi częstotliwościami  $f_1$  i  $f_2$  i usuń je ze sterty H;
      • zastąp X i Y znakiem zastępczym Z, którego częstotliwość występowania wynosi  $f=f_1 + f_2$ ;
      • wstaw znak Z do kolejki H;
      • wstaw X i Y do drzewa T jako potomków Z;
    }
  }
  zwróć drzewo T;
}
```

Algorytm ten jest oczywiście równoważny podanemu wcześniej, zmieniliśmy tylko formę zapisu.

Zachęcam Czytelnika do głębszych studiów teorii kodowania i informacji, gdyż są to bardzo ciekawe zagadnienia o dużym znaczeniu praktycznym. Z braku miejsca nie mogłem podjąć wielu interesujących wątków, poza tym pewne zagadnienia trudno przełożyć na łatwy do zrozumienia kod C++. Proszę zatem potraktować ten rozdział jako wstęp, za którym kryje się bardzo rozległa i ciekawa dziedzina wiedzy!



Definicja

W archiwum ZIP dostępnym na *ftp* w pliku *huf.zip*, znajdują się programy kodujące i dekodujące w algorytmie *Huffmana* autorstwa Shaun Case. Są to programy na licencji *public domain*, ściągnięte przez *ftp* z Internetu. Pliki są dostarczone w nietkniętej postaci i mogą wymagać dostosowania do konkretnej wersji kompilatora C++. (Oryginalnie są napisane w języku C dla kompilatora Borland C++ 2.0, program daje się jednak bez problemu kompilować w innych kompilatorach tej firmy, np. Borland C++ Builder 5.0; aby skompilować w Visual C++, przeczytaj plik *huf.txt*).

Kodowanie LZW

Kod *LZW* jest dobrym i znanym przykładem na *kodowanie za pomocą słownika*, w którym przechowywane są często powtarzane wzorce. Zaletą takiego słownika jest dopasowanie do zawartości tekstu, w przeciwieństwie do dość statycznego podejścia prezentowanego w kodowaniu tabelkowym (patrz kod ASCII czy Morse'a), w którym bardziej interesujemy się alfabetem danego języka niż tekstem do zakodowania.

Metoda kodowania *LZW* została wymyślona w 1977 i 1978 roku przez Abrahama Lempela i Jacoba Ziva, a następnie udoskonalona w roku 1984 przez Terry'ego Welch¹⁴ — z pierwszych liter nazwisk twórców bierze się nazwa metody. Jest ona używana m.in. w systemie UNIX (funkcja *compress*) oraz kompresji obrazów *GIF* (ang. *Graphics Interchange Format*). Warto także wiedzieć, że algorytm *LZW* od 1995 roku jest chroniony patentem — jego właścicielem jest firma *Unisys*

¹⁴ Terry A. Welch, „A Technique for High Performance Data Compression”, *IEEE Computer*, Vol. 17, No. 6, 1984, pp. 8 – 19.

Corporation. Producenci, którzy chcą sprzedawać oprogramowanie używające algorytmu *LZW*, muszą uzyskać płatną licencję od *Unisys Corporation*, zwolnieni z opłat są tylko użytkownicy końcowi i organizacje niekomercyjne.

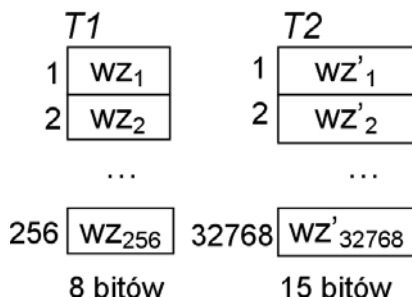
Powstała tu ciekawa sytuacja, gdyż o ile specyfikacja formatu *GIF* jest bezpłatna¹⁵, to z uwagi na używanie do niej algorytmu *LZW* (płatnego) de facto stała się ona standardem płatnym!

Idea kodowania słownikowego na przykładach

Metoda kodowania za pomocą słownika nie jest zbyt skomplikowana. Budujemy w niej dwa słowniki dla kodowania wzorców często występujących w kodowanym tekście (liczba wzorców jest skończona i generalnie niezbyt duża). Słownik ten zapiszemy w tablicy *T1*. Do kodowania pozostałych wzorców (elementów) występujących w tekście służy słownik *T2* (jest to de facto funkcja tłumacząca, a nie realny słownik z alokacją pamięci).

Przykład znajduje się na rysunku 13.8. Zakładam na nim, że znaków kodowych jest tyle, iż do ich zakodowania jest potrzebnych 15 bitów — stąd wszystkich możliwych wartości jest właśnie 2^{15} (patrz *T2*). Wiedząc jednak (a nie miejsce tu, aby zastanawiać się, skąd płynie ta wiedza), że w danym tekście najczęściej występuje 256 ciągów kodowych *WZ*₁, ..., *WZ*₂₅₆, pogrupowaliśmy je w rzeczywistej tablicy *T1*. Każdy często występujący ciąg zabierze nam w zakodowanym tekście 8 bitów (plus może jakiś specjalny bit kontrolny, po którym rozpoznamy ten ciąg) i analogicznie — ciąg zaklasyfikowany jako rzadko występujący zostanie zakodowany na 15 bitach (plus ewentualnie 1 bit rozpoznawczy).

Rysunek 13.8.
Słownikowa metoda
kodowania



Zysk kompresji w metodzie słownikowej zależy ściśle od tego, jak dużo ciągów kodowych zostanie rozpoznanych jako często występujące (i w konsekwencji zajmą mniej miejsca w kodzie) w stosunku do wszystkich. Zatem musimy jakoś doprowadzić do tego, aby algorytm uzyskał wystarczającą wiedzę o kodowanym tekście, co pozwoli na skonstruowanie takiej tablicy.

Metoda *LZW* jest elegancką implementacją algorytmu kodowania słownikowego, w którym algorytm samoczynnie buduje swój słownik najczęściej występujących znaków. Popatrzmy na zapis algorytmu kodowania *LZW* na przykładzie prostego ciągu kodowego. (Przykład ten powinien wyjaśnić algorytm lepiej niż jakiegokolwiek dywagacje teoretyczne).

Zanim przejdziemy do samego algorytmu, chciałbym podsumować jego główne cechy:

- ◆ Nie istnieje predefiniowany słownik — jest on dynamicznie tworzony dla każdego ciągu wejściowego (dotyczy kodowania i dekodowania).
- ◆ Algorytm jest jednoprzebiegowy!

¹⁵ Jej właścicielem jest CompuServe Incorporated, w 1997 roku wchłonięta przez AOL (America Online).

Kodowanie w metodzie *LZW* można opisać algorytmem w pseudokodzie:

```
z = NULL;
while(czytaj znak k)      // aż strumień wejściowy się wyczerpie
{
    if (zk ∈ SŁOWNIK)      // zk jest ciągiem złożonym z konkatencji z i k
        z = zk;
    else
    {
        dodaj zk do SŁOWNIKA
        wypisz kod dla z;
        z = k; // ostatni znak jest początkiem nowego ciągu
    }
}
```

Zapis powyższy oznacza, że koder (urządzenie — algorytm kodujący) gromadzi symbole z ciągu wejściowego, tworząc z nich pewien ciąg *s*, tak długo, dopóki *s* jest ciągle elementem słownika. W przeciwnym razie dodajemy *s* do słownika i kontynuujemy algorytm, inicjalizując *s* ostatnim jego znakiem.

Przykład:

Załóżmy, że znaki podstawowe są kodowane od 0 do 255 (kod ASCII), a na znaki symboli dodatkowych przeznaczymy w słowniku indeksy od 256 w górę.

Strumień wejściowy: ^MNO^MN^MNN^MNP^MNQ

Z	k	WYJŚCIE	index	symbol	Pozycja w pliku
NULL	^				
^	M	^	256	^M	^MNO^MN^MNN^MNP^MNQ
M	N	M	257	MN	^MNO^MN^MNN^MNP^MNQ
N	O	N	258	NO	^MNO^MN^MNN^MNP^MNQ
O	^	O	259	O^	^MNO^MN^MNN^MNP^MNQ
^	M				
^M	N	256	260	^MN	^MNO^MN^MNN^MNP^MNQ
N	^	N	261	N^	^MNO^MN^MNN^MNP^MNQ
^	M				
^M	N				
^MN	N	260	262	^MNN	^MNO^MN^MNN^MNP^MNQ
N	^				
N^	M	261	263	N^M	^MNO^MN^MNN^MNP^MNQ
M	N				
MN	P	257	264	MNP	^MNO^MN^MNN^MNP^MNQ
P	^	P	265	P^	^MNO^MN^MNN^MNP^MNQ
^	M				
^M	N				
^MN	Q	260	266	^MNQ	^MNO^MN^MNN^MNP^MNQ
Q	NULL	Q			

Zaczynamy od początku pliku (NULL). Następny znak przychodzący (^) znajduje się w słowniku, wypisujemy go (WYJŚCIE) i idziemy dalej: aktualny ciąg *s*=^M. Ciagu tego nie ma w słowniku, zatem dodajemy go na pierwszy wolny indeks (256). Bierzymy ostatni znak *s*, tj. M, sklejamy go ze znakiem kolejnym (N) — tego ciągu także nie ma w słowniku, i dodajemy go. W ten sam sposób dodajemy do słownika kolejne kody. Najciekawsza sytuacja ma miejsce, gdy coraz dłuższe teksty zaczynają się powtarzać i zastępujemy je zdefiniowanymi wcześniej kodami, uzyskując efekt kompresji!

Algorytm dekodujący nie jest skomplikowany. Rozpoczynamy go z identycznym słownikiem wejściowym, jak w algorytmie kodującym.

Dekodowanie LZW:

```
czytaj znak k:    // pierwszy znak
wypisz k;
w = k;            // pierwszy kod
while (czytaj k)  // aż strumień wejściowy się wyczerpie
{
    s = zawartość słownika dla k;
    wypisz s;
    dodaj w + s[0] do SŁOWNIKA;
    w = s;
}
```

Strumień wejściowy: ^MNO<256>N<260><261><257>P<260>Q

w	k	wyjście	index	symbol
	^	^		
^	M	M	256	^M
M	N	N	257	MN
N	O	O	258	NO
O	<256>	^M	259	O^
<256>	N	N	260	^MN
N	<260>	^MN	261	N^
<260>	<261>	N^	262	^MNN
<261>	<257>	MN	263	N^M
<257>	P	P	264	MNP
P	<260>	^MN	265	P^
<260>	Q	Q	266	^MNQ

Zauważmy, że algorytm, dekodując, jednocześnie odtwarza taki sam słownik, jak algorytm kodujący, i używa go w trakcie dekodowania!

Opis formatu GIF

Celem tego podpunktu jest pokazanie, jak w praktyce stosowana jest metoda kompresji *LZW* w standardzie przemysłowym w formacie pliku graficznego *GIF*, królującego w Internecie.

Format *GIF* występuje w dwóch wariantach: *GIF87a* i *GIF89a* (patrz pierwsze 6 znaków dowolnego pliku w tym formacie), ten drugi pozwala na zapis animacji i przeźroczystości. Warto wiedzieć, iż format *GIF* powstał na bazie założenia, iż paleta kolorów¹⁶ wynosi maksymalnie 256 kolorów (2, 4, 8, 16, 32, 64, 128 lub 256), co powoduje utratę informacji po konwersji grafiki w formacie *True Color* (24 bity) na ten format — 8 bitów¹⁷.

Jednak w wielu przypadkach redukcja palety kolorów ma mniejsze znaczenie niż zalety związane ze zmniejszeniem wielkości plików graficznych. A jak łatwo zauważyć, w Internecie grafiki wbudowane w strony HTML używają często właśnie formatu *GIF*.

Działanie algorytmu kompresji wykorzystuje algorytm *LZW*. W obrazie odszukiwane są sekwencje pikseli, np. ciągi punktów o tym samym kolorze. Sekwencje te są zastępowane odpowiednimi kodami liczbowymi umieszczonymi w tablicy. Im więcej takich powtarzających się sekwencji, tym lepszy uzyskujemy efekt kompresji, dzięki czemu rozmiar pliku jest mniejszy, ale oczywiście silnie zależy to od samego obrazu!

¹⁶ Paleta jest dla komputera tablicą kodów, pozwalającą zinterpretować kod znakowy w pliku graficznym do postaci widocznej na ekranie.

¹⁷ Przy konwersji z formatu *True Color* na *GIF* dokonane zostanie odwzorowanie kolorów z dużej palety na mniejszą.

Format pliku *GIF* jest przedstawiony w tabeli 13.5, oprócz samych danych obrazu (skompre-sowanych) uwzględnia on wszelkie dodatkowe informacje niezbędne do wyświetlenia obrazu na fizycznym urządzeniu, jakim jest ekran.

Tabela 13.5. *Format pliku GIF*

Blok	Zawartość
Nagłówek	6 bajtów, „GIF87a” lub „GIF89a”
Opis ekranu logicznego (deskryptor ekranu) — 7 bajtów	
Szerokość ekranu	2 bajty
Wysokość ekranu	2 bajty
Znacznik globalnej mapy kolorów	1 bit (1 — jest, 0 — nie ma)
Rozdzielczość kolorów	3 bity
Znaczniki sortowania kolorów w mapie	1 bit (gdzie 1, to narastająco, kolor najczęściej występujący jako pierwszy)
Rozmiar globalnej mapy kolorów	3 bity, rozmiar = (2^{n+1})
Kolor tła	1 bajt — indeks koloru tła w globalnej mapie kolorów
Współczynnik kształtu piksela	1 bajt, 0 — piksele kwadratowe, jeśli wartość niezerowa, to współczynnik szerokości do wysokości określa proporcja $(n+15)/64$
Globalna mapa kolorów	(Opcja), każdy kolejny kolor jest kodowany trzema składowymi RGB
Blok deskryptora obrazu	
Separator	1 bajt (znak przecinka, 0x2c)
Odległość od lewej strony ekranu	unsigned (2 bajty)
Odległość od góry ekranu	unsigned (2 bajty)
Szerokość	unsigned (2 bajty)
Wysokość	unsigned (2 bajty)
Znacznik lokalnej mapy kolorów	1 bit (1 — tak, 0 — nie)
Znacznik przeplotu	1 bit (1 — tak, 0 — nie)
Znacznik sortowania kolorów w mapie	1 bit
<Zarezerwowane>	2 bity
Rozmiar lokalnej mapy kolorów	3 bity, rozmiar = $(2n+1)$
Lokalna mapa kolorów (nadpisuje mapę globalną, dotyczy obrazka za nią występującego)	
Dane pikslowe	Pojedynczy obraz — w jednym pliku <i>GIF</i> może znajdować się od 1 do n obrazów
...	
Znacznik końca	

Słowo komentarza dla pojęcia „przeplotu obrazu”. Otóż kolejne piksele są zapisywane w kolejności od lewej do prawej i od góry do dołu. Przy wolnych łączach telekomunikacyjnych możliwe jest zastosowanie sztuczki optycznej, polegającej na stopniowym uzupełnianiu obrazu o kolejne brakujące linie. W tym celu stosuje się przeplatanie (zmianę kolejności) zapisu linii wg schematu:

- ♦ W *pierwszym* przejściu zapisywana jest co ósma linia, poczynając od zerowej.
- ♦ W *drugim* przejściu zapisywana jest co ósma linia, poczynając od czwartej.
- ♦ W *trzecim* przejściu zapisywana jest co czwarta linia, poczynając od drugiej.
- ♦ W *czwartym* przejściu co druga linia, poczynając od linii pierwszej.

Zapis z przeplotem pozwala na pokazanie w miarę czytelnego obrazka już po odebraniu jednej ósmej obrazu!

Pierwszy bajt w skompresowanej postaci obrazu określa minimalną liczbę b bitów na piksel w obrazie oryginalnym. Wartość 2^b jest tzw. kodem oczyszczającym (ang. *clear code*), służącym do nadania wszystkim parametrom kompresji i dekompresji wartości początkowych. Początkowy rozmiar słownika jest równy 2^{b+1} , jest on podwajany w miarę potrzeby aż do osiągnięcia maksymalnego rozmiaru 4096.

Słowa kodowe generowane przez LZW w GIF przechowywane są w postaci bloków znaków o długości 8 bitów. Maksymalny rozmiar bloku wynosi 255 — każdy blok poprzedzony jest nagłówkiem, w którym znajduje się informacja o jego rozmiarze. Gdy bajt długości podbloku ma wartość 0×00 , oznacza to koniec bloku danych (tzw. *terminator bloku*).



Patrz także

Szczegółowy opis formatu GIF znajduje się na serwerze FTP w pliku *GIF 89a.txt* (<ftp://ftp.helion.pl/przyklady/algo4.zip>).

Rozdział 14.

Zadania różne

W tym rozdziale została zamieszczona grupa zadań, które nie zmieściły się w rozdziałach poprzednich. Są to proste wprawki programistyczne o dość atrakcyjnej tematyce — ich rozwiązanie może stanowić test na sprawność w stawianiu czoła codziennym zadaniom programistycznym. Niektóre zadania nie posiadają rozwiązań, sądzę jednak, że ich prostota powinna usprawiedliwić ten zabieg.

Teksty zadań

Zadanie 1.

Tzw. *sito Erastotenesa* jest jedną ze starszych metod otrzymywania liczb pierwszych (tzn. tych, które dzielą się tylko przez siebie i przez 1). Algorytm polega na następującym odsiewie liczb:

- ♦ wypisać ciąg liczb naturalnych:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ..., N

- ♦ usunąć z nich liczby większe od 2 i podzielne przez 2:

1, 2, 3, *, 5, *, 7, *, 9, *, 11, *, 13, *, 15... N

- ♦ usunąć z nich liczby większe od 3 i podzielne przez 3:

1, 2, 3, *, 5, *, 7, *, *, *, 11, *, 13, *, *... N

- ♦ usunąć z nich wielokrotności liczby 5 (kolejna najmniejsza liczba), 7...

Napisz program, który:

- ♦ Sprawdza metodą *brute-force*, czy dana liczba jest liczbą pierwszą.
- ♦ Wykorzystując metodę *sita Erastotenesa*, liczy wszystkie liczby pierwsze mniejsze od 100.

Zadanie 2.

Napisz funkcję, która otrzymując na wejściu datę zakodowaną w postaci liczby całkowitej (np. 220744), wypisze słownie jej znaczenie (tutaj: „22 lipca 1944”).

Zadanie 3.

W operacjach macierzowych są często używane tablice z dużą liczbą zer. Reprezentowanie ich w postaci dwuwymiarowej wydaje się marnotrawstwem pamięci. Spróbuj zaproponować strukturę danych, która będzie zawierała tylko informację o „współrzędnych” elementów

niezerowych. Zakładamy, że wszystkie pozostałe liczby, niezaprezentowane w niej, są zerowe. Zaproponuj funkcje obsługujące taką strukturę danych: wypisujące macierz w formie odkodowanej, dodające i mnożące dwie macierze, etc.

Spróbuj określić w przybliżeniu, do jakiego stopnia zapęnlennia tablicy zerami taka struktura danych jest opłacalna, jeśli chodzi o zużycie pamięci.

Zadanie 4.

Zaproponuj dwie wersje rekurencyjnego algorytmu obliczania funkcji x_n (rekurencja „naturalna” i rekurencja „z parametrem dodatkowym”).

Zadanie 5.

Spróbuj stworzyć *nierekurencyjną* funkcję, która na podstawie dwóch list posortowanych jako wynik zwróci listę posortowaną, zawierającą wszystkie ich elementy. Wymóg: nie wolno tworzyć nowych komórek pamięci, jedyne, co jest dozwolone, to manipulacja wskaźnikami.

Zadanie 6.

Napisz funkcję, która na podstawie ceny podanej w postaci liczby całkowitej typu np. `long` wydrukuje ją w postaci słownej. Przykład: wywołanie `cena_slownie(12304)` powinno dać w rezultacie tekst: „dwanaście tysięcy trzysta cztery zł”.

Zadanie 7.

Napisz program, który realizuje permutację cykliczną danej tablicy wejściowej o zadaną liczbę pozycji. Spróbuj przeprowadzić dokładną analizę problemu, wybierając technikę programowania: rozwiązanie iteracyjne, rekurencyjne — jeśli tak, to jakiego typu?

Zadanie 8.

Napisz program liczący w najprostszy sposób liczbę wystąpień danego słowa w tekście wejściowym.

Zadanie 9.

Napisz program obliczający w najprostszy możliwy sposób dane statystyczne tekstu wejściowego: liczbę wystąpień każdej litery, słowa, etc.

Zadanie 10.

Napisz program sprawdzający, czy zdanie wejściowe jest palindromem (tzn. czy da się czytać tak samo z lewej do prawej, jak i z prawej do lewej strony). Przykład takiego zdania: „kobyła ma mały bok”.

Zadanie 11.

Napisz program sprawdzający, czy zdanie wejściowe zawiera ukryte słowo, np. tekst „**Bronek** alergicznie nie znośł makaronu z **kaszą**” ukrywa słowo „bramka”.

Zadanie 12.

Napisz funkcję obliczającą wyrażenia postaci: $2+2+1$, $1+2*3$, etc. podane w postaci wskaźnika typu `char*`. (Funkcją biblioteczną C++, która zamienia ciąg znaków na liczbę zmiennopozycyjną, jest `atof`, ale jej działanie zatrzymuje się na pierwszym znaku tekstu, który nie jest cyfrą).

Rozwiązania

Zadanie 1.

Do rozwiązania zadania (a) będziemy potrzebowali funkcji zwracającej nam pierwszą całkowitą liczbę x spełniającą warunek $x^2 < n$:



erastot.cpp

```
inline int sqrt_int(int n)
{
    return (int)sqrt((double)n)/1;
}
```

(Wykorzystujemy fakt, że dzielenie całkowite w C++ obcina część ułamkową).

Odpowiedź na pytanie, „czy n jest liczbą pierwszą?”, sprowadza się do sprawdzenia, czy istotnie dzieli się ona tylko przez 1 i przez siebie samą:

```
bool pierwsza(int n)      // czy n jest liczbą pierwszą?
{
    int i, limes=sqrt_int(n);
    for(i=2; n!=(n/i)*i && i<=limes;i++);
    if (i>limes)
        return true;  // tak, liczba pierwsza
    else
        return false; // nie, „zwyčajna” liczba
}
```

Nieco bardziej skomplikowana jest realizacja *sita Erastotenesa* (b).

Problemem jest konieczność deklaracji dużych tablic, ale jest to jedyna wada tej prostej metody:

```
void sito(int n) // wypisuje wszystkie liczby pierwsze < n
{
    int cpt = 1, i, *tp=new int[n+1];

    for(i=1; i<=n; i++) t
        p[i]=i; //zaznaczenie wszystkich liczb naturalnych od 1 do n

    while(cpt<n)
    { //szukamy pierwszego niezerowego elementu tablicy tp:
        for(cpt++; (tp[cpt]==0) && (cpt!=100); cpt++);
        int k=2; //zerujemy wielokrotności tego elementu (cpt) w tp

        while(cpt*k<=n)
        {
            tp[cpt*k]=0;
            k++;
        }
    }

    for(i=1; i<=n; i++)
        if (tp[i]!=0) cout << "Liczba pierwsza:"<< tp[i] <<endl;
    delete tp; //usunięcie tablicy pamięci
}
```

Zadanie 3.

Struktura danych obsługująca „tablicę zer” może mieć postać następującej listy:

```
struct zero_lst
{
    int x,y;
```

```
int val;           // lub inny dowolny typ danych
struct zero_lst *nastepny;
}
```

Zakładając, że wskaźnik następny zajmuje dwa bajty pamięci (podobnie jak i zmienne typu `int`), dowolny element listy zajmuje $p = 2+2+2+2 = 8$ bajtów pamięci. Z drugiej zaś strony w przypadku klasycznej tablicy pojedynczy element kosztuje nas tylko 2 bajty (jest to zmienna typu `int`), ale za to z góry musimy przydzielić pamięć na całą tablicę.

Oznaczmy rozmiar tablicy przez N . Wówczas całkowita zajęta przez nią pamięć wynosi $2N^2$ bajtów. „Magiczną” granicę k , przy której sens stosowania listy jest wątpliwy, można z łatwością obliczyć za pomocą równości: $k \cdot p = 2N^2$. Przykładowo: dla $p = 8$, $N = 10$, dwudziesty szósty niezerowy element już przebiera miarę. Praktycznie rzecz ujmując, typowa liczba niezerowych elementów powinna być *znacznie mniejsza* od $\frac{2N^2}{p}$ — programista musi sam podjąć decyzję co do właściwej interpretacji słowa „znacznie”.

Zadanie 4.

Dwie wersje programów rekurencyjnych służących do obliczania x^n znajdują się poniżej:



pot.cpp

```
int pot1(int x, int n)
{
    if (n==0)
        return 1;
    else
        return (pot1(x,n-1)*x);
}

int pot2(int x, int n, int temp=1)
{
    if (n==0)
        return temp;
    else
        return (pot2(x,n-1,temp*x));
}

int main()
{
    cout << "Dwa do potęgi trzeciej\n";
    cout << "Metoda 1\t" << pot1(2,3)<< "\n";
    cout << "Metoda 2\t" << pot2(2,3)<< "\n";
}
```

Zadanie 10.

Zadanie należy do elementarnych, nie powinno zatem nikomu sprawić trudności dojście do następującego rozwiązania:



palindro.cpp

```
void palindrom(char *s)
{
    int dl=strlen(s), cpt=0;
    bool test=true; // 's' jest (na razie) palindromem
    while( (cpt<=dl/2) && (test==true) )
        if(s[cpt]==s[dl-cpt-1])
            cpt++;
}
```

```

    else
        test=false;
    cout << s;
    if(test==true)
        cout << " ...jest palindromem\n";
    else
        cout << " ...jest zwykłym słowem...\n";
}

```

Zadanie 12.

Problem obliczania wartości wyrażeń zapisanych w postaci słownej występuje dość często w praktyce programistycznej. Zadanie jest ogólnie dość skomplikowane i warto dobrze je przemyśleć. Niech zatem rozwiązanie, które przedstawiam poniżej, posłuży raczej za przyczynek do rozważań niż gotowy wzorec. Tym bardziej że dla pewnych konfiguracji danych wyrażenie jest obliczane źle!



oblicz.cpp

```

double transl(char *s)
// zamienia ciągi znaków typu 1+1 na 2, 1+1+2*5 na 12, etc.
// uwaga: funkcja nie analizuje dzielenia i badania przypadku
// dzielenia przez zero!
{
    int i,n;
    char *s1;
    n=strlen(s);
    s1= new char[n+1]; // kopia robocza tekstu wejściowego
    strcpy(s1,s);      // kopia ciągu wejściowego

    for(i=0;i<n;i++)    // szukamy znaków + i *
        if(s[i]=='+' || s[i]=='*')
        {
            s1[i]='\0';
            if(s[i]=='+')
                return transl(s1)+transl(s+i+1);
            else
                return transl(s1)*transl(s+i+1);
        }
    // Przypadek elementarny:
    delete s1;
    return atof(s); // atof = „ascii to float”
}

int main()
{
    cout << "1+1=" << transl("1+1") << endl;           // 2 OK
    cout << "2*2*3=" << transl("2*2*3") << endl;         // 12 OK
    cout << "2+2*3=" << transl("2+2*3") << endl;         // 8 OK
    cout << "2+2+3=" << transl("2+2+3") << endl;         // 7 OK
    cout << "2+2*0=" << transl("2+2*0") << endl;         // 2 OK
    cout << "2*3+4*5=" << transl("2*3+4*5") << endl;     // 46 źle!
}

```

Proszę się zastanowić, dlaczego funkcja transl źle obliczyła ostatnie wyrażenie? (Wskazówka: proszę odtworzyć kierunek analizy wyrażenia.)

Dla zaawansowanych programistów C++: proszę przeanalizować zarządzanie pamięcią w funkcji transl. Czy użycie new i delete jest na pewno optymalne w tym przypadku?

Dodatek A

Poznaj C++ w pięć minut!

Dodatek ten w swoim założeniu stanowi pomost dla programistów pascalowych, którzy chcą szybko i bezboleśnie poznać podstawowe elementy języka C++, tak by lektura książki nie napotykała bariery niezrozumienia na poziomie użytej składni. Materiał ten celowo został umieszczony w dodatku, bowiem nie wchodzi on w zasadniczy nurt książki.

Rozdział ten nie zastąpi z pewnością monograficznego podręcznika poświęconego językowi C++, nie to jest jednak jego celem. Poniższy szybki kurs języka C++ obejmuje tylko te elementy, które są konieczne do zrozumienia prezentowanych listingów. Jest to niezbędne minimum, zorientowane wyłącznie na składnię. Jeśli poważnie interesujesz się programowaniem w C++, zainwestuj w dobry podręcznik, np. [Eck02] lub [Str04].

Elementy języka C++ na przykładach

Kolejne sekcje zawierają serię przykładów, na podstawie których osoba znająca język Pascal może na zasadzie analogii poznać podstawowe zasady zapisu algorytmów w C++.

Pierwszy program

Spójrzmy na poniższy program z doskonale znanego gatunku hello world:

```
program pr1: {komentarz}                                #include <iostream>
begin                                                    using namespace std;
writeln('Witaj!')                                       int main() //komentarz
                                                         {
end.                                                     cout << "Witaj!\n";
                                                         }
```

- ◆ Blok w C++ jest ograniczany przez nawiasy klamrowe { }.
- ◆ Działanie programu rozpoczyna się od *funkcji* o nazwie main. W C++ funkcja ta zwraca zawsze wartości typu int, stąd bierze się konieczność poprzedzenia jej nazwy tym typem. W starszym kodzie można było spotkać czasami typ void, czyli informację o funkcji niezwracającą wartości w sensie arytmetycznym.
- ◆ Tekst w C++ można wypisać, wysyłając do standardowego wyjścia (cout) ciąg znaków ograniczony przez cudzysłów ("tekst"). Sekwencja \x oznacza znak specjalny, np. \n to skok do nowej linii podczas wypisywania tekstu na ekranie, \t — znak tabulacji, etc.
- ◆ W C++ komentarz // obowiązuje do końca wiersza. Chcąc coś napisać w komentarzu pomiędzy instrukcjami, użyj raczej /* komentarz */ niż // komentarz.

Dyrektywa #include

Symbol # zawarty w kodzie C++ oznacza dyrektywę, czyli polecenia dla kompilatora. Język C++ pozwala na zaawansowane manipulowanie treścią kodu źródłowego, ale dla większości programistów wystarczającą jest na początek wiedza o konieczności użycia dyrektywy `#include <iostream>`, która oznacza dołączenie, jeszcze przed właściwą kompilacją, całej zawartości pliku *iostream* do pliku z programem. Plik ten jest wymagany, gdy zamierzamy używać strumieni `cout`, `cin`, `cerr`, które odpowiadają standardowemu wyjściu (np. ekran), wejściu (np. klawiatura) oraz miejscu, do którego należy wysyłać komunikaty o błędach. To ostatnie zazwyczaj odpowiada ekranowi.

Dyrektywa towarzysząca: `using namespace std` oznacza „odsłonięcie” deklaracji i nazw zawartych w standardowej bibliotece C++. Wiedza o tym, czym są „przestrzenie nazw”, przydatna jest w nieco dalszym etapie programowania w C++, wykraczającym poza proste algorytmy zawarte w tej książce, teraz po prostu przyjmij do wiadomości, że programy nie będą działały, gdy tego nie zrobisz.



W dalszych przykładach obie te dyrektywy będą dla oszczędności miejsca pomijane, ale nie zapomnij o nich w prawdziwych programach. Warto też wiedzieć, że w starszych programach w C++ stosowana była tylko instrukcja `include <iostream.h>`, obecnie kompilatory już nie akceptują tej formy.

Pliki dołączane zawierają zazwyczaj deklarację często używanych stałych i typów. Plik dołączany opisany w nawiasach `<>` jest plikiem standardowym z biblioteki kompilatora. Pliki, których nazwa jest otoczona cudzysłowem `" "`, są plikami stworzonymi przez programistę, zazwyczaj są one umieszczane w tym samym katalogu, co pliki z kodem C++.

Podprogramy

W języku C++, podobnie zresztą jak i w klasycznym C, wszystkie podprogramy są nazywane *funkcjami*. Odpowiednikiem znanej z Pascala *procedury* jest funkcja, która niczego nie zwraca — używamy słowa `void`.

Procedury

Oto przykład definicji i użycia procedur w Pascalu i C++:

<pre> program pr9; procedure procl(a,b:integer; var m:integer) {zmienna lokalna;} var c:integer; begin c:=a+b; writeln(c); m:=c*a*b end; var i,j,k:integer; begin i:=10; j:=20; procl(i,j,k) end.</pre>	<pre> void procl(int a, int b, int& m.) { //zmienna lokalna: int c; c=a+b; cout << c << endl; m=c*a*b; } int i,j,k; int main() { i=10; j=20; procl(i,j,k); }</pre>
--	---

- ♦ C++ nie umożliwia tworzenia procedur i funkcji lokalnych.
- ♦ Zdefiniowane funkcje i procedury są ogólnie dostępne w całym programie.
- ♦ Odpowiednikiem deklaracji typu `var` w nagłówku funkcji jest w C++ zasadniczo tzw. referencja (&), np. `Fun(var i:integer;...)` jest równoważne funkcjonalnie formie `Fun(int& i,...)`.
- ♦ W C++ tablice są z założenia przekazywane przez adres. Przykładowo zapis `Fun(int tab[3])` oznacza chęć użycia jako parametru wejściowego tablicy elementów typu `int`. Podczas wywołania funkcji `Fun` tablica użyta jako parametr jest przekazywana poprzez swój adres i jej zawartość może być fizycznie zmodyfikowana (patrz też strona 300).

Funkcje

Zasadnicze różnice pomiędzy funkcjami w C++ i w Pascalu dotyczą sposobu zwracania wartości:

<pre>program pr10; function plus2(a:integer):integer; begin plus2:=a+2 end; var i:integer; begin i:=10; writeln(plus2(i)) end.</pre>	<pre>int plus2(int a) { return a+2; } int i; int main() { i=10; cout<<plus2(i)<<endl; }</pre>
--	--

- ♦ W C++ instrukcja `return(v)` powoduje natychmiastowy powrót z funkcji z wartością `v`. Przykładowo: po instrukcji `if(v) return(val)` nie trzeba używać `else`¹ — w przypadku prawdziwości warunku `v` ewentualna dalsza część procedury już nie zostanie wykonana.
- ♦ Dobrym zwyczajem programisty jest używanie tzw. *nagłówków funkcji*, czyli informowanie kompilatora o swoich intencjach, o typach parametrów wejściowych i wyjściowych. Nagłówek funkcji jest tym wszystkim, co zostaje z funkcji po usunięciu z niej jej definicji i *nazw* parametrów wejściowych. Przykładowo: jeśli gdzieś w programie jest zdefiniowana funkcja:

```
void f(int k, char* s[3]){...tutaj kod...},
```

to tuż za dyrektywami `#include` możemy dopisać linię:

```
void f(int, char*[]); // tu średnik!
```

Deklaracje nagłówkowe często grupuje się w plikach `.h` dołączanych dyrektywą `#include` jest to zwłaszcza praktykowane w przypadku definiowania tzw. klas (patrz strona 305). Nagłówki są bardzo ważne, gdyż pozwalają już na etapie wstępnych kompilacji uniknąć wielu błędów związanych z wywołaniem funkcji ze złymi parametrami. Notabene, niektóre kompilatory z założenia nie tolerują ich braku, nie pozwalając na skomplikowanie kodu z nieznaną sobie funkcją, czyli taką, której definicji lub deklaracji nagłówka nie odnalazły w pierwszym „przebiegu” procesu kompilacji.

Oto przykład błędnego (bez użycia nagłówka funkcji) i poprawnego kodu C++:

<pre>// kod błędny: void fun() // definicja {...} // int main() { fun(); }</pre>	<pre>// kod poprawny: void fun2(int a); // nagłówek void fun() {...} int main() { }</pre>
--	---

¹ Ale nie jest to, oczywiście, zabronione.

```
fun2(5): //kompilator nie zna 'fun2'!
}
void fun2(int a) //definicja
{...}
```

```
fun():
fun2(5):
}
void fun2(int a) //definicja
{...}
```

Operacje arytmetyczne

Niewielkie różnice dotyczą pewnych operatorów, które w Pascalu nazywają się nieco inaczej niż w C++. To co może nas uderzyć przy pierwszym spojrzeniu na program napisany w C++, to duża liczba skrótów w zapisie operacji arytmetycznych. Wywołują one wrażenie małej czytelności, jednak po zaznajomieniu się ze składnią języka ten efekt szybko mija. Mam tu przede wszystkim na myśli operatory ++, -- oraz całą rodzinę wyrażeń typu:

zmienna OPERATOR = wyrażenie;

Należy podkreślić, iż stosowanie tych form nie jest obowiązkowe, a mimo to wskazane — kod wynikowy programu będzie dzięki temu nieco efektywniejszy.

```
const pi=3.14;
program pr2;
var a,b,c:integer:{globalne}
begin
  a:=1;
  b:=1;
  a:=a+1; {inkrementacja}
  b:=b-2
end.
```

```
const float pi=3.14;
// lub double dla większej precyzji
int a,b,c;
int main()
{
  a:=1;
  b:=1;
  a++; //inkrementacja
  b-=2; //ŚREDNIK!
}
```

- ◆ Miejsce deklarowania zmiennych w C++ jest dowolne. Można to uczynić *przed*, *za* i *w ciele* niektórych instrukcji.
- ◆ Przypisanie wartości zmiennej odbywa się za pomocą =, a nie :=.
- ◆ Znanym z Pascala div i mod odpowiadają w C++ odpowiednio / i %.

Zwróćmy uwagę na często używane w C++ operatory zwiększania o 1 i zmniejszania o 1 (++ , --). Zastosowane w wyrażeniu są uwzględniane w pierwszej kolejności, jeśli są użyte *przedrostkowo*, natomiast w przypadku użycia *przyrostkowego* priorytet ma wyrażenie.

Przykład:

```
int a = 2;
int b = 5;
int n = a + b++; // n = 7 (priorytet ma dodawanie)
cout << n << endl; // wypisuje 7
cout << b << endl; // wypisuje 6
cout << a + ++b << endl; // wypisuje 9 (priorytet ma inkrementacja)
```

- ◆ Zapis zmienna *op* = wyrażenie jest równoważne klasycznemu zapisowi:
zmienna = zmienna *op* wyrażenie,
gdzie *op* oznacza pewien operator dwuargumentowy.

Operacje logiczne

Podobnie jak arytmetyczne, operacje logiczne także mają swoje osobliwości. Na szczęście nie jest ich aż tak wiele. Programiści pascaliowi powinni zwrócić szczególną uwagę na różnicę pomiędzy = w Pascalu, a == w C++. Niestety kompilator nie wykaże błędu, jeśli w C++ spróbujemy skompilować instrukcję: if (a=1) a=a-3 zamiast if (a==1) a=a-3.

Przykład poprawnych instrukcji logicznych:

```
Program pr3:
var a:boolean;
begin
  a:=true;
  if a=true then
    writeln('true')
  else
    writeln('false')
  end.
```

```
int main()
{
  bool a;
  a=(2>3);
  if (a==true)
    cout << "Prawda!\n";
  else
    cout << "Fałsz!\n";
}
```

W C++ słowa: `bool`, `true` i `false` są słowami kluczowymi języka.

Zwróć uwagę na rolę średnika w C++, który oznacza *koniec* danej instrukcji. Z tego powodu nawet instrukcja znajdująca się przed `else` musi być nim zakończona!

Niektóre operatory logiczne używane w porównaniach są odmienne w obu językach. Przedstawia je tabela A.1.

Tabela A.1. Porównanie operatorów Pascalu i C++

Pascal	C++
=	==
not	!
<>	!=
OR	
AND	&&

Wskaźniki i zmienne dynamiczne

C++ umożliwia stosowanie różnych paradygmatów programowania o wysokim poziomie abstrakcji (jest to przeciwieństwo tzw. języka strukturalny) z możliwościami zbliżającymi go do języka assemblera. Umiejętne wykorzystanie zarówno jednych, jak i drugich umożliwia łatwe programowanie efektywnych aplikacji. Zmienne dynamiczne, adresy i wskaźniki są kluczem do dobrego poznania C++ i trzeba je dobrze opanować. Poniższy przykład ukazuje sposób tworzenia zmiennych dynamicznych i operowania nimi.

```
program pr4:
type example=^real;
var p:example;
begin
  new(p);
  p^:=3.13;
  dispose(p)
end.
```

```
int main()
{
  float *p, q=3.14;
  p=new float;
  *p=q;
  delete p;
}
```

- ♦ W C++ operacje wskaźnikowe (na adresach) nie są ograniczone do zmiennych dynamicznych.
- ♦ Jeśli chcemy wyluskać ze zmiennej wskaźnikowej *wartość*, na którą wskazuje, poprzedźmy ją symbolem gwiazdki (przykładowo `p` zawiera adres `*p` — wartość, oczywiście pod warunkiem że zmienna `p` została wcześniej zainicjowana).
- ♦ Operator `&` wyluskuje ze zmiennej jej adres i stanowi, jak widać, przeciwieństwo operatora `*` (np. `int *p=&m` spowoduje wskazywanie `p` na komórkę pamięci zawierającą zmienną `m`).
- ♦ Adres dowolnej zmiennej w C++ może być z niej pobrany poprzez poprzedzenie jej nazwy operatorem `&`.

Przykład:

```
int k=12;
int *wsk=&k;
cout << *wsk << endl; //program wypisze 12
```

- ◆ W C++ nie ma pozaskładniowych ograniczeń co do operacji na adresach, zmiennych wskaźnikowych, dynamicznych przydzielach pamięci, etc.

Referencje

Osobom słabiej znającym C++ polecam zapoznanie się z pojęciem *referencji*, mechanizmu nie-używanego w tej książce, jednak często spotykanego w praktyce do przekazywania obiektów przez adres, np. do funkcji.

Cechą tego mechanizmu jest przekazywanie do funkcji *adresu* zmiennej, a nie jej kopii lokalnej — wszelkie operacje wykonywane na przekazanej zmiennej będą modyfikowały jej oryginał, a nie kopię lokalną.

Popatrzmy na przykład wywołania funkcji przez referencję:

```
void fun(int& x)
{
    x=5;
}

int main()
{
    int m=6;
    fun(m);
    cout << m << endl;
}
```

Ten nieco „wydumany” przykład pokazuje, jak naturalnie używa się zmiennych przekazywanych przez referencję — jedyna różnica składniowa występuje w nagłówku funkcji (znak &). Zmienna *m* zostanie zmodyfikowana w wyniku wywołania w funkcji *fun* i program wyświetli 5. Referencje są o tyle wygodne, że nie zmuszają do używania operatora wyluskiwania (*) wywołującego się z języka C.

Typy złożone

W języku C++ występuje komplet typów prostych i złożonych, dobrze znanych z języków strukturalnych. Należą do nich między innymi tablice i rekordy. W porównaniu z Pascalem C++ oferuje tu pozornie mniejsze możliwości. Podstawowe ograniczenie tablic dotyczy zakresu indeksów: zawsze zaczynają się one od zera. Nie jest możliwe również deklarowanie rekordów „z wariantami”. Te niedogodności są oczywiście do obejścia, ale nie w sposób bezpośredni.

Tablice

Indeksy w tablicach deklarowanych w C++ startują zawsze od zera. Tak więc deklaracja tablicy *t* o rozmiarze 4 oznacza w istocie 4 zmienne: *t*[0], *t*[1], *t*[2] i *t*[3]. Aby uzyskać zgodność indeksów w programach napisanych w Pascalu i w C++, konieczne jest zastosowanie właściwej translacji tychże!

```
Program pr5;
type tab=array[3..5] of integer;
var t:tab;
```

```
typedef int tab[3];
tab t;
int main()
```

```

begin
    t[3]:=11;
    t[4]:=t[3]+1;
end.

{
    t[0]=11;
    *(t+1)=t[0]+1;
}

```

- ♦ Język C++ w zasadzie nie zapewnia kontroli przekroczenia granic tablic podczas dostępu do nich za pomocą indeksowania, ufając niejako programiście. Radą na to jest zastosowanie mechanizmów obiektowych, ale w wersji pierwotnej trzeba po prostu uważać, aby nie znaleźć się w malinach².
- ♦ Nazwa tablicy w C++ jest jednocześnie wskaźnikiem do niej. Przykładowo: `t` wskazuje na pierwszy element tablicy, a `(t+3)` na czwarty. Notacja `*(t+1)` jest równoważna `t[1]`.
- ♦ Deklaracja `int *x` jest równoważna `int x[]`.

Rekordy

Prosty przykład pokazuje elementarne operacje na rekordach:

```

program pr6;
type Komorka=
    record
        znak:char;
        a,b,d:integer;
    end;
var x:Komorka;
begin
    x.znak:= 'a';
    x.a:=1;
end.

struct Komorka
{
    char znak;
    int a,b,d;
};
Komorka x;
int main()
{
    x.znak= 'a';
    x.a=1;
}

```

- ♦ Rekordy w C++ są zwane *strukturami*, dostęp do nich jest podobny, jak w przypadku Pascala (notacja z kropką).
- ♦ Nie można wprost zadeklarować rekordu z wariantami.
- ♦ Jest możliwe, podobnie jak w Pascalu, włożenie tablicy do rekordu i odwrotnie.
- ♦ Pole nazwa_pola wskaźnika lub rekordu dynamicznego, wskazywanego przez zmienną `y`, nie jest dostępne poprzez `y.nazwa_pola`, lecz przez konstrukcję: `x->nazwa_pola`. Przykład:

```

Komorka x, *y; //zmienna typu Komorka oraz wskaźnik do niej.
x.znak='a';
y=&x; //pobiera adres zmiennej x
y->a=1; //notacja z kropką byłaby błędna!

```

Instrukcja switch

Instrukcja `switch` w C++ różni się w kilku zdradzieckich szczegółach od swojej odpowiedniczki w Pascalu — proszę zatem uważnie przeanalizować podany przykład!

Najważniejsza do zapamiętania informacja jest związana ze słowem kluczowym `break` (*przerwij*). Ominięcie go spowodowałoby wykonanie instrukcji znajdujących się dalej, aż do napotkania jakiegoś innego `break` lub końca instrukcji `switch`.

² Tutaj mogłoby to oznaczać „złe adresy”.

```

program pr7;
var w:integer;
begin
  w:=2;
  case w of
    1: writeln('1');
    2: writeln('2');
    otherwise:
      writeln('?');
  end
end.

int w;
int main()
{
  w=2;
  switch(w)
  {
    case 1: cout<< "1\n"; break;
    case 2: cout<< "2\n"; break;
    default:
      cout<<"?\n"; break;
  }
}

```

- ◆ W C++ break pełni rolę separatora przypadków.

Iteracje

Instrukcje iteracyjne są podobne w obu językach:

```

program pr8;
var i,j:integer;
begin
  j:=1;
  for i:=1 to 5 do
    begin
      writeln(i*j);
      j:=j+1;
    end;
  i:=1;
  j:=10;
  while j>i do
    begin
      i:=i+1;
      writeln(i)
    end
  end.

int i,j;
int main()
{
  j=1;
  for(i=1;i<=5;i++)
  {
    cout << i*j << endl;
    j++;
  }
  i=1;
  j=10;
  while (j>i++)
    cout << i << endl;
}

```

- ◆ endl oznacza znak powrotu do nowej linii.
- ◆ Niewymieniona tu instrukcja do{... }while(v) jest wykonywana w C++ dopóki, dopóty wyrażenie v jest różne od zera³.
- ◆ Elementy instrukcji for(e1; e2; e3) oznaczają odpowiednio:
 - ◆ e1: inicjację pętli,
 - ◆ e2: warunek wykonania pętli,
 - ◆ e3: modyfikator zmiennej sterującej (może nim być funkcja, grupa instrukcji oddzielonych przecinkiem — wtedy są one wykonywane od lewej do prawej).

Przykład:

```
for(int i=6; i<100; Insert(tab[i++]), Pisz(i));
```

(Pisz i Insert są funkcjami, tab zaś pewną tablicą.)

³ Porównaj np. z repeat... until.

Struktury rekurencyjne

Przykład następny pokazuje sposób deklarowania rekurencyjnych (odwołujących się do siebie samych) struktur danych.

```

Program pr11:
type wsk=^element;
element=record
    wartosc : integer;
    nastepny:wsk
end;

var p:wsk;
begin
    new(p);
    read(p^.wartosc);
    p^.wartosc=nil
end.

typedef struct x
{
    int wartosc;
    struct x* nastepny;
} ELEMENT;

int main()
{
    ELEMENT *p;
    p=new ELEMENT;
    cin >> p->wartosc;
    p->nastepny=NULL;
}

```

♦ Odpowiednikiem nil w C++ jest NULL.

Parametry programu main()

Czasami istnieje potrzeba podania, przed wykonaniem programu w C++, parametrów wejściowych, np. opcji wywołania, nazw plików z danymi wejściowymi itp. Parametry wywołania są ciągami znaków i można dość łatwo się do nich dostać, choć składnia, pochodząca jeszcze z języka C, jest nieco dziwaczna.

Następujący program wypisuje parametry podane przy wywołaniu programu (skompilowanego kodu wykonywalnego) w linii poleceń:

```

#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    for (int i=0; i< argc; i++)
        cout << "Parametr nr " << i << " : "<< argv[i] << endl;
}

```

Parametr argc jest licznikiem parametrów, wartość argv[0] jest nazwą samego programu wykonywalnego, pozostałe (jeśli istnieją) są już zwykłymi parametrami wywołania.

Operacje na plikach w C++

Operacje na plikach w C++ w porównaniu z językiem C to przysłowiowa bułka z masłem. Popatrzmy na przykład programu, który odczytuje pewien plik wejściowy, kopiuje go (wiersz po wierszu) do pliku wyjściowego oraz wypisuje, znak po znaku, plik wejściowy w wersji znakowej oraz kodami (dziesiętnymi i szesnastkowo). Nasz program operuje strumieniami, występuje tu bardzo podobna koncepcja do innych, znanych nam strumieni (np. cout).



Listing

pliki.cpp

```

#include <string>
#include <iostream>
#include <fstream>
using namespace std;

```

```

int main()
{
    ifstream plik_WEJ ("input.txt"); //plik wejściowy
    ifstream plik_BIN ("input.txt"); //ten sam plik wejściowy
    ofstream plik_WYJ ("output.txt"); //plik wyjściowy

    string s;

    while (getline(plik_WEJ,s)) //kopiujemy, linia po linii, plik
        plik_WYJ << s << endl; //input.txt do output.txt

    char c; //wypisujemy znak po znaku plik wejściowy:
    while (plik_BIN.read(&c,1) )
        cout << "Znak: "<< c << ", dec:"<< dec << (int)c << ", hex: "
            << hex << (int)c << endl;
    cout << endl;
}

```

Oto przykładowy wynik programu: (plik wejściowy *input.txt* zawierał dwa wiersze z napisami: 123 i abA.

```

Znak: 1, dec:49, hex: 31
Znak: 2, dec:50, hex: 32
Znak: 3, dec:51, hex: 33
Znak:
, dec:10, hex: a
Znak: a, dec:97, hex: 61
Znak: b, dec:98, hex: 62
Znak: A, dec:65, hex: 41
itp.

```

Programowanie obiektowe w C++

Cała siła i piękno języka C++ zawiera się nie w cechach odziedziczonych od swojego przodka⁴, lecz w nowych możliwościach udostępnionych przez wprowadzenie elementów obiektowych. W zasadzie po raz pierwszy w historii informatyki mamy do czynienia z przypadkiem aż tak dużego zainteresowania jakimś językiem programowania, jak to miało miejsce z C++. Niegdyśjsza moda staje się już powoli wymogiem chwili: jest to narzędzie tak efektywne, iż niekorzystanie z niego naraża programistę na stanie w miejscu, w momencie gdy świat coraz szybciej podąża do przodu!

Tylko dla formalności przypomnę jeszcze ostrzeżenie zawarte we wstępie: cały ten rozdział służy wyłącznie nauczaniu programisty pascalogowego *czytania* i *rozumienia* listingów napisanych w C++. Ograniczona objętość książki, w konfrontacji z rozpiętością tematyki, nie pozwala na omówienie wszystkiego. Mimo to cytowane tu przykłady zostały wybrane ze względu na ich dużą reprezentatywność. Osoby głębiej zainteresowane programowaniem obiektowym w C++ mogą skorzystać np. z [Poh89], [Eck02] w celu poszerzenia swojej wiedzy.

Terminologia

Typowe pojęcia związane z programowaniem obiektowym poglądowo zgrupowano na rysunku A.1.

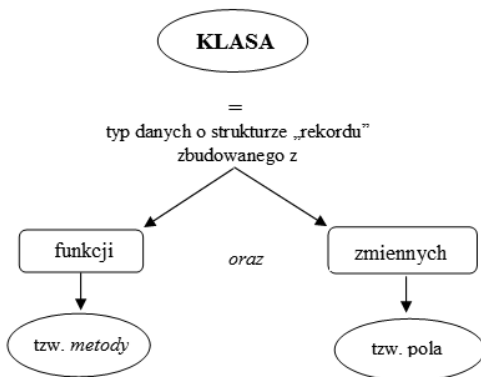
- ◆ Zmienna tego nowego typu danych zwana jest *obiekt*em.
- ◆ *Metody* są to zwykłe funkcje lub procedury operujące polami, stanowiące jednak własność klasy⁵.

⁴ Którym jest naturalnie język C.

⁵ Tzn. mogą z nich korzystać obiekty danej klasy — inne, zewnętrzne funkcje programu już nie!

Rysunek A.1.

Terminologia
w programowaniu
obiektowym



Istnieją dwie metody specjalne:

- ♦ *Konstruktor*, który tworzy i inicjalizuje obiekt (np. przydziela niezbędną pamięć, inicjuje w żądany sposób pewne pola, etc.). W deklaracji klasy można bardzo łatwo rozpoznać tę metodę po nazwie — jest ona identyczna z nazwą klasy, ponadto konstruktor ani nie zwraca żadnej wartości, ani nawet nie jest typu `void`.
- ♦ *Destruktor*, który niszczy obiekt (zwalnia zajęta przezeń pamięć). Podobnie jak i konstruktor, posiada on specjalną nazwę: identyczną z nazwą klasy, ale poprzedzoną znakiem tyldy (~);
- ♦ Każda metoda ma dostęp do pól obiektu, na rzecz którego została aktywowana, poprzez ich nazwy. Inny sposób dostępu jest związany ze wskaźnikiem o nazwie `this` (słowo kluczowe C++): wskazuje on na własny obiekt. Tak więc dostęp do atrybutu `x` może się odbyć albo poprzez `.x`, albo przez `this->x`. Typowo jednak wskaźnik `this` służy w sytuacjach, w których metoda, po uprzednim zmodyfikowaniu obiektu, chce go zwrócić jako wynik (np.: `return *this;`).

Obiekty na przykładzie

Klasa, jako specjalny typ danych, przypomina w swojej konstrukcji rekord, który został wyposażony w możliwość wywoływania funkcji. Definicja klasy może być podzielona na kilka sekcji charakteryzujących się różnym stopniem dostępności dla pozostałych części programu. Najbardziej typowe jest używanie dwóch rodzajów sekcji: *prywatnej* i *publicznej*. W części prywatnej na ogół umieszcza się informacje dotyczące organizacji danych (np. deklaracje typów i zmiennych), a w części publicznej wymienia dozwolone operacje, które można na nich wykonywać. Operacje te mają, oczywiście, postać funkcji, czyli — używając już właściwej terminologii — metod przypisanych klasie.

Spójrzmy na sposób deklaracji klasy, która w sposób dość uproszczony obsługuje tzw. *liczby zespolone*:



complex.h

```

class Complex
{
public:
    Complex(double x, double y) // początek sekcji publicznej // konstruktor klasy
    {
        Re=x;
        Im=y;
    }
}
  
```

```

void wypisz(): // nagłówek funkcji, która wypisuje liczbę urojoną
double Czesc_Rzecz() // zwraca część rzeczywistą
{
    return Re;
}
double Czesc_Uroj () // zwraca część urojoną
{
    return Im;
}
// nagłówek funkcji, którą przeddefiniowuje operator + (plus), aby
// umożliwić dodawanie liczb zespolonych:
friend Complex& operator +(Complex,Complex);
// nagłówek funkcji, którą przeddefiniowuje operator << aby umożliwić wypisywanie liczb zespolonych:
friend ostream& operator << (ostream&,Complex);
private: // początek sekcji prywatnej
    double Re,Im; // reprezentacja jako Re+j*Im
}; // koniec deklaracji (i częściowej definicji)
// klasy Complex

```

Konstrukcja klasy `Complex` informuje o naszych intencjach:

- ◆ Wiemy, że liczby zespolone są wewnętrznie widziane jako część rzeczywista i część urojona. Ponieważ sposób budowy klasy jest jej prywatną sprawą, informację o tym umieszczamy w sekcji prywatnej, która redukuje się w naszym przypadku do deklaracji zmiennych `Re` i `Im`.
- ◆ Z punktu widzenia obserwatora zewnętrznego (czyli po prostu użytkownika klasy) liczba zespolona jest to obiekt, na którym można wykonywać operację dodawania⁶ (mnożenia, dzielenia, etc.) oraz wypisywać ją w pewnej określonej postaci⁷.
- ◆ W celu dodawania liczb zespolonych przeddefiniujemy znaczenie standardowego operatora `+`, podobnie uczynimy w przypadku wypisywania — tym razem z operatorem `<<`.

Konstruktor klasy oraz dwie proste metody `Czesc_Rzecz` i `Czesc_Uroj` są zdefiniowane już wewnątrz deklaracji klasy ograniczonej nawiasami klamrowymi `{ }`. Decyzja o miejscu definicji jest najczęściej podyktowana długością kodu: jeśli metoda ma pokaźną objętość⁸, to zwykle przemieszcza się ją na zewnątrz, w środku pozostawiając tylko nagłówek.

Deklaracja przykładowego obiektu `20+10*j` ma w programie postać:

- ◆ **przypadek 1:** (niejawne tworzenie obiektu poprzez jego deklarację):

```
Complex NazwaObiektu(20, 10);
```

- ◆ **przypadek 2 :** (jawne tworzenie obiektu poprzez `new`):

```
Complex *NazwaObiektu_Ptr = new Complex(20,10);
```

Wywoływanie metod odbywa się za pomocą standardowej notacji „z kropką”:

```
NazwaObiektu.NazwaMetody(parametry); // przypadek 1
```

lub

```
NazwaObiektu_Ptr->NazwaMetody(parametry); //przypadek 2
```

Wiedząc już, jak to wszystko powinno działać, popatrzmy, jak zrealizować brakujące metody.

⁶ Przykład ogranicza się tylko do dodawania — pozostałe operacje arytmetyczne Czytelnik może z łatwością dopisać samodzielnie.

⁷ Reprezentacja za pomocą modułu i fazy zostaje pozostawiona do realizacji Czytelnikowi jako proste ćwiczenie programistyczne.

⁸ Powszechnie zalecaną regułą jest nieprzekraczanie jednej strony przy konstrukcji procedury — tak aby całość mogła zostać objęta wzrokiem bez konieczności gorączkowego przerzucania kartek.

Funkcja wypisz jest tak trywialna, iż równie dobrze mogłaby być zdefiniowana wprost w ciele klasy. Ponieważ jest to metoda klasy `Complex`, musimy o tym poinformować kompilator przez poprzedzenie jej nazwy nazwą klasy zakończoną operatorem `::` (wymóg składniowy). Jako metoda klasy procedura ta ma dostęp do prywatnych pól obiektu, na rzecz którego została aktywowana. Gdyby jednym z parametrów tej metody był inny obiekt klasy `Complex` (np. `Complex x`), to dostęp do jego pól odbywałby się za pomocą notacji z kropką. Przykład: `x.Re`.



complex.cpp

```
void Complex::wypisz()
{
    cout << Re << "+j*" << Im << endl;
}
```

Język C++ umożliwia łatwe *przedefiniowanie znaczenia operatorów standardowych*, tak aby operacje na obiektach uczynić możliwie najprostszymi. Ponieważ liczby zespolone nieco inaczej dodaje się niż te zwykłe, celowe będzie ukrycie sposobu dodawania w funkcji, a w świecie zewnętrznym pozostawienie do tego celu operatora `+`. Najwygodniejszym sposobem przeddefiniowania operatora dwuargumentowego jest użycie do tego celu tzw. *funkcji zaprzyjaźnionej*: jest to specjalna funkcja, która — nie będąc metodą⁹ pewnej określonej klasy — może operować obiektami należącymi do niej. Dotyczy to również dostępu do pól prywatnych!

Nasza funkcja zaprzyjaźniona ma następujące działanie: dwa obiekty `x` i `y` są przekazywane jako parametry. Odczytując wartości ich pól `Re` i `Im`, możliwe jest skonstruowanie nowego obiektu klasy `Complex` wg prostego wzoru: $(a+j\cdot b)+(c+j\cdot d)=(a+c)+(b+d)\cdot j$. Po utworzeniu nowy obiekt jest zwracany przez referencję — czyli jako w pełni adresowalny obiekt. Może on być przypisany innemu obiektowi, na rzecz którego może być aktywowana jakaś metoda klasy `Complex`, etc. Prawidłowe będą zatem instrukcje:

```
Complex x(1,2),y(2,3),c;    // deklaracje obiektów
c=x+y;                      // c = (1+2)+j(2+3)
```

Popatrzmy na listing funkcji `+`:

```
Complex& operator +(Complex x,Complex y)
{
    double tmp_Re=x.Czesc_Rzecz()+y.Czesc_Rzecz();
    double tmp_Im=x.Czesc_Uroj()+y.Czesc_Uroj();
    Complex *NowyObiekt=new Complex(tmp_Re,tmp_Im);
    return (*NowyObiekt);
}
```

Warto zwrócić uwagę na fakt, iż obiekt `NowyObiekt` jest tworzony w sposób *jawny* za pomocą `new`. Tego typu postępowanie zapewnia nam, że zwrócona referencja będzie się odnosiła do obiektu trwałego (zwykła instrukcja `Complex NowyObiekt` użyta wewnątrz bloku stworzyłaby obiekt tymczasowy, który zniknąłby po wykonaniu instrukcji zawartych we wspomnianym bloku).

Podobnie jak w przypadku operatora `+` celowe mogłoby być przeddefiniowanie operatora `<<`, który wysyła sformatowane dane do strumienia wyjściowego. W C++ służy do tego celu klasa o nazwie `ostream`. Bez wnikania w szczegóły¹⁰ proponuję zapamiętać zastosowaną poniżej sztuczkę:

```
ostream& operator << (ostream &str,Complex x)
{
    str << x.Czesc_Rzecz()<< "+j*" << x.Czesc_Uroj();
    return str;
}
```

⁹ W konsekwencji nie mogą być wywoływane za pomocą notacji „z kropką”!

¹⁰ Nie miejsce tu bowiem na omawianie dość złożonej hierarchii bibliotek klas dostarczanych z dobrymi kompilatorami C++. Początkującego fana C++ taki opis mógłby dość skutecznie zanudzić.

Spójrzmy wreszcie na program przykładowy, który tworzy obiekty i manipuluje nimi:

```
#include "complex.h"
int main()
{
    Complex c1(1,2),c2(3,4);
    cout << "c1=";
    c1.wypisz();
    cout << "c2=";
    c2.wypisz();
    cout << "c1+c2="<<(c1+c2) << endl;
    Complex *c_ptr=new Complex(1,7);
    cout << "a)c_ptr wskazuje na obiekt";
    c_ptr->wypisz();
    cout << "b) c_ptr wskazuje na obiekt"<<*c_ptr<< endl;
}
```

Dla formalności prezentuję rezultaty wykonania programu:

```
c1=1+j*2
c2=3+j*4
c1+c2=4+j*6
c_ptr wskazuje na obiekt 1+j*7
c_ptr wskazuje na obiekt 1+j*7
```

Składowe statyczne klas

Każdy nowo utworzony obiekt posiada pewne unikatowe cechy (wartości swoich atrybutów). Od czasu do czasu zachodzi jednak potrzeba dysponowania czymś w rodzaju zmiennej globalnej w obrębie danej klasy: służą do tego tzw. *poła statyczne*.

Poprzedzenie w definicji klasy C atrybutu x słowem *static* spowoduje utworzenie właśnie tego typu zmiennej. Inicjacja takiego pola może nastąpić nawet przed utworzeniem jakiegokolwiek obiektu klasy C! W tym celu piszemy po prostu

```
C::x=jakaś_wartość;
```

Zbliżone ideowo jest pojęcie *metody statycznej*: może być ona wywołana jeszcze przed utworzeniem jakiegokolwiek obiektu. Oczwistym ograniczeniem metod statycznych jest brak dostępu do pól *niestatycznych danej klasy*, ponadto wskaźnik *this* nie ma żadnego sensu. W przypadku metody statycznej, jeśli chcemy jej umożliwić dostęp do pól niestatycznych pewnego obiektu, trzeba go jej przekazać jako... parametr!

Metody stałe klas

Metoda danej klasy może zostać przez programistę określona mianem *stalej* (np. `void fun() const;`). Nazwa ta jest dość nieszczęśliwie wybrana, chodzi w istocie o metodę deklarującą, że nigdy nie zmodyfikuje pól obiektu, na rzecz którego została aktywowana.

Dziedziczenie własności

Załóżmy, że dysponujemy starannie opracowanymi klasami A i B. Dostaliśmy je w postaci *skompilewanych bibliotek*, tzn. oprócz kodu wykonywalnego mamy do dyspozycji tylko szczegółowo skomentowane pliki nagłówkowe, które informują nas o sposobach użycia metod i o dostępnych atrybutach.

Niestety twórca klas A i B dokonał kilku wyborów, które nas niespecjalnie satysfakcjonują, i zaczęło nam się wydawać, że my zrobilibyśmy to nieco lepiej.

Czy musimy wobec tego napisać własne klasy A i B, a dostępne biblioteki wyrzucić na śmietnik? Powinno być oczywiste dla każdego, że nie zadawałbym tego pytania, gdyby odpowiedź nie brzmiała: *NIE*. Język C++ pozwala na bardzo łatwą „ponowną utylizację” kodu już napisanego (a nawet skompilowanego), przy jednoczesnym umożliwieniu wprowadzenia niezbędnych zmian. Weźmy dla przykładu deklaracje dwóch klas A i B, zamieszczone na listingu poniżej:



dziedzic.h

```
class C1
{
protected:
    int x;
public:
    C1(int n) //konstruktor
    {
        x = n;
    }
    void pisz()
    {
        cout<<"**Stara wersja ";
        cout <<"metody `pisz:x=" << x << endl;
    }
};

class C2
{
private:
    int y;
public:
    C2(int n) //konstruktor
    {
        y = n;
    }

    int ret_y()
    {
        return y;
    }
};
```

Słowo kluczowe *protected* (*chroniony*) oznacza, że mimo prywatnego dla użytkownika klasy charakteru informacji znajdujących się w tej sekcji zostaną one przekazane ewentualnej klasie pochodnej (zaraz zobaczymy, co to oznacza). Znaczy to, że klasa dziedzicząca będzie ich mogła używać zwyczajnie poprzez nazwę, ale już użytkownik nie będzie miał do nich dostępu poprzez np. notację „z kropką”. Jeszcze większymi ograniczeniami charakteryzują się pola *private*: klasa dziedzicząca traci możliwość używania ich w swoich metodach za pomocą nazwy. Ten brak dostępu można, oczywiście, sprytnie ominąć, definiując wyspecjalizowane metody służące do *kontrolowanego* dostępu do pól klasy.

Dołożenie tego typu ochrony danych znakomicie izoluje tzw. *interfejs użytkownika* od bezpośredniego dostępu do danych, ale to już jest temat na osobny rozdział!

Przeanalizujmy wreszcie konkretny przykład programu. Nowa klasa C dziedziczy własności po klasach A i B oraz dokłada nieco swoich własnych elementów:



dziedzic.cpp

```
#include "dziedzic.h"
class C3:public C1,C2
{
    int z; //pole prywatne
```

```

public:
    C3(int n) : C1(n+1), C2(n-1)    // nowy
    {                               // konstruktor
        z=2*n;
    }
    pisz_wszystko()
    {
        cout << "Wszystkie pola:\n";
        cout << "\t x=" << x << endl;
        cout << "\t y=" << ret_y() << endl;
        cout << "\t z=" << z << endl;
    }
};
int main()
{
    C3 ob(10);
    ob.pisz_wszystko();
}

// wynik:
// Wszystkie pola:
// x = 11
// y = 9
// z = 20

```

Konstruktor klasy C3, oprócz tego, że inicjalizuje własną zmienną z, wywołuje jeszcze konstruktory klas C1 i C2 z takimi parametrami, jakie mu aktualnie odpowiadają. Kolejność wywoływania konstruktorów jest logiczna: najpierw konstruktory klas bazowych (w kolejności narzuconej przez ich pozycję na liście znajdującej się po dwukropku), a na sam koniec konstruktor klasy C3. W naszym przypadku parametry $n+1$ i $n-1$ zostały wzięte „z kapelusza”.

Kod zaprezentowany na powyższych listingach jest poglądowo wyjaśniony na rysunku A.2.

W C++ kilka różnych pod względem zawartości funkcji może nosić taką samą nazwę — taka sytuacja nosi nazwę *przeciążenia*, „ta właściwa” funkcja jest rozróżniana poprzez typy swoich parametrów wejściowych. Przykładowo, jeśli w pliku z programem są zdefiniowane dwie procedury: `void p(char* s)` i `void p(int k)`, to wówczas wywołanie `p(12)` niechybnie będzie dotyczyć tej drugiej wersji.

Mechanizm przeciążania może być zastosowany bardzo skutecznie w powiązaniu z mechanizmami dziedziczenia. Załóżmy, że nie podoba nam się funkcja `pisz`, dostępna w klasie C3 dzięki temu, że w procesie dziedziczenia przeszła ona z klasy C1 do C3. Z drugiej zaś strony podoba nam się nazwa `pisz` w tym sensie, że chcielibyśmy jej używać na obiektach klasy C3, ale do innego celu. Uzupełniamy wówczas klasę C3 o następującą definicję¹¹:

```

void C3::pisz()
{
    cout << "nowa wersja metody 'pisz'\n";
}
// wynik użycia ob.C1::pisz(); w main:
// ** stara wersja metody 'pisz': x = 11 **
// wynik użycia ob.pisz(); w main:
// ** nowa wersja metody 'pisz': z = 20 **

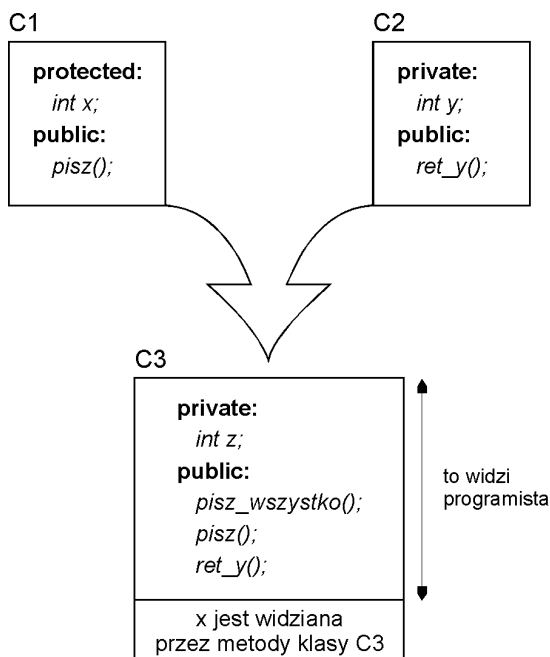
```

Teraz instrukcja `ob.pisz()` wywoła nową metodę `pisz` (z klasy C3), gdybyśmy zaś koniecznie chcieli użyć starej wersji, to należy jawnie tego zażądać poprzez `ob.C1::pisz()`.

¹¹ Ponadto należy dodać linię `void pisz();` do sekcji publicznej klasy C3.

Rysunek A.2.

Przykład dziedziczenia
własności w C++



Nasz przykład zakłada kilka celowych niedomówień. Wynika to z tego, że problematyka dziedziczenia własności w C++ zawiera wiele niuansów, które mogłyby być nużące dla nieprzygotowanego odbiorcy. Mimo to zaprezentowana powyżej wiedza jest już wystarczająca do tworzenia dość skomplikowanych aplikacji zorientowanych obiektowo. Inne mechanizmy, takie jak np. bardzo ważne funkcje *wirtualne* i tzw. *klasy abstrakcyjne*, trzeba już pozostawić wyspecjalizowanym publikacjom — zachęcam Czytelnika do lektury.

Kod warunkowy w C++

W kodach źródłowych (głównie wersja *ftp*), można napotkać na dyrektywy zlecające warunkowe wykonywanie kodu programu.

Jeśli napotkasz fragment kodu o strukturze podobnej do:

```

#define TEST // (*)
// tutaj jakieś deklaracje i instrukcje (1)
#ifdef TEST
// tutaj jakieś inne deklaracje i instrukcje (2)
#endif
  
```

to kompilator wykona instrukcje oznaczone (1). Gdyby teraz usunąć lub wstawić do komentarza linijkę (*), to wykonałby się kod zawarty w drugiej odnodze (2).

Dodatek B

Systemy obliczeniowe w pigułce

W niniejszym dodatku przedstawię kilka podstawowych informacji dotyczących systemów kodowania — dwójkowego i szesnastkowego i arytmetyki binarnej. Z moich obserwacji wynika bowiem, że nie każdy programista potrafi się przyznać, że ma z tymi pojęciami kłopoty, i ten właśnie suplement pozwoli mu wyrównać tę lukę w wykształceniu.

Kilka definicji

W ramach wstępu wprowadzę pojęcie *systemu pozycyjnego*, w którym liczby zapisuje się cyframi C_1, C_2, \dots, C_n z pewnego niewielkiego zbioru, a zapis taki interpretuje się jako sumę iloczynów liczb reprezentowanych przez poszczególne cyfry i potęg liczby naturalnej n , nazywanej podstawą systemu, o wykładnikach równych numerowi pozycji cyfry w ciągu. W branży informatycznej popularne są szczególnie podstawy systemu: 2, 8 i 16 — czyli systemy: dwójkowy, ósemkowy i szesnastkowy.

Powszechnie używane układy cyfrowe, z których składają się komputery, wykonują operacje arytmetyczne i są sterowane z wykorzystaniem zasad tzw. algebry Boole’a, czyli zbioru działań zakładających operowanie tylko dwoma elementami oznaczonymi 0 i 1. Zasady tej algebry można podsumować zbiorem praw przedstawionych w tabeli B.1.

Operacje logiczne występujące w tabeli to: suma (+), iloczyn (·) i dopełnienie (kreska nad elementem).

System dwójkowy

System dwójkowy pozwala na zapis liczb za pomocą dwóch umownych znaków: 0 (zero) i 1 (jedynek). Umowność polega na niewnikaniu w szczegóły, co dla danego komputera (lub sprzętu, np. układu scalonego) oznacza fizycznie każdy z tych stanów — czy jest to zero woltów, czy np. 5 woltów. Z naszego punktu widzenia liczy się tylko fakt, iż ciągle dla wielu zastosowań reprezentacja dwójkowa liczb jest najwygodniejsza, a ponieważ jest ona doskonale wspomagana w językach programowania, np. C++, to tym cenniejsze będzie pełne jej zrozumienie.

Tabela B.1. Prawa i twierdzenia algebry Boole’a

Prawa przemienności $x + y = y + x$ $x \cdot y = y \cdot x$	Prawa łączności $(x + y) + z = x + (y + z)$ $(x \cdot y) \cdot z = x \cdot (y \cdot z)$	Prawa rozdzielności $x + (y \cdot z) = (x + y) \cdot (x + z)$ $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Prawa identyczności $x + 0 = x$ $x \cdot 1 = x$	Prawa dopełnienia $x + \overline{x} = 1$ $x \cdot \overline{x} = 0$	Prawa idempotentności $x + x = x$ $x \cdot x = x$ $x + 1 = 1$ $x \cdot 0 = 0$
Prawa pochłaniania: $(x \cdot y) + x = x$ $(x + y) \cdot x = x$	Prawa de Morgana $\overline{x + y} = \overline{x} \cdot \overline{y}$ $\overline{x \cdot y} = \overline{x} + \overline{y}$	

Popatrzmy, jak w naturalnym dla człowieka systemie dziesiętnym można przedstawić liczbę 1624:

$$1\,624 = 1 \cdot 1\,000 + 6 \cdot 100 + 2 \cdot 10 + 4 = 1 \cdot 10^3 + 6 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0$$

Sumujemy „wagi” (1, 6, 2 i 4) mnożone przez potęgi podstawy systemu obliczeniowego (tutaj: 10) i wychodzi nam dowolna liczba, którą chcemy przedstawić. Albo na odwrót: to właśnie dowolna liczba może zostać rozłożona na podaną wyżej reprezentację.

Wadą systemu dziesiętnego jest kłopot w budowie sprzętu elektronicznego, który operowałby liczbami tego systemu. Nie tylko komplikuje się reprezentacja wewnętrzna (aż dziesięć stanów do zakodowania na poziomie technicznym, czyli np. bramek i tranzystorów), ale jeszcze obliczenia w systemie dziesiętnym są dość pamięciożerne.

System dwójkowy używa do reprezentacji liczb dwóch znaków: 0 i 1, a podstawą systemu jest dwójka.

Popatrzmy, jak łatwo przekształca się liczbę dwójkową na jej reprezentację dziesiętną.

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$$

W rzeczywistych komputerach mamy do czynienia z wewnętrznymi rejestrami, które pozwalają zapamiętywać i operować na grupach bitów. Najbardziej znaną jednostką, w której podaje się tak długość takiej grupy, jest *bajt*, czyli osiem bitów.

Bajt może reprezentować liczby od 00000000 do 11111111, czyli dziesiętnie od 0 do 255.

W przypadku systemu dwójkowego poważnym problemem okazało się reprezentowanie liczb ujemnych. Pierwszy pomysł, aby najbardziej znaczący bit (ten wysunięty najbardziej na lewo) oznaczał znak, np. 1 to liczba ujemna, a 0 — dodatnia: 0011 = 3, 1011 = -3. Z przyczyn technicznych system ten się nie przyjął, gdyż niejednoznaczność związana ze znakiem zera (1000 = +0, 0000 = -0???) rodziła sporo kłopotów. Matematycy przyszli wówczas z pomocą, wymyślając *system uzupełnienia dwójkowego*, w którym liczba ujemna jest uzyskiwana poprzez zanegowanie wszystkich bitów (0 na 1, 1 na 0) i dodanie liczby jeden do wyniku.

Przykład:

1101	+13
0010	negujemy bity
+0001	dodajemy 1
0011	wynik -13.

Zaletą systemu uzupełnienia jest rozwiązanie problemu znaku zera i brak potrzeby posiadania operacji odejmowania — wystarczy negacja bitów i dodawanie jedynki (inkrementacja).

Operacje arytmetyczne na liczbach dwójkowych

Arytmetyka liczb dwójkowych jest podobna do dziesiętnych (stosujesz reguły „słupkowe” poznane w szkole podstawowej!). Popatrz na kilka prostych przykładów zawartych w tabeli B.2.

Tabela B.2. Arytmetyka liczb dwójkowych na przykładzie

	Przykład	Komentarz
Dodawanie	$A = 11011 = 27_{10}$ $B = 11001 = 25_{10}$ $A+B = 110100$	Pamiętaj 0 przeniesieniu 1 dla (1+1)!
Odejmowanie	$A = 11010 = 26_{10}$ $B = 10001 = 17_{10}$ $A-B = 1001$	W przypadku odejmowania (0–1) musimy dokonać „zapożyczenia” 1 na następnej pozycji liczby.
Mnożenie	$A = 101 = 5_{10}$ $B = 010 = 2_{10}$ $A*B = 1010$	Tak jak w systemie dziesiętnym (mnożysz przez kolejne cyfry i przesuwasz się w lewo).
Dzielenie	110 $1100:10 = 12_{10}:6_{10}=2_{10}$ -10 0100 -10 000	Tak jak w systemie dziesiętnym (odejmujesz przesunięty na lewo dzielnik od dzielnej aż do uzyskania 0 lub reszty).

Operacje logiczne na liczbach dwójkowych

Podstawowe operacje logiczne wykonywane na liczbach dwójkowych są przedstawione w tabeli B.3. W programach często używa się kodowania bitowego do oznaczania opcji wywołania funkcji — zazwyczaj niskopoziomowych, np. na plikach.

Tabela B.3. Podstawowe operacje na liczbach dwójkowych

	Zasada obliczania wyniku na dwóch bitach	Przykład (zapisany w konwencji C++)
LUB (ang. <i>OR</i>) — suma logiczna	Jeśli choć jeden z bitów jest jedynką, to wynik jest równy 1.	$1101 \mid 0001 = 1101$
I (ang. <i>AND</i>) — iloczyn logiczny	Jeśli choć jeden z bitów jest zerem, to wynik jest równy 0.	$1001 \& 1100 = 1000$
Różnica symetryczna ¹ (ang. <i>XOR</i>)	Jeśli oba bity są równe, to wynik jest zerem.	$1100 \wedge 1001 = 0101$
Negacja	1 staje się 0, 0 staje się 1	$\sim 1101 = 0010$
Przesunięcie w lewo o <i>n</i> bitów	-	<code>zmienna << n</code>
Przesunięcie w prawo o <i>n</i> bitów	-	<code>zmienna >> n</code>

¹ Inna popularna nazwa to „suma modulo 2”.

```
#define OPCJA_1      1      // 0001
#define OPCJA_2      2      // 0010
#define OPCJA_3      4      // 0100
#define OPCJA_4      8      // 1000
...
int opcje = 0;
opcje = OPCJA_1 | OPCJA_4; // „włączenie” opcji 1 i 4, wynik: 1001
```

Aby zaprzyjaźnić Czytelnika z operacjami bitowymi, proponuję analizę następującego programu, który pokazuje kilka typowych operacji bitowych i przy okazji zaznajamia Czytelnika z prostym sposobem wyświetlania liczby w postaci dwójkowej.



bit_operations.cpp

```
#include <iostream>
using namespace std;
void showbits(unsigned char s)
//funkcja pokazuje reprezentację binarną znaku
{
    unsigned char wagi[8]={1,2,4,8,16,32,64,128}; //maska bitu wagi
    for(int i=7; i >= 0; i--)
    {
        int bit = (wagi[i] & s);
        if (bit !=0 )
            cout << '1';
        else
            cout << '0';
    }
}
//--Przykładowe operacje na bitach
int main()
{
    cout << "i\tbinarnie\tprzes.w lewo\tnegacja\n";
    for (int i=0; i<16; i++)
    {
        cout << i << "\t"; showbits(i); cout << "\t"; //dec oraz binarnie
                                                //przesunięcie o 1 bit w lewo:
        int j= i << 1;
        showbits(j);
        cout << "\t";
        int k= ~i;
        showbits(k);
        cout << endl; //negacja bitowa
    }
}
```

Wynik działania programu:

i	binarnie	przes.w lewo	negacja
0	00000000	00000000	11111111
1	00000001	00000010	11111110
2	00000010	00000100	11111101
3	00000011	00000110	11111100
4	00000100	00001000	11111011
5	00000101	00001010	11111010
6	00000110	00001100	11111001
7	00000111	00001110	11111000
8	00001000	00010000	11110111
9	00001001	00010010	11110110
10	00001010	00010100	11110101
11	00001011	00010110	11110100
12	00001100	00011000	11110011

13	00001101	00011010	11110010
14	00001110	00011100	11110001
15	00001111	00011110	11110000

System ósemkowy

Liczba ósemkowa jest liczbą zapisaną w pozycyjnym systemie ósemkowym, tj. za pomocą ośmiu cyfr, od 0 do 7, np. 074, 0322. W C++ liczba poprzedzona zerem jest traktowana jak liczba ósemkowa.

Zapis ósemkowy łatwo zamienia się na dwójkowy (i odwrotnie): wystarczy zamieniać cyfry ósemkowe na trzy cyfry dwójkowe lub odwrotnie. Przykład:

$$(78)_{10} = (001001110)_2 = (116)_8$$

System szesnastkowy

Liczba szesnastkowa pozwala zapisać w zwięzłej postaci długie liczby binarne. System szesnastkowy do oznaczania cyfr używa czwórek bitów (półbajtów), które zastępują ciągi bitów wg podanej poniżej tabeli B.4.

Tabela B.4. Tabela konwersji liczb szesnastkowych na dwójkowe i dziesiętne

Cyfra szesnastkowa	Wartość dziesiętna	Wartość dwójkowa
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

W języku C++ liczby szesnastkowe oznacza się przedrostkiem 0x.

Przykład konwersji liczby dwójkowej na szesnastkową:

$$(78)_{10} = (01001110)_2 = (4E)_{16}$$

Zmienne w pamięci komputera

Wiemy już, że komputer działa i funkcjonuje w systemie dwójkowym. Dla naszej wygody w programach czasami stosuje się zapisy liczbowe z wykorzystaniem systemu ósemkowego lub szesnastkowego, ale wynika to wyłącznie z chęci uproszczenia zapisu liczb i zminimalizowania

ryzyka błędów (liczby dwójkowe są długie i niezbyt czytelne dla większości osób). Możemy jednak zadać w tym miejscu pytanie, jak się ma zasada zapisu binarnego do języków programowania wykorzystujących przecież całkiem zrozumiałe, logiczne typy danych, takie jak `integer`, `char`, `float` itp.? Jak komputer przechowuje *liczbę* 15, a jak *ciąg znaków* „15”?

Tak jak wspomniałem o tym w rozdziale 5., lista typów podstawowych oraz ich precyzja jest w C++ określona przez standard tego języka, ale ich konkretna implementacja, np. liczba bitów zajmowanych przez zmienną określonego typu, zależy już od konkretnej technologii (sprzęt) i systemu operacyjnego. Architektura sprzętu (typ procesora komputera i obsługiwane przezeń operacje na tzw. rejestrach — komórkach pamięci specjalnego przeznaczenia) determinują zasady adresowania danych i kodu. Bezpośrednie programowanie procesora i zasobów sprzętowych umożliwia język niskiego poziomu, tzw. *assembler*. Biegle programowanie w *assemblerze* obecnie jest rzadkością nawet w środowiskach informatyków.

System operacyjny można traktować jako swego rodzaju nakładkę na sprzęt, pozwalającą izolować programy tworzone przez programistów od warstwy fizycznej zasobów komputera. Jedną z cech wymaganych od systemu operacyjnego jest zapewnienie bezpieczeństwa dostępu do zasobów sprzętowych: programy użytkownika są izolowane od programów systemowych, teoretycznie nie powinno być możliwe bezpośrednie adresowanie zasobów sprzętowych, np. wysyłanie komend do karty graficznej lub nagrywarki DVD. Oczywiście w praktyce bywa inaczej, ale warto mieć świadomość, że programy ingerujące w sprzęt są nieprzenośne i ich czas życia bywa dość krótki.

Po tych uwagach nie jest trudno się domyślić, że to właśnie kompilator danego języka programowania, przygotowany pod konkretny sprzęt (np. procesory Intel) i system (np. Windows XP/Vista) dokonuje „przekodowania” umownych, logicznych typów danych, na ich wersje „fizyczne”, umiejscowione „gdzieś w pamięci”. Programista nie musi się już martwić zasadami składowania zmiennych w pamięci operacyjnej, oczywiście pod warunkiem że nie ma takiej jawnej potrzeby (optymalizacja kodu, programowanie sprzętu).

Kodowanie znaków

Każdy użytkownik komputera na co dzień pracuje, używając klawiatury komputerowej, i w ogóle nie zastanawia się, w jaki sposób naciśnięcie klawisza z symbolem np. litery *A* powoduje wyświetlenie takiej litery na ekranie monitora. Dla nikogo nie powinno być już niespodzianką, że komputer nie zapisuje bezpośrednio znaków w pamięci, tylko używa do tego pewnej umownej reprezentacji dwójkowej. Kodowaniem i dekodowaniem znaków zajmuje się sprzęt, np. klawiatura „wie”, że po naciśnięciu znaku *A* powinna wysłać określony kod (zazwyczaj 65), podobnie ekran (karta graficzna) potrafi „narysować” określony znak na podstawie otrzymanego kodu.

Jednym z najczęściej stosowanych standardów kodowania znaków jest tzw. kod ASCII (ang. *American Standard Code for Information Interchange*). Kod ten jest 7-bitowy, co oznacza, że do dyspozycji mamy liczby z zakresu 0 – 127, którym przypisano litery alfabetu angielskiego, cyfry, znaki interpunkcyjne (tabela B.6) i polecenia sterujące sposobem wyświetlania znaków na drukarce lub terminalu znakowym (np. znak nowej linii, tabulacja, *backspace*) — (tabela B.5).

Tabela B.5. *Kod ASCII — kody sterujące*

Dec	Hex	Znak	Skrót
0	00	Null	NUL
1	01	Start Of Heading	SOH
2	02	Start of Text	STX
3	03	End of Text	ETX

Tabela B.5. Kod ASCII — kody sterujące — ciąg dalszy

Dec	Hex	Znak	Skrót
4	04	End of Transmission	EOT
5	05	Enquiry	ENQ
6	06	Acknowledge	ACK
7	07	Bell	BEL
8	08	Backspace	BS
9	09	Horizontal Tab	HT
10	0A	Line Feed	LF
11	0B	Vertical Tab	VT
12	0C	Form Feed	FF
13	0D	Carriage Return	CR
14	0E	Shift Out	SO
15	0F	Shift In	SI
16	10	Data Link Escape	DLE
17	11	Device Control 1 (XON)	DC1
18	12	Device Control 2	DC2
19	13	Device Control 3 (XOFF)	DC3
20	14	Device Control 4	DC4
21	15	Negative Acknowledge	NAK
22	16	Synchronous Idle	SYN
23	17	End of Transmission Block	ETB
24	18	Cancel	CAN
25	19	End of Medium	EM
26	1A	Substitute	SUB
27	1B	Escape	ESC
28	1C	File Separator	FS
29	1D	Group Separator	GS
30	1E	Record Separator	RS
31	1F	Unit Separator	US
32	20	Spacja	
127	7F	Delete	DEL

Tabela B.6. Kod ASCII — cyfry, litery alfabetu, znaki interpunkcyjne

Dec	Hex	Znak	Dec	Hex	Znak	Dec	Hex	Znak
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	\$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(72	48	H	104	68	h
41	29)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k

Tabela B.6. *Kod ASCII — cyfry, litery alfabetu, znaki interpunkcyjne — ciąg dalszy*

Dec	Hex	Znak	Dec	Hex	Znak	Dec	Hex	Znak
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	Y
58	3A	:	90	5A	Z	122	7A	Z
59	3B	;	91	5B	[123	7B	{
60	3C	<	92	5C	\	124	7C	
61	3D	=	93	5D]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			
64	40	@	96	60	`			

Siedem bitów w kodzie ASCII stanowiło pewną zaszczość historyczną i ponieważ komputery operowały już słowami 8-bitowymi, szybko uzupełniono kod ASCII o kody powyżej 127, gdzie wstawiono znaki semigraficzne, służące do tworzenia obramowań (np. ¶, ¶ itp.) oraz znaki narodowe. Rozszerzona część tablicy ASCII zawiera zestaw zależny od kraju i czasami producenta sprzętu komputerowego (drukarki, terminale znakowe). W celu obsługi polskich znaków stworzono wariant kodu ASCII o nazwie ISO 8859-2 (tzw. ISO Latin-2).

Niestety w międzyczasie powstało kilkadziesiąt innych standardów, o czym łatwo się przekonać, oglądając różne programy i strony internetowe (czasami w miejsce polskich znaków pojawiają się dziwne symbole). Świetne, tabelaryczne zestawienie standardów kodowania polskich znaków znajduje się na stronie internetowej:

http://pl.wikipedia.org/wiki/Sposób_kodowania_polskich_znaków.

Wadą standardu ASCII jest zamieszanie wprowadzone przez producentów oprogramowania i sprzętu (w zasadzie tylko część kodów 0... 127 jest jednoznaczna) i brak obsługi wielu języków na raz (do dyspozycji w wersji rozszerzonej standardu ASCII mamy tylko 8 bitów, co daje 256 możliwości).

Alternatywą dla ASCII staje się powoli 16-bitowy system kodowania Unicode, pozwalający na reprezentowanie do 65 tysięcy znaków w sposób jednoznaczny, nie są potrzebne żadne informacje dodatkowe, pozwalające rozróżniać języki czy wręcz pojedyncze znaki. Jako ciekawostkę można podać, że w pliku zgodnym z Unicode kolejność znaków określa kolejność ich faktycznego czytania, a nie oglądania (pomyśl o np. tekście artykułu gazetowego zawierającego tekst w języku angielskim z wtrąconymi słowami lub zdaniami arabskimi).

Pojemność Unicode wydaje się ogromna, choć znając historię wieży Babel, być może kiedyś i ten kod się zapełni w wyniku twórczej działalności człowieka!

Dodatek C

Kompilowanie programów przykładowych

Zawartość archiwum ZIP na ftp

Wszystkie programy przykładowe i materiały uzupełniające są umieszczone na serwerze ftp wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/algo4.zip>. Pobierz i rozpakuj te programy na dysk swojego komputera, aby je modyfikować lub sprawdzać ich działanie.

Struktura plików i folderów dołączonych do książki jest następująca:

- ◆ Folder *CPP* zawiera programy do kompilacji (rozszerzenie *.cpp*) i dwa pliki wsadowe, dokonujące hurtowej kompilacji przy pomocy GCC:
 - ◆ `kompiluj_gcc.bat` (dla systemu DOS/Windows)
 - ◆ `kompiluj_gcc.sh` (dla Linuxa).

Powyższe pliki wsadowe niczym się nie różnią, ale dla wygody przygotowałem wersje w obu standardach zapisu znaków nowej linii.

- ◆ Folder *Visual* zawiera gotowe projekty programów do kompilacji z użyciem kompilatora Microsoft Visual C++ Express Edition. Projekty zawarte w tym folderze używają plików źródłowych zawartych w *CPP*.
- ◆ Folder *Dodatki* zawiera materiały pobrane z Internetu, znalazło się tam m.in. dodatkowe oprogramowanie freeware, które powinno zainteresować zwłaszcza Czytelników pragnących eksperymentować z kodowaniem i kompresją danych. Oprócz tego znajdziesz tam listę komend GCC i inne uzupełniające materiały.

Darmowe kompilatory C++

Do kompilacji większości programów przykładowych powinno wystarczyć proste środowisko np. GCC obejmujące kompilatory C, C+ itp. GCC, czyli GNU Compiler Collection, dawniej znany pod nazwą GNU C Compiler, jest to zestaw kompilatorów różnych języków programowania, rozwijany w ramach projektu GNU i udostępniany na licencji GPL. Systemy oparte na Uniksie używają GCC jako podstawowego kompilatora, ale GCC może działać również w systemach Windows lub Mac OS (ten ostatni zresztą korzeniami tkwi w BSD Unix).



W systemach typu Linux/Unix kompilator C++ często jest już wstępnie zainstalowany. Aby to zweryfikować, spróbuj wykonać komendę `c++` lub `g++` z poziomu terminala tekstowego. Jeśli komenda nie zadziała, wejdź do instalatora Twojej dystrybucji Linuxa (np. YAST w Suse) i dokonaj wyboru pakietu do programowania w C++. W Linuksie Ubuntu, gdy brakuje kompilatora, wystarczy z poziomu konsoli zainstalować go, uruchamiając polecenie `sudo apt-get install g++`.

Aby pobrać GCC, zajrzyj pod adres <http://gcc.gnu.org>, gdzie w sekcji Download/ Binaries możesz wybrać wersję dla swojego systemu (np. Windows, Linux). Jeśli używasz Windows, to polecam instalację MinGW (<http://www.mingw.org>) — nazwa jest skrótem od „Minimalist GNU for Windows”. Pakiet instaluje się domyślnie w folderze C:\MinGW. Aby go używać, wystarczy dodać folder C:\MinGW\bin do zmiennej środowiskowej *Path* i od tego momentu można kompilować programy z poziomu konsoli¹.

Jeśli używasz komputera Apple i systemu Mac OS, to polecam rejestrację w serwisie Apple Developer Connection (<http://connect.apple.com>). Można tam np. pobrać całe środowisko graficzne Xcode, w ramach którego dostaniemy komplet kompilatorów, także najnowszy kompilator C++. Xcode jest ciężki w użyciu i generuje dużo nadmiarowych plików, ale możemy poprzestać na używaniu kompilatora C++ z linii poleceń terminala.

Użytkownikom Windows polecam jednak zainstalowanie świetnego, darmowego kompilatora Microsoft Visual C++ Express Edition, który można pobrać pod adresem <http://www.microsoft.com/express/download>. Na stronie producenta można pobrać go w wersji online (Web Install) lub jako obraz dysku DVD-ROM, np. do wypalenia programem Nero (Offline Install). Plik, który musimy w tym przypadku pobrać, zajmuje ok. 750 MB, zatem bez stałego dostępu do Internetu się nie obejdzie. Jeśli kogoś jednak przeraża rozmiar tej instalacji, to może skorzystać z darmowego kompilatora Dev C++, który choć dość stary (ostatnia stabilna wersja pochodzi z 2005 roku), jest nakładką na kompilator GNU C++ (MinGW) i oferuje polski interfejs użytkownika! Pakiet można pobrać pod adresem: <http://www.bloodshed.net>, sekcja *Download*. Instalacja jest bardzo prosta i sprowadza się do uruchomienia pliku instalacyjnego, a potem wybrania języka polskiego jako domyślnego.

Kompilacja i uruchamianie

Jak skompilować i uruchomić program z tej książki?

Pierwsze, co należy zrobić, to skopiować i rozpakować do dowolnego, roboczego katalogu pliki pobrane z *ftp* (pliki są zawarte w archiwum ZIP). Reszta zależy już od systemu operacyjnego i środowiska, które posiadamy. Ponieważ celem tej książki nie jest kurs użycia kompilatorów, opiszę tylko podstawowe warianty kompilacji i uruchamiania programów, które powinny okazać się wystarczające dla Czytelników używających różnych systemów i narzędzi deweloperskich.

GCC

Opisywany w niniejszym punkcie sposób kompilacji jest niezależny od systemu operacyjnego. Zakładam też, że chcesz pracować, używając tylko komend z linii poleceń (terminal tekstowy), co w dobie graficznych systemów operacyjnych jest dla wielu osób wielkim wyzwaniem!

Wykonaj następujące czynności (zakładam, że kompilator GCC jest już zainstalowany):

¹ Wejdź do *Panel sterowania/System i konserwacja/System*, następnie *Zaawansowane ustawienia systemu* i *Zmienne środowiskowe*, aby dopisać po średniku nową ścieżkę.

- ♦ Uruchom terminal (okienko linii poleceń). W systemie Windows można to zrobić, wywołując z menu *Start* program o nazwie *cmd* (*Start/Wszystkie programy/Akcesoria/Wiersz poleceń*), w Mac OS uruchom program *Terminal*. W Linuksie wyszukaj w swoim środowisku graficznym program *Terminal* (np. w Suse może to być *Komputer/Pokaż wszystkie programy/Terminal GNOME*).
- ♦ Przejdź do folderu, w którym znajduje się interesujący Cię plik w języku C++ rozpakowany z archiwum ZIP dołączonego do tej książki. Do nawigacji w linii poleceń możesz używać komend: `cd \`, `cd .`, `cd nazwa` pozwalających na, odpowiednio: przejście do folderu głównego, przejście do folderu nadrzędnego, przejście do podfolderu *nazwa*). W Ubuntu wskaż myszką folder, np. *CPP* i prawym klawiszem myszki wywołaj polecenie *Otwórz w terminalu*.
- ♦ Wpisz z linii poleceń komendę kompilacji:

```
c++ -o plik plik.cpp
```
- ♦ Powyższe polecenie skompiluje przykład w C++ zawarty w pliku o nazwie *plik.cpp*, tworząc w tym samym katalogu jego wersję wykonywalną o tej samej nazwie (w zależności od systemu może zostać doklejone rozszerzenie, np. dla Windows będzie to *.exe*).
- ♦ Skompilowany do postaci binarnej program możesz uruchomić już klasycznie, klikając go dwukrotnie myszką lub wywołując z tej samej konsoli tekstowej (przykłady na rysunku C.1). Oprócz komendy `cd` możesz używać `ls` do wyświetlania zawartości katalogu. Uruchomienie pliku wykonywalnego wymaga poprzedzenia go symbolami `./` (kropka i ukośnik), co stanowi cechę charakterystyczną terminala opartego na powłocie *bash*. W linii poleceń Windows wystarczy, że wpiszesz nazwę pliku wykonywalnego łącznie z rozszerzeniem *.exe*.

Mac OS

```
Terminal — bash — 71x24
imac-piotr-wroblewski-2:Desktop piotrwrblewski$ ls
$RECYCLE.BIN  bin      rek1.cpp
imac-piotr-wroblewski-2:Desktop piotrwrblewski$ g++ rek1.cpp -o rek1
imac-piotr-wroblewski-2:Desktop piotrwrblewski$ ./rek1
Element 7 nie został odnaleziony
Znalazłem szukany element 5
imac-piotr-wroblewski-2:Desktop piotrwrblewski$
```

Windows

```
Administrator: C:\Windows\System32\cmd.exe
C:\>cd algo4
C:\algo4>dir /w
Wolumin w stacji C to Uista
Numer seryjny woluminu: 0C1D-A9B9

Katalog: C:\algo4

[.]          [..]      REK1.CPP      REK1.exe
                2 plik(ów)      497 387 bajtów
                2 katalog(ów)   5 699 416 064 bajtów łącznie
C:\algo4>c++ REK1.CPP -o REK1
C:\algo4>REK1.exe
Element 7 nie został odnaleziony
Znalazłem szukany element 5
```

Linux

```
piotr@linux-z75z:~/Desktop
Plik Edycja Widok Terminal Karty Pomoc
piotr@linux-z75z:~$ ls
bin Desktop Documents public_html
piotr@linux-z75z:~$ cd Desktop/
piotr@linux-z75z:~/Desktop$ ls
rek1.cpp
piotr@linux-z75z:~/Desktop$ c++ rek1.cpp -o rek1
piotr@linux-z75z:~/Desktop$ ./rek1
Element 7 nie został odnaleziony
Znalazłem szukany element 5
piotr@linux-z75z:~/Desktop$
```

Rysunek C.1. Kompilacja i uruchamianie programu w konsoli tekstowej

Visual C++ Express Edition

W tym punkcie opisuję przypadek użycia graficznego środowiska Visual C++ Express Edition dostępnego dla systemu Windows, które pozwala na kompilowanie programów zarówno graficznych, jak i pracujących w trybie tekstowym.

Dla ułatwienia na *ftp* zawarłem gotowe projekty pod to środowisko, opiszę jednak także, jak utworzyć nowy konsolowy projekt w tym kompilatorze na podstawie gotowego pliku CPP.

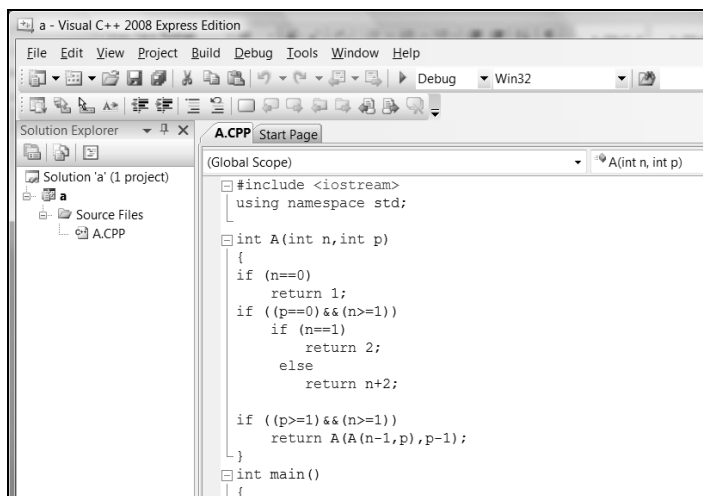
Otwieramy gotowy projekt

Aby otworzyć istniejący projekt stworzony w Visual C++ Express Edition, należy wykonać następujące czynności (rysunek C.2):

- ◆ Uruchom program Visual C++ z menu Start.
- ◆ Wybierz z menu *File/Open*, a następnie *Project/Solution* (możesz też użyć skrótu *Ctrl+Shift+O*).
- ◆ Wskaż na dysku plik projektu (rozszerzenie *.vcproj*) i otwórz go (przycisk *Otwórz*).
- ◆ Komplikacja i utworzenie kodu wynikowego: klawisz *F7* (także menu *Build/Build Solution*).
- ◆ Uruchomienie kodu: klawisz *F5*.

Rysunek C.2.

Otwieramy gotowy projekt Visual C++ Express Edition



Visual C++ po otwarciu projektu powinien w lewym panelu pokazać drzewko *Solution Explorer*, w którym można wyszukiwać pliki źródłowe (ang. *Source Files*) i załadować je do edytora wizualnego, podwójnie klikając w ich nazwę. Edytor w środowisku Visuala jest znakomity, podświetlanie słów kluczowych, pomoc w formatowaniu (wcięcia) i pomoc składniowa na pewno ułatwią pracę każdemu programiście.

Jeśli panel *Solution Explorer* został przez przypadek zamknięty i straciliśmy możliwość efektywnej pracy z kompilatorem, to nie stało się nic strasznego, w menu *View* można ponownie włączyć ten widok.

Warto mieć świadomość, że program konsolowy po uruchomieniu z poziomu środowiska Visual C++ może otworzyć swoje okno i szybko je zamknąć.

Aby go „zatrzymać” na ekranie, można do kodu funkcji `main` wprowadzić na koniec np. prostą instrukcję pobierania danych:

```
char klawisz; cin >> klawisz ;

lub

cin.get();
```

W systemie DOS lub Windows można użyć funkcji `getch()`:

```
#include <conio.h> //z powodu getch
...
getch();
```

Tworzymy nowy projekt konsolowy w Visual C++

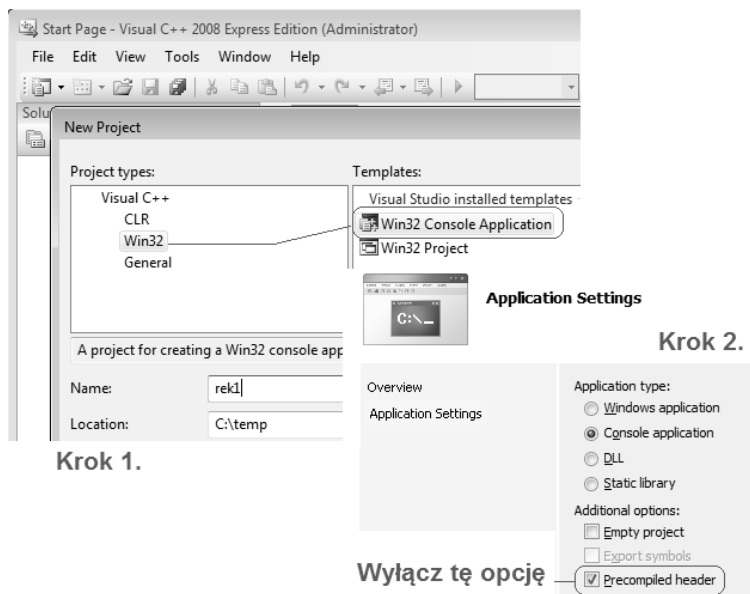
Na pewno prędzej czy później przyda nam się wiedza, jak utworzyć projekt w Visual C++. Niekoniecznie musimy dysponować plikiem źródłowym, ale założmy na początek, że takowy posiadamy i chcemy go np. dostosować do naszych potrzeb.

Aby na podstawie gotowego pliku *.cpp* utworzyć nowy projekt „gołego” języka C++, który będzie działał w konsoli tekstowej, należy wykonać następujące czynności:

- ♦ Uruchom program Visual C++ z menu Start.
- ♦ Wybierz z menu *File/New/Project/Win32* typ *Win32 Console Application*.
- ♦ Wpisz nazwę projektu (rysunek C.3).

Rysunek C.3.

Utworzenie tzw. projektu w Visual C++ Express Edition

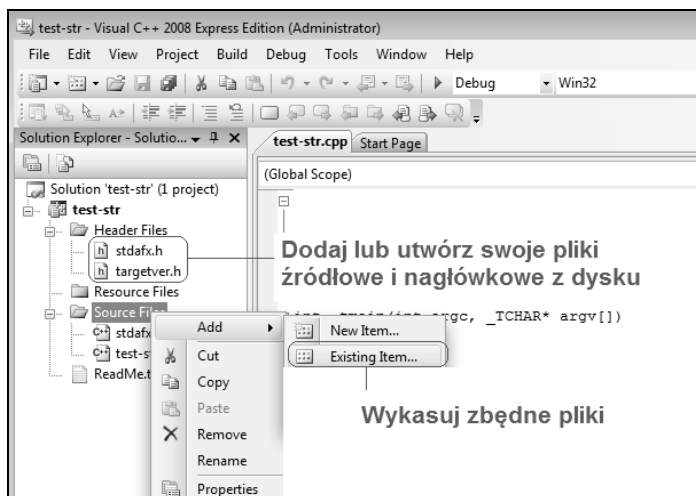


(Na rysunku pokazałem moment utworzenia projektu o nazwie *rek1*, którego pliki będą zawarte w folderze *c:\temp*. Dla uproszczenia nazwałem projekt tak samo, jak plik źródłowy programu, którego w nim użyję, ale oczywiście nie jest to obowiązkowe).

- ♦ W okienku *Win 32 Application Wizard*, który pojawi się na ekranie, wciśnij klawisz *Next* i wyłącz opcję *Precompiled headers*. To samo możesz zrobić później, w menu *Project\<nasza nazwa projektu>\Properties*, w sekcji *C/C++/Precompiled Headers* (działa dopiero po załadowaniu pliku *.cpp*).
- ♦ W kolejnym etapie (rysunek C.4) znajdziesz się już na ekranie znanego nam deweloperskiego środowiska graficznego (IDE), oferującego wygodny eksplorator plików projektowych i świetny edytor, wyposażony w funkcje wyróżniania słów kluczowych języka C++ i mechanizmy poprawiania formatowania kodu (np. wcięcia).
- ♦ W eksploratorze projektu wejdź do sekcji *Source Files* (czyli po polsku „pliki źródłowe”) i wykasuj (klawisz *Delete*) znajdujące się tam pliki (wersje na dysku twardym pozostaną nienaruszone; jeśli chcesz, to również je usuń).

Rysunek C.4.

Kompilacja projektu
w Visual C++ Express
Edition



- ◆ W eksploratorze projektu wejdź do sekcji *Header Files* (czyli po polsku „pliki nagłówkowe”) i wykasuj znajdujące się tam pliki (wersje na dysku twardym pozostaną nienaruszone; jeśli chcesz, to również je usuń).
- ◆ Od tego momentu możesz dodawać do projektu swoje własne pliki źródłowe: kliknij prawym klawiszem myszki na sekcji *Source Files* i z menu podręcznego wywołaj polecenie *Add/Existing item* (po polsku: dodaj istniejący element). Wskaż interesujący Cię plik źródłowy CPP pobrany np. z *ftp* lub stworzony przy pomocy zwykłego edytora (systemowy Notepad wystarczy!).
- ◆ Naciśnij klawisz *F7*, aby zbudować plik wykonywalny zawierający kod programu przykładowego. Jeśli kompilator wykryje błędy, popraw je².
- ◆ Przy pomocy menu *Start* uruchom program *cmd* (*Wiersz polecenia*) i przejdź do folderu *Debug* lub *Release* zawartego w lokalizacji podanej w polu *Location* na rysunku C.3). W naszym przykładzie będzie to *c:\temp\rek1\Debug*.
- ◆ Uruchom skompilowany program (rysunek C.5).

Rysunek C.5.

Uruchamianie
skompilowanego
programu konsolowego
w systemie Windows

```

C:\> Administrator: Wiersz polecenia
Microsoft Windows [Wersja 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\Administrator>cd \

C:\>cd temp\rek1\Debug

C:\temp\rek1\Debug>dir /w
Wolunin w stacji C to Vista
Numer seryjny woluninu: 0C1D-A9B9

Katalog: C:\temp\rek1\Debug

[.]          [..]          rek1.exe      rek1.ilc      rek1.pdb
3 plik(ów)   1 052 416 bajtów
2 katalog(ów) 6 569 467 904 bajtów wolnych

C:\temp\rek1\Debug>rek1
Element ? nie został odnaleziony
Znalazłem szukany element 5

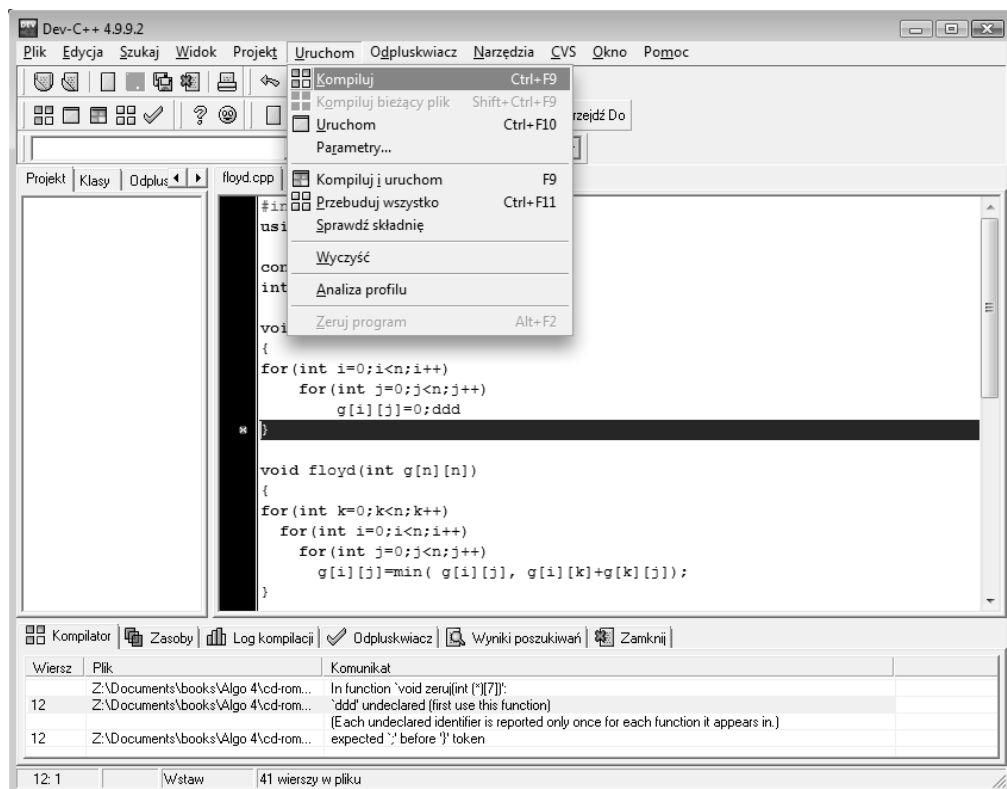
C:\temp\rek1\Debug>

```

² Programy są oczywiście sprawdzone przeze mnie, ale możesz je modyfikować, co czasami prowadzi do błędów.

Dev C++

Środowisko Dev C++ (rysunek C.6) jest dość popularne wśród początkujących programistów, gdyż nie wymaga zbyt dużo miejsca (pliki instalacyjne zajmują około 10 MB, a po rozpakowaniu na dysku około 60 MB). Podczas instalacji jedyne, na co trzeba uważać, to okienko konfiguracyjne, w którym możemy wybrać język polski dla komunikatów i graficznego interfejsu użytkownika.



Rysunek C.6. Darmowe środowisko Dev C++

Kompilacja programu przykładowego nie wymaga nawet założenia pliku projektu. Wystarczy otworzyć plik w edytorze (*Plik/Otwórz*) i skompilować go przy pomocy polecenia *Uruchom/Kompiluj* (*Ctrl+F9*). Jeśli w programie nie było błędów, to w tym samym folderze, w którym znajduje się plik źródłowy *.cpp*, zostanie zapisany plik wykonywalny *(.exe)*, który można uruchomić np. z linii poleceń lub z menu *Uruchom/Kompiluj* (*Ctrl+F10*). Polonizacja Dev C++ nie jest stuprocentowa, ale nawet osoba nieznająca angielskiego z łatwością zrozumie ostrzeżenia i komunikaty o błędach w programach, gdyż program wskazuje linijki kodu, w których jest błąd.

Literatura

- [Ara01] J. Arabas. *Wykłady z algorytmów ewolucyjnych*. Wydawnictwa Naukowo-Techniczne, 2001.
- [AHU03] A. V. Aho, J. E. Hopcroft i J. D. Ullmann. *Algorytmy i struktury danych*. Helion, 2003.
- [BB87] G. Brassard i P. Bratley. *Algorithmique conception et analyse*. Masson Les Presses de l'Université de Montréal, 1987.
- [BC89] L. Bolc i J. Cytowski. *Metody przeszukiwania heurystycznego*. PWN, 1989.
- [Ben92] J. Bentley. *Perelki oprogramowania*. WNT, 1992.
- [CLR01] Cormen T. H., Leiserson Ch. E., Rivest R.L. *Wprowadzenie do algorytmów — wydanie piąte*. Wydawnictwa Naukowo-Techniczne, 2001.
- [CP84] A. Couvert i R. Pendrono. *Techniques de programmation*. IFSIC Cours C45, Octobre 1984.
- [CR90] J. Chojcan i J. Rutkowski. *Zbiór zadań z teorii informacji i kodowania*. Skrypt nr 1501 Politechniki Śląskiej w Gliwicach, Gliwice 1990.
- [Del93] C. Delannoy. *Ćwiczenia z języka C++ programowanie obiektowe*. Wydawnictwa Naukowo-Techniczne, 1993.
- [DF89] E. Dijkstra i W. H. Feijen. *A Method of Programming*. Addison-Wesley Publishing Company, 1989.
- [Eck02] B. Eckel. *Thinking in C++*. Edycja polska. Helion, 2002.
- [FGS90] C. Froidevaux, M-C Gaudel, i M. Soria. *Types de données et algorithmes*. McGraw-Hill (Paris), 1990.
- [Gri84] D. Gries. *The Science of Programming*. Springer-Verlag, 1984.
- [Har92] D. Harel. *Algorithmics: The Spirit of Computing • Second Edition*. Addison-Wesley Publishing Company, 1992.
- [Hel86] J-M. Helary. *Algorithmique des graphes (version partielle)*. IFSIC Cours C66, 1986.
- [HS78] E. Horowitz i S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
- [Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.
- [Kas03] M. J. Kasperski. *Sztuczna Inteligencja*. Helion, 2003.
- [Kla87] Praca zbiorowa pod redakcją Jerzego Klamki. *Laboratorium metod numerycznych*. Skrypt nr 1305 Politechniki Śląskiej w Gliwicach, Gliwice 1987.
- [KM05] A. Koenig, B. Moo. *Język C++. Koncepcje i techniki programowania*. Helion, 2005.
- [Knu69] D. E. Knuth. *The Art of Computer Programming. • Volume 2: Seminumerical Algorithms*. Addison-Wesley Publishing Company, 1969.

- [Knu73] D. E. Knuth. *The Art of Computer Programming. • Volume 1: Fundamental Algorithms*. Addison-Wesley Publishing Company, 1973.
- [Knu75] D. E. Knuth. *The Art of Computer Programming. • Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, 1975.

Ostatnio w Polsce ukazało się wreszcie tłumaczenie dzieła Knutha, osobom nieprzepadającym za językiem angielskim można zatem polecić:

- [Knu01] D. E. Knuth. *Sztuka programowania* — tom 1, 2 i 3. Wydawnictwa Naukowo-Techniczne, 2001. (W roku 2007 został też wydany tom poświęcony algorytmom kombinatorycznym).
- [Kro89] D. Krob. *Algorithmique et structures de données*. Elipses, 1989.
- [Lou03] K. Loudon. *Algorytmy w C*. Helion, 2003.
- [Nil82] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982.
- [Poh89] I. Pohl. *C++ for C Programmers*. The Benjamin/Cummings Publishing Company Inc., 1989.
- [Sed92] R. Sedgewick. *Algorytmy w C++*. Wydawnictwo RM, 1999.
- [Sed03] R. Sedgewick. *Algorytmy w C++. Grafy*. Wydawnictwo RM, 2003.
- [Say02] K. Sayood. *Kompresja danych — wprowadzenie*. Read Me, 2002.
- [Str04] B. Stroustrup. *Język C++*. Wydanie siódme. Wydawnictwa Naukowo-Techniczne 2004. (Tłumaczenie z języka angielskiego).
- [WF92] K. Weiskamp i B. Flaming. *The Complete C++ Primer — 2nd ed.* Academic Press, Inc., 1992.

Spis tabel

Tabela 3.1. <i>Czasy wykonania programów dla algorytmów różnej klasy</i>	51
Tabela 3.2. <i>Złożoność teoretyczna algorytmów — przykłady</i>	55
Tabela 3.3. <i>Analiza przeszukiwania binarnego</i>	65
Tabela 7.1. <i>Przykład kodowania liter za pomocą 5 bitów</i>	169
Tabela 9.1. <i>Przykładowe rozwiązania problemu plecakowego</i>	201
Tabela 10.1. <i>Problem komiwojażera</i>	215
Tabela 10.2. <i>Kroki algorytmu Kruskala</i>	230
Tabela 10.3. <i>Kroki algorytmu Prima</i>	230
Tabela 10.4. <i>Problem doboru na przykładzie</i>	237
Tabela 13.1. <i>Uproszczony alfabet zdefiniowany dla celów kodowania</i>	266
Tabela 13.2. <i>Alfabet Morse’a</i>	276
Tabela 13.3. <i>Przykład kodowania znaków pewnego alfabetu 5-znakowego</i>	279
Tabela 13.4. <i>Prawdopodobieństwa występowania liter w języku polskim</i>	281
Tabela 13.5. <i>Format pliku GIF</i>	287
Tabela A.1. <i>Porównanie operatorów Pascala i C++</i>	299
Tabela B.1. <i>Prawa i twierdzenia algebry Boole’a</i>	314
Tabela B.2. <i>Arytmetyka liczb dwójkowych na przykładzie</i>	315
Tabela B.3. <i>Podstawowe operacje na liczbach dwójkowych</i>	315
Tabela B.4. <i>Tabela konwersji liczb szesnastkowych na dwójkowe i dziesiętne</i>	317
Tabela B.5. <i>Kod ASCII — kody sterujące</i>	318
Tabela B.6. <i>Kod ASCII — cyfry, litery alfabetu, znaki interpunkcyjne</i>	319

Spis ilustracji

Rysunek 1.1. <i>Etapy konstrukcji programu</i>	22
Rysunek 2.1. <i>„Sprzątanie klocków”, czyli rekurencja w praktyce</i>	28
Rysunek 2.2. <i>Drzewo wywołań funkcji silnia(3)</i>	31
Rysunek 2.3. <i>Obliczanie czwartego elementu ciągu Fibonacciego</i>	32
Rysunek 2.4. <i>Liczba wywołań funkcji MacCarthy’ego w funkcji parametru wywołania</i>	34
Rysunek 2.5. <i>Nieskończony ciąg wywołań rekurencyjnych</i>	36
Rysunek 2.6. <i>Spirala narysowana rekurencyjnie</i>	38
Rysunek 2.7. <i>Spirala narysowana rekurencyjnie — szkic rozwiązania + wynik</i>	39
Rysunek 2.8. <i>Kwadraty narysowane rekurencyjnie ($n = 3$)</i>	40
Rysunek 2.9. <i>Przeszukiwanie binarne na przykładzie</i>	43
Rysunek 2.10. <i>Trójkąty narysowane rekurencyjnie</i>	44
Rysunek 3.1. <i>Graficzna ilustracja czasów wykonania algorytmów różnej klasy</i>	52
Rysunek 3.2. <i>Żłudna wiara w moc komputera</i>	53
Rysunek 3.3. <i>Koszt zerowania tablicy</i>	57
Rysunek 4.1. <i>Sortowanie przez wstawianie (1)</i>	74
Rysunek 4.2. <i>Sortowanie przez wstawianie (2)</i>	74
Rysunek 4.3. <i>Sortowanie przez wstawianie (3)</i>	74
Rysunek 4.4. <i>Sortowanie „bąbelkowe”</i>	76
Rysunek 4.5. <i>Podział tablicy w metodzie Quicksort</i>	77
Rysunek 4.6. <i>Zasada działania procedury Quicksort</i>	78
Rysunek 4.7. <i>Budowa niezmiennika dla algorytmu Quicksort</i>	78
Rysunek 4.8. <i>Sortowanie metodą Quicksort na przykładzie</i>	79
Rysunek 4.9. <i>Kopiec i jego własności (1)</i>	80

Rysunek 4.10. <i>Kopiec i jego własności (2)</i>	80
Rysunek 4.11. <i>Przywracanie własności kopca na przykładzie</i>	81
Rysunek 4.12. <i>Ogólny model sortowania zewnętrznego</i>	87
Rysunek 5.1. <i>Typy rekordów używanych podczas programowania list</i>	93
Rysunek 5.2. <i>Przykład listy jednokierunkowej (1)</i>	94
Rysunek 5.3. <i>Przykład listy jednokierunkowej (2)</i>	94
Rysunek 5.4. <i>Dołączanie nowego elementu listy na jej początek</i>	97
Rysunek 5.5. <i>Dołączanie elementu listy z sortowaniem</i>	98
Rysunek 5.6. <i>Wstawianie nowego elementu do listy — analiza przypadków</i>	98
Rysunek 5.7. <i>Fuzja list na przykładzie</i>	102
Rysunek 5.8. <i>Sortowanie listy bez przemieszczania jej elementów (1)</i>	105
Rysunek 5.9. <i>Sortowanie listy bez przemieszczania jej elementów (2)</i>	105
Rysunek 5.10. <i>Tablicowa implementacja listy</i>	115
Rysunek 5.11. <i>Metoda „tablic równoległych” (1)</i>	117
Rysunek 5.12. <i>Metoda „tablic równoległych” (2)</i>	117
Rysunek 5.13. <i>Lista dwukierunkowa</i>	118
Rysunek 5.14. <i>Usuwanie danych z listy dwukierunkowej</i>	119
Rysunek 5.15. <i>Lista cykliczna</i>	119
Rysunek 5.16. <i>Stos i podstawowe operacje na nim</i>	120
Rysunek 5.17. <i>Tablicowa realizacja kolejki FIFO — koncepcja</i>	124
Rysunek 5.18. <i>Tablicowa realizacja kolejki FIFO — przykład</i>	126
Rysunek 5.19. <i>Konstrukcja sterty na przykładzie</i>	127
Rysunek 5.20. <i>Poprawne wstawianie nowego elementu do sterty</i>	128
Rysunek 5.21. <i>Ilustracja procedury NaDol</i>	130
Rysunek 5.22. <i>Drzewo i podstawowe pojęcia z nim związane</i>	132
Rysunek 5.23. <i>Drzewa binarne i wyrażenia arytmetyczne</i>	133
Rysunek 5.24. <i>Tablicowa reprezentacja drzewa</i>	134
Rysunek 5.25. <i>Tworzenie drzewa binarnego wyrażenia arytmetycznego</i>	135
Rysunek 5.26. <i>Kompresja danych zaletą Uniwersalnej Struktury Słownikowej</i>	140
Rysunek 5.27. <i>Reprezentacja słów w USS</i>	141
Rysunek 6.1. <i>Wieża Hanoi — prezentacja problemu</i>	151
Rysunek 6.2. <i>Wieża Hanoi — sposób rozwiązania</i>	152

Rysunek 7.1. <i>Idea transformacji kluczowej</i>	168
Rysunek 7.2. <i>Użycie list do obsługi konfliktów dostępu</i>	172
Rysunek 7.3. <i>Podział tablicy do obsługi konfliktów dostępu</i>	173
Rysunek 7.4. <i>Obsługa konfliktów dostępu przez próbkowanie liniowe</i>	174
Rysunek 7.5. <i>Utrudnione poszukiwanie danych przy próbkowaniu liniowym</i>	175
Rysunek 8.1. <i>Algorytm typu brute-force przeszukiwania tekstu</i>	179
Rysunek 8.2. <i>„Falszywe starty” podczas poszukiwania</i>	180
Rysunek 8.3. <i>Wyszukiwanie optymalnego przesunięcia w algorytmie K-M-P</i>	182
Rysunek 8.4. <i>Przesuwanie się wzorca w algorytmie K-M-P (1)</i>	183
Rysunek 8.5. <i>Przesuwanie się wzorca w algorytmie K-M-P (2)</i>	183
Rysunek 8.6. <i>Optymalne przesunięcia wzorca „ananas” w algorytmie K-M-P</i>	184
Rysunek 8.7. <i>Przeszukiwanie tekstu metodą Boyera i Moore'a</i>	186
Rysunek 9.1. <i>Mnożenie macierzy</i>	196
Rysunek 9.2. <i>Obliczanie wartości elementów ciągu Fibonacciego</i>	204
Rysunek 9.3. <i>Dwuwymiarowy wzór rekurencyjny</i>	205
Rysunek 9.4. <i>Problem 8 hetmanów</i>	209
Rysunek 10.1. <i>Model grafowy problemu mostów w Królewcu</i>	212
Rysunek 10.2. <i>Przykład grafu</i>	212
Rysunek 10.3. <i>„Normalizowanie” grafu (1)</i>	214
Rysunek 10.4. <i>„Normalizowanie” grafu (2)</i>	214
Rysunek 10.5. <i>Graf przejść dla maszyny stanów skończonych</i>	214
Rysunek 10.6. <i>Cykl Hamiltona</i>	215
Rysunek 10.7. <i>Algorytm Fleury'ego na przykładzie</i>	216
Rysunek 10.8. <i>Tablicowa reprezentacja grafu skierowanego</i>	217
Rysunek 10.9. <i>Tablicowa reprezentacja grafu skierowanego</i>	218
Rysunek 10.10. <i>Reprezentacja grafu za pomocą słownika węzłów</i>	219
Rysunek 10.11. <i>Przykładowe wykonanie algorytmu Roy-Warshalla</i>	222
Rysunek 10.12. <i>Poszukiwanie drogi w grafie</i>	223
Rysunek 10.13. <i>Algorytm Floyda-Warshalla (1)</i>	224
Rysunek 10.14. <i>Algorytm Floyda-Warshalla (2)</i>	225
Rysunek 10.15. <i>Drzewo rozpinające minimalne</i>	229
Rysunek 10.16. <i>Przeszukiwanie grafu „w głąb”</i>	231

Rysunek 10.17. <i>Przeszukiwanie grafu „wszerz”</i>	233
Rysunek 10.18. <i>Zawartość kolejki podczas przeszukiwania grafu „wszerz”</i>	233
Rysunek 10.19. <i>Problem doboru</i>	235
Rysunek 10.20. <i>Listy rankingowe w problemie doboru</i>	236
Rysunek 10.21. <i>Drzewo rozpinające — ćwiczenie</i>	239
Rysunek 11.1. <i>Algorytm Newtona odszukiwania miejsc zerowych</i>	242
Rysunek 11.2. <i>Interpolacja funkcji $f(x)$ za pomocą wielomianu $F(x)$</i>	244
Rysunek 11.3. <i>Tablica różnic centralnych w metodzie Stirlinga</i>	246
Rysunek 11.4. <i>Przybliżone całkowanie funkcji</i>	247
Rysunek 12.1. <i>Schemat systemu eksperckiego</i>	255
Rysunek 12.2. <i>Model neuronu</i>	256
Rysunek 12.3. <i>Sieć neuronowa</i>	257
Rysunek 12.4. <i>Problem konika szachowego (1)</i>	258
Rysunek 12.5. <i>Problem konika szachowego (2)</i>	258
Rysunek 12.6. <i>Przykład drzewa pewnej wymagowanej gry</i>	259
Rysunek 12.7. <i>Reguła mini-max</i>	261
Rysunek 12.8. <i>Pojęcie linii otwartych w grze w „kółko i krzyżyk”</i>	262
Rysunek 12.9. <i>Generowanie listy możliwych ruchów gracza na podstawie danego węzła</i> ...	264
Rysunek 12.10. <i>Kodowanie listy węzłów potomnych przy użyciu tylko jednego węzła</i> ...	264
Rysunek 13.1. <i>Kodowanie symetryczne</i>	267
Rysunek 13.2. <i>System kodujący z kluczem publicznym</i>	269
Rysunek 13.3. <i>Kompresja wykorzystująca matematyczne modelowanie zbioru danych</i>	277
Rysunek 13.4. <i>Przykład drzewa kodowego (1)</i>	280
Rysunek 13.5. <i>Przykład drzewa kodowego (2)</i>	280
Rysunek 13.6. <i>Konstrukcja kodu Huffmana — faza redukcji</i>	282
Rysunek 13.7. <i>Konstrukcja kodu Huffmana — faza tworzenia kodu</i>	282
Rysunek 13.8. <i>Słownikowa metoda kodowania</i>	284
Rysunek A.1. <i>Terminologia w programowaniu obiektowym</i>	305
Rysunek A.2. <i>Przykład dziedziczenia własności w C++</i>	311

Rysunek C.1. <i>Kompilacja i uruchamianie programu w konsoli tekstowej</i>	323
Rysunek C.2. <i>Otwieramy gotowy projekt Visual C++ Express Edition</i>	324
Rysunek C.3. <i>Utworzenie tzw. projektu w Visual C++ Express Edition</i>	325
Rysunek C.4. <i>Kompilacja projektu w Visual C++ Express Edition</i>	326
Rysunek C.5. <i>Uruchamianie skompilowanego programu konsolowego w systemie Windows</i>	326
Rysunek C.6. <i>Darmowe środowisko Dev C++</i>	327

Skorowidz

- , 298
- #include, 296
- ++, 298
- 255.cpp, 274
- 3DES, 268
- A*, 234, 260
- A.cpp, 66
- abstrakcyjne struktury danych, 92
- Abu Ja'far Mohammed ibn Mūsā al-Khowārizmī, 17
- acykliczny graf skierowany, 214
- Adleman L., 268
- ADSL, 265
- Aiken H., 20
- alfabet Braille'a, 276
- alfabet Morse'a, 275, 276
- algebra Boole'a, 313, 314
- algorytm, 17
 - A*, 260
 - Bellmana-Forda, 228
 - Boyera i Moore'a, 185
 - brute-force, 179
 - cechy, 18
 - cięcie α - β , 260, 261
 - definicja, 17
 - DES, 268
 - Dijkstry, 227
 - Euklidesa, 17
 - Fleury'ego, 216
 - Floyda, 225
 - Floyda-Warshalla, 224
 - generowanie automatyczne, 25
 - genetyczny, 210
 - heurystyczny, 208
 - Huffmana, 279, 281
 - język prezentacji, 10
 - K-M-P, 181, 182
 - Kruskala, 229
 - kryteria wyboru, 49
 - LZW, 283, 284
 - mini-max, 260, 264
 - Newtona odszukiwania miejsc zerowych, 242
 - niestabilny, 41
 - opis słowny, 23
 - poprawność, 24
 - poziom abstrakcji opisu, 23
 - prefiksowy, 279
 - Prima, 229, 230
 - Quicksort, 78
 - Rabina i Karpa, 187
 - Roy-Warshalla, 221
 - RSA, 273
 - sposób prezentacji, 23
 - SSS*, 260
 - Strassena, 197
- algorytmika, 9, 18
- algorytmika grafów, 211
- algorytmy numeryczne, 241
 - algorytm Newtona odszukiwania miejsc zerowych, 242
 - całkowanie funkcji metodą Simpsona, 247
 - gauss.cpp, 250
 - interpol.cpp, 245
 - interpolacja funkcji metodą Lagrange'a, 244
 - iteracyjne obliczanie wartości funkcji, 243
 - metoda Gaussa, 248
 - metoda Stirlinga, 245
 - newton.cpp, 242
 - pochodna.cpp, 246
 - poszukiwanie miejsc zerowych funkcji, 241
 - rozwiązywanie układów równań liniowych, 248
 - różniczkowanie funkcji, 245
 - simpson.cpp, 247
 - wartf.cpp, 243

algorytmy przeszukiwania, 165
 binary-i.cpp, 166
 hashing, 167
 linear.cpp, 165
 przeszukiwanie binarne, 166
 przeszukiwanie liniowe, 165
 transformacja kluczowa, 167
 algorytmy sortowania, 73
 duże pliki wejściowe, 85
 Heap Sort, 80
 klasa $O(N \log N)$, 77
 klasa $O(N^2)$, 74, 75
 kryteria wyboru algorytmu, 87
 quicksort, 77
 shaker-sort, 76
 sortowanie bąbelkowe, 75
 sortowanie przez kopcowanie, 80
 sortowanie przez scalanie, 83
 sortowanie przez wstawianie, 74
 sortowanie przez wytrząsanie, 76
 sortowanie systemowe, 86
 sortowanie zewnętrzne, 84
 algorytmy żarłoczne, 199
 greedy.cpp, 201
 problem plecakowy, 200
 schemat algorytmu, 199
 analiza bezpieczeństwa sieci, 212
 analiza obwodów elektronicznych, 212
 analiza wyrażenia beznawiasowego, 135
 analiza złożoności algorytmów, 49
 ananas.cpp, 184
 AND, 315
 argc, 303
 argv[0], 303
 arytmetyka dużych liczb, 267, 270
 arytmetyka liczb dwójkowych, 315
 ASCII, 266, 279, 318
 assembler, 23
 assembly output, 185
 atoi(), 86
 automat skończony, 214
 automatyczne generowanie algorytmów, 25

B

Babbage Ch., 19
 bajt, 314
 baza wiedzy, 255, 256
 Bellman R. E., 202, 228
 BFS, 231
 binary_s.cpp, 45
 binary_search(), 64
 binary-i.cpp, 166
 bit_operations.cpp, 316

blok kodu, 295
 bm(), 187
 bm.cpp, 186
 bool, 89
 Borland C++, 185
 Boyer R. S., 181
 breadthf.cpp, 233
 breadth-first search, 231
 brute-force, 179, 275
 bubble.cpp, 75
 BULL Gamma3, 20

C

C++, 10, 23, 295
 #include, 296
 blok, 295
 deklaracje nagłówkowe, 297
 destruktor, 305
 dziedziczenie własności, 308
 funkcje, 297
 iteracje, 302
 klasy, 305
 kod warunkowy, 311
 komentarze, 295
 konstruktor, 305
 main(), 295
 metody, 304
 obiekty, 304, 305
 operacje arytmetyczne, 298
 operacje logiczne, 298
 operacje na plikach, 303
 operatory, 298, 299
 parametry programu main(), 303
 pętle, 302
 pliki dołączane, 296
 podprogramy, 296
 procedury, 296
 program, 295
 programowanie obiektowe, 304
 protected, 309
 przeciążanie, 310
 przededefiniowanie operatorów, 307
 referencje, 300
 return, 297
 składowe statyczne klas, 308
 struktury, 301
 struktury rekurencyjne, 303
 switch, 301
 tablice, 297, 300
 tekst, 295
 this, 305
 using namespace, 296

- void, 296
- wskazniki, 299
- zmienne dynamiczne, 299
- cache, 69
- calloc(), 94
- całkowanie funkcji, 247
 - metoda Simpsona, 247
- Carnot L. M., 19
- case, 302
- char, 89
- ciąg Fibonnaciego, 31, 69, 203
 - programowanie dynamiczne, 203
- ciągi znaków, 90, 318
- cięcie α - β , 260, 261
- COBOL, 147
- Complex, 306
- complex.cpp, 307
- complex.h, 305
- Cook S. A., 181
- cout, 295
- cykl Eulera, 215, 216
- cykl Hamiltona, 215
- cykl skierowany, 214
- czas jednostkowy wykonania instrukcji, 59
- czas wykonania, 50, 54
- cząstkowy dobór, 236

D

- DAG, 214
- DARPA, 254
- Data Encryption Standard, 268
- debugger, 25
- default, 302
- dekompozycja problemu, 38, 195
- delete, 100
- depthf.cpp, 231
- depth-first search, 231
- derekursywacja, 147, 151
 - eliminacja zmiennych lokalnych, 154
 - metoda funkcji przeciwnych, 156
 - schemat typu if-else, 160
 - schemat typu while, 159
 - schemat z podwójnym wywołaniem rekurencyjnym, 162
 - schematy, 158
 - stos, 154
- DES, 268
- destruktor, 100, 305
- Dev C++, 13, 327
- DFS, 231
- Diffie W., 268
- digraf, 213

- Dijkstra E., 21, 25, 227
- directed acyclic graph, 214
- directed graph, 213
- długość ciągów znakowych, 91
- do...while, 302
- dobor.cpp, 237
- domknięcie przechodnie grafu, 221
- double, 89
- droga w grafie, 213
- drzewa, 131
 - głębokość węzła, 132
 - korzeń, 131
 - liść, 131
 - m-drzewa, 132
 - ojciec, 134
 - potomek lewy, 134
 - potomek prawy, 134
 - reprezentacja C++, 132
 - tablice, 134
 - Uniwersalna Struktura Słownikowa, 139
 - uporządkowane, 132
 - węzły, 131
 - węzły końcowe, 131
 - wysokość węzła, 132
- drzewa binarne, 93, 126, 131, 132, 133, 135
 - komórka, 132
 - wypisywanie drzewa w postaci wrostkowej, 137
- wyrazen.cpp, 136
- wyrażenia arytmetyczne, 135
- wysokość, 132
- drzewo gry, 259
- drzewo kodowe, 279
- drzewo rozpinające minimalne, 228
- dyn.cpp, 205
- dziedzic.cpp, 309
- dziedzic.h, 309
- dziedziczenie własności, 308
- dziel i zwyciężaj, 83, 192, 203
 - algorytm Strassena, 197
 - min_max2(), 194
 - min-max.cpp, 193
 - mnożenie liczb całkowitych, 197
 - mnożenie macierzy, 195
 - odnajdywanie największego i najmniejszego elementu w tablicy, 193
 - quicksort, 199
- dziel_i_rządź(), 192

E

- Eckert J. P., 20
- EDVAC, 20
- eliminacja Gaussa, 248, 249

eliminacja zmiennych lokalnych, 155, 157
 else, 148
 end–recursion, 150
 ENIAC, 20
 erastot.cpp, 291
 etapy konstrukcji programu, 22
 Euklides, 17
 Euler L., 211

F

false, 89
 fib(), 69
 fib-dyn.cpp, 204
 FIFO, 123, 124, 125
 First In First Out, 123
 Fleury H., 216
 float, 89
 Floyd R., 21
 floyd.cpp, 224
 floyd2.cpp, 225
 for, 27, 302
 Ford L. R. Jr., 228
 format GIF, 286
 Forth, 136
 FORTRAN, 147, 251
 fraktale, 27
 funkcja Ackermanna, 66
 funkcja MacCarthy'ego, 33, 41
 funkcja O, 55, 56
 funkcja odwrotna, 157
 funkcje, 297
 inline, 111
 modulo, 188
 nagłówek, 37
 parametry domyślne, 37
 przeciążanie, 111
 wskaźniki do funkcji, 108
 wywołanie przez wartość, 36
 zaprzyjaźnione, 102, 307
 funkcje H, 169, 188
 mnożenie, 171
 suma modulo 2, 170
 suma modulo R_{\max} , 170
 fuzja list, 102

G

gauss.cpp, 250
 GCC, 13, 322
 GCD, 44
 getmaxx(), 39
 getmaxy(), 39

getx(), 39
 gety(), 39
 GIF, 267, 283, 286
 GIF87a, 286
 GIG89a, 286
 głębokość węzła, 132
 głowa, 94, 124
 GNU C++, 185
 GO, 260
 Gödel K., 20
 Goldman A., 215
 Gosper R. W., 181
 goto, 148, 149
 gra w „kółko i krzyżyk”, 260
 grafy, 93, 211, 212
 acykliczne skierowane, 214
 algorytm Bellmana-Forda, 228
 algorytm Dijkstry, 227
 algorytm Fleury'ego, 216
 algorytm Floyda, 225
 algorytm Floyda-Warshalla, 224
 algorytm Kruskala, 229
 algorytm Prima, 229, 230
 algorytm Roy-Warshalla, 221
 cykl Eulera, 215, 216
 cykl Hamiltona, 215
 cykle, 214
 DAG, 214
 diagonalne, 220
 digraf, 213
 domknięcie przechodnie, 221
 droga, 213
 eulerowskie, 215
 floyd.cpp, 224
 floyd2.cpp, 225
 implementacja, 218
 kompoz.cpp, 220
 kompozycja, 220
 liczba chromatyczna grafu, 213
 macierz sąsiedztwa, 217
 minimalizowanie konfliktów, 235
 minimalne drzewo rozpinające, 228
 nieskierowane, 214
 normalizacja, 214
 numery węzłów, 213
 operacje, 220
 operacje matematyczne, 220
 parzysty stopień, 216
 planarne, 213
 podgraf, 213
 poszukiwanie drogi, 223
 potęga, 220
 problem doboru, 235
 problem komiwojażera, 215

przechodnie, 221
przeszukiwanie, 230
przeszukiwanie w głąb, 231
przeszukiwanie wszcz, 231, 232
przeszukiwanie zstępujące, 231
regulane, 213
reprezentacja, 217
reprezentacja tablicowa, 217
route.cpp, 223
skierowane, 213
słownik węzłów, 217, 218
spójne, 213
stopień wejściowy wierzchołka, 213
suma, 220
symetria, 221
ścieżka, 213
tablica dwuwymiarowa, 217
warshall.cpp, 222
węzeł początkowy, 213
węzły, 213
wierzchołki, 213
zbiory, 219
grafy stanów, 258
greedy, *Patrz* algorytmy żarłoczne
greedy.cpp, 201
gry dwuosobowe, 259

H

H (funkcja), 169, 188
Hanoi, 151, 156
hanoi.cpp, 152
hanoi_it.cpp, 162
hanoi_iter(), 163
hash.cpp, 177
hashing, 167
heap, 126
Heap Sort, 80
heap.cpp, 81
Hellman M., 268
heurystyka, 208, 230
Hilbert D., 20
hill-climbing, 235
historia maszyn algorytmicznych, 18
Hoare C. A. R., 21
Hollerith H., 19
horner.cpp, 271
Huffman(), 283

I

IBM, 20
IBM 604, 20
IBM 650, 20
IDE, 13

if, 148
IFIP, 21
iloczyn logiczny, 315
implementacja grafów, 218
implementacja kolejki FIFO, 124
implementacja zbioru znaków, 144
indukcja matematyczna, 25
inline, 111
insert.cpp, 75
int, 89
INT_MAX, 90
INT_MIN, 90
interfejs użytkownika, 22
interpol.cpp, 245
interpolacja funkcji, 244
interpolacja funkcji metodą Lagrange'a, 244
ISO 8859-2, 320
ISO Latin-2, 320
iteracje, 27, 302
iteracyjne obliczanie wartości funkcji, 243

J

Jacquard J. M., 19
Java, 22
język, 23
język asemblera, 23
język C++, 10, 295
język Java, 22
język prezentacji programów, 23
język programowania, 10
język strukturalny, 23

K

Karp R. M., 181
klasa algorytmu, 51, 65
klasa O, 56
klasy, 305
 destruktor, 100
 funkcje zaprzyjaźnione, 102
 metody statycznej, 308
 pola statyczne, 308
 składowe statyczne, 308
 szablonowe, 121
klucz, 168, 187, 219
klucz prywatny, 268
klucz publiczny, 268
K-M-P, 181, 182
 ananas.cpp, 184
 kmp.cpp, 183
 optymalne przesunięcia wzorca, 184
 przesuwanie się wzorca, 183
 tablica przesunięć, 183

- kmp.cpp, 183
- Knuth D. E., 181
- kod
 - ASCII, 279, 318, 319
 - Huffmana, 267, 279, 280
 - nadmiarowy, 267
 - nierównomierny, 266
 - równomierny, 266
 - warunkowy, 311
 - wykonywalny, 22
 - źródłowy, 22
- kodowanie danych, 265, 267
 - 3DES, 268
 - asymetryczne, 268
 - DES, 268
 - klucz prywatny, 268
 - klucz publiczny, 268
 - LZW, 267, 283
 - podpis cyfrowy, 269
 - problem transmisji klucza, 268
 - RSA, 269
 - symetryczne, 267, 268
 - z kluczem publicznym, 267, 268
 - z pomocą słownika, 283, 284
- kodowanie liter za pomocą 5 bitów, 169
- kodowanie znaków, 318
- kody ASCII, 266
- kolejka, 125, 233
- kolejka FIFO, 123
 - głowa, 124
 - implementacja, 124
 - kolejka.cpp, 125
 - kolejka.h, 124
 - ogon, 124
 - realizacja tablicowa, 124
- kolejka LIFO, 120
- kolejka priorytetowa, 125
 - operacje, 126
- kolejka.cpp, 125
- kolejka.h, 124
- komentarze, 295
- komiwojażer, 215
- kompilacja, 10, 321, 322
- kompilator, 22, 148, 212, 321
 - Borland C++, 185
 - C++, 321
 - GCC, 321, 322
 - GNU C++, 185
 - Visual C++ Express Edition, 323
- kompletne drzewo binarne, 126
- kompoz.cpp, 220
- kompozycja grafów, 220
- kompresja, 139, 140, 265, 266, 275
 - algorytm Huffmana, 281
 - algorytmy, 277
 - bezstratna, 276
 - GIF, 267, 283, 286
 - LZW, 283
 - metoda Huffmana, 279
 - poprzez modelowanie matematyczne, 277
 - redundancja, 277
 - RLE, 278
 - stopień kompresji, 277
 - stratna, 276
 - szybkość działania, 277
- komputer, 20
- konik szachowy, 258
- konstruktor, 305
- kontroler wyводу, 256
- kopiec, 80
- korzeń, 131
- kółko i krzyżyk, 259, 260, 262
 - generowanie listy możliwych ruchów, 264
 - kodowanie listy węzłów potomnych, 264
 - linie otwarte, 262
- Kruskal J., 229
- kryptosystem RSA, 269
- kwadraty „parzyste”, 40
- kwadraty.cpp, 40
- kwadraty_win.cpp, 41

L

- Last In First Out, 120
- LCS, 206
- lcs.cpp, 207
- Leibniz G. W., 19
- Lempel A., 283
- LF, 266
- liczba chromatyczna grafu, 213
- liczby, 318
- liczby całkowite, 89
- liczby zespolone, 305
- liczby zmiennopozycyjne, 89
- LIFO, 120
- linear.cpp, 165
- lineto(), 39
- Lisp, 256
- LISP, 23, 102
- LISTA, 95, 107
- lista sąsiedztwa, 217
- lista.cpp, 97
- lista.h, 95
- lista_tab.cpp, 116
- lista2.cpp, 109
- lista2.h, 107, 110
- listy, 93
 - cykliczne, 119
 - implementacja tablicowa, 115

- listy dwukierunkowe, 118
 - struktura wewnętrzna, 118
 - usuwanie elementów, 119
 - listy jednokierunkowe, 93
 - dołączanie elementu, 97, 116
 - dołączanie elementu z sortowaniem, 98
 - fuzja list, 102
 - głowa, 94
 - implementacja, 95
 - LISTA, 95, 107
 - lista.cpp, 97
 - lista.h, 95
 - lista_tab.cpp, 116
 - lista2.cpp, 109
 - lista2.h, 107, 110
 - metoda tablic równoległych, 117
 - ogon, 94
 - rekord natury informacyjnej, 93
 - rekord o charakterze roboczym, 93
 - reprezentacja tablicowa, 115
 - sortowanie bez przemieszczania, 105
 - struktura informacyjna, 94
 - tworzenie, 96
 - usuwanie elementów, 109, 116
 - usuwanie ostatniego elementu, 100
 - usuwanie wskaźników, 112
 - wady i zalety, 104
 - wstawianie elementu, 98
 - wyszukiwanie, 109, 110
 - listy rankingowe, 235
 - liść, 131
 - long, 89
 - longest common subsequence, 206
 - Lovelace A., 19
 - LPTR_INFO, 109
 - LZW, 267, 283
 - dekodowanie, 286
 - kodowanie, 285
- L**
- łamanie szyfrów, 275
 - brute-force, 275
 - dedukcja na podstawie fragmentu
 - lub całości treści, 275
 - kradzież klucza, 275
 - metoda „na siłę”, 275
- M**
- macierz kierowania ruchem, 222
 - macierz sąsiedztwa, 217
 - macierz trójkątna, 249
 - main(), 295
 - maksimum w tablicy liczb, 193
 - malloc(), 94
 - MARK 1, 20
 - Markow A. A., 20
 - maszyna, 20
 - maszyna algorytmiczna, 18
 - maszyna analityczna, 19
 - maszyna Moore’a, 214
 - maszyna stanów skończonych, 214
 - maszyna tkacka Jacquarda, 19
 - maszyna Turinga, 20
 - Mathcad, 241
 - Mathematica, 241
 - Matlab, 241
 - Mauchly J. W., 20
 - McCarthy J., 21
 - m-drzewa, 132
 - merge.cpp, 83
 - metoda Floyd’a, 24
 - metoda funkcji przeciwnych, 156
 - metoda Gaussa, 248, 249
 - metoda Huffmana, 279
 - metoda Newtona, 241
 - metoda niezmienników, 24
 - metoda Simpsona, 247, 248
 - metoda Stirlinga, 245
 - metoda tablic równoległych, 117
 - metoda transformacji kluczowej, 176
 - metody, 304
 - statycznej, 308
 - metody programowania, 191
 - metody prymitywne, 274
 - miara złożoności obliczeniowej, 50
 - miejsca zerowe funkcji, 241
 - min_max2(), 194
 - minimalizowanie konfliktów, 235
 - minimalne drzewo rozpinające, 228
 - mini-max, 260, 264
 - MiniMax(), 261
 - minimum w tablicy liczb, 193
 - min-max.cpp, 193
 - mnożenie liczb całkowitych, 197
 - mnożenie macierzy, 195
 - model, 20
 - model neuronu, 256
 - model sortowania zewnętrznego, 87
 - modelowanie matematyczne, 277
 - modelowanie problemów, 212
 - modelowanie rozwiązywanego zagadnienia, 257
 - modulo, 188, 273
 - moduł dialogowy, 256
 - Moore J. S., 181
 - Morris J. H., 181

Morse S., 275
 motor wywodu, 256
 moveto(), 39
 MYCIN, 255
 myślenie rekurencyjne, 38

N

najbardziej czasochłonna operacja, 53, 59
 najdłuższa wspólna podsekwencja, 205
 największy wspólny dzielnik, 17
 napisy, 90
 negacja, 315
 Neumann, Johannes von, 20
 neuron, 256
 new, 94
 newton.cpp, 242
 niezmiennik, 24
 notacja dużego O, 56
 notacja Landaua, 56
 numery węzłów, 213
 NWD, 17, 44

O

O(1), 55
 O(2n), 55
 O(log n), 55
 O(n), 55
 O(n²), 55
 O(n³), 55
 obiekty, 304, 305
 oblicz.cpp, 293
 obliczanie wartości funkcji, 243
 obwód zamknięty, 222
 odnajdywanie największego i najmniejszego elementu w tablicy, 193
 odpluskwanie, 21
 odwrot2.cpp, 161
 Odwrotna Notacja Polska, 135, 137
 odwrotna.cpp, 157
 ogon, 94, 124
 ojciec, 134
 O-notacja, 55, 56, 57
 ONP, 135, 137
 operacje arytmetyczne, 298
 operacje arytmetyczne na liczbach dwójkowych, 315
 operacje logiczne, 298
 operacje logiczne na liczbach dwójkowych, 315
 operacje na grafach, 220
 operacje na plikach, 303
 operandy, 135

operatory, 135, 209, 298, 299
 opis słowny, 23
 optymalizacja algorytmów, 147
 optymalizacja programów, 68
 OR, 315

P

palindro.cpp, 292
 pamięć, 73
 parametry domyślne, 37
 parametry programu main(), 303
 Pascal, 10, 24
 Pascal B., 19
 pętle, 27, 302
 pivot, 77
 pliki, 303
 pliki dołączane, 296
 pliki GIF, 287
 pliki wykonywalne, 22
 pochodna.cpp, 246
 podejście iteracyjne, 27
 podgraf, 213
 podjeżdżanie, 235
 podpis cyfrowy, 269
 podprogramy, 296
 podwójne kluczowanie, 175
 pola statyczne, 308
 pop(), 120, 122
 poprawność algorytmów, 24
 porównywanie sekwencji kodów, 206
 post_2.cpp, 46
 postać uwikłana, 243
 Postscript, 136
 poszukiwanie drogi w grafie, 223
 poszukiwanie miejsc zerowych funkcji, 241
 pot.cpp, 292
 potęga grafu, 220
 potomek lewy, 134
 potomek prawy, 134
 poziomy abstrakcji opisu, 23
 Pratt V. R., 181
 prawdopodobieństwa występowania liter w języku polskim, 281
 prefiks, 279
 prezentacja algorytmów, 23
 Prim R. C., 230
 private, 96, 102
 problem 8 hetmanów, 209
 problem doboru, 235, 237
 cząstkowy dobór, 236
 dobor.cpp, 237
 listy rankingowe, 235
 problem komiwojażera, 215

problem konika szachowego, 258
problem mostów w Królewcu, 212
problem NP-zupełny, 215
problem optymalnego doboru, 212
problem plecakowy, 200
problem transmisji klucza, 268
problem wyrażania preferencji, 235
procedury, 296
program, 10, 22, 295
 etapy konstrukcji, 22
 rekurencyjny, 30, 36, 62
 warunki końcowe, 25
 warunki wstępne, 25
programowanie, 21, 191, 208
programowanie dynamiczne, 33, 202, 203
 ciąg Fibonacciego, 203
 dyn.cpp, 205
 etapy, 203
 fib-dyn.cpp, 204
 inicjacja, 203
 koncepcja, 203
 lcs.cpp, 207
 najdłuższa wspólna podsekwencja, 205
 progresja, 203
 równania z wieloma zmiennymi, 204
programowanie obiektowe, 304
programowanie typu dziel i zwyciężaj, 192
Prolog, 256
PROLOG, 23, 102
prostota algorytmu, 49
protected, 102, 309
próbkowanie liniowe, 173, 176
przeciążanie, 310
przeciążanie funkcji, 111
przedefiniowanie operatora, 101
przedefiniowanie operatorów, 307
przedrostek, 279
przepełnienie stosu, 33, 67
przesunięcie bitowe, 315
przeszukiwanie, 165
przeszukiwanie binarne, 42, 64, 65, 166, 199
przeszukiwanie grafów, 230
 A*, 234
 BFS, 231
 breadthf.cpp, 233
 depthf.cpp, 231
 DFS, 231
 ekspansja węzłów, 231
 metoda podejżdżania, 235
 strategie, 234
 w głąb, 231
 wszerz, 231, 232
 z powracaniem, 234
 zstępujące przeszukiwanie, 231

przeszukiwanie liniowe, 165
przeszukiwanie tekstów, 179
 algorytm Boyera i Moore'a, 185
 algorytm K-M-P, 181, 182
 algorytm Rabina i Karpa, 187
 algorytm typu brute-force, 179
ananas.cpp, 184
bm.cpp, 186
kmp.cpp, 183
rk.cpp, 188
txt-1.cpp, 180
przydzielanie pamięci, 94
przypadek najgorszy, 60
przypadek najlepszy, 60
przypadek średni, 61
przypadek typowy, 61
public, 96
push(), 120, 122

Q

qsort2.cpp, 88
quick.cpp, 79
quicksort, 77, 199
 budowa niezmiennika, 78
 podział tablicy, 77
Quicksort(), 78

R

Rabin M. O., 181
RAM, 73
RC, 62
redukcja wsteczna, 249
redundancja, 277
referencje, 300
reguła mini-max, 261
rek1.cpp, 29
rek2.cpp, 30
rek3.cpp, 32
rek4.cpp, 33
rek5.cpp, 37
rekordy, 90
rekurencja, 27, 62, 191
 ciąg Fibonacciego, 31
 dekompozycja problemu, 38
 drzewo wywołań, 31
 funkcja MacCarthy'ego, 33
 ilustracja, 28
 kontekst, 37
 kwadraty „parzyste”, 40
 myślenie rekurencyjne, 38
 naturalna, 36

rekurencja

- nieskończona ilość wywołań, 34
- pamięciożerność, 41
- poprawność definicji, 35
- poziomy, 30, 31, 37
- problemy, 31
- przebiegi, 27
- przekazywanie parametrów, 31
- przepełnienie stosu, 33
- rozkład na problemy elementarne, 28
- silnia, 30, 37
- skrótna rekurencja, 42
- spiralą, 38
- sposób wykonywania programu, 30
- sprzątanie klocek, 28
- typy programów, 36
- uwagi praktyczne, 41
- z parametrem dodatkowym, 36, 37, 153
- zajętość pamięci, 33
- zakończenie algorytmu, 28
- zakończenie programu, 29
- zasada działania, 27
- zmniejszanie rozmiaru problemu, 41

rekursja, 27

relacje binarne, 221

relaksacja, 227

reprezentacja grafów, 217

reprezentacja problemów, 257

reprezentacja tablicowa grafów, 231

return, 297

rev_tab.cpp, 44

reverse-engineeringu, 268

REVERSI, 260

Rivest R., 268

rk.cpp, 188

RLE, 278

RO, 63

route.cpp, 223

routing matrix, 222

rozdzielenie i scalanie połączone z sortowaniem, 85

rozkład logarytmiczny, 64

rozmiar danych, 51

rozwiązanie ogólne, 63

rozwiązanie równania rekurencyjnego, 63

rozwiązanie szczególne, 63

rozwiązanie układów równań, 248

rozwiązanie układów równań liniowych, 248

równania z wieloma zmiennymi, 204

równanie charakterystyczne, 62

różnica symetryczna, 315

różniczkowanie funkcji, 245

RS, 63

RSA, 268, 269

ruchy dozwolone, 258

Run Length Encoding, 278

S, Ś

scalaj.cpp, 82

scalanie zbiorów posortowanych, 82

schemat Hornera, 243, 271

schematy derekursywacji, 158

- if-else, 160
- while, 159
- z podwójnym wywołaniem rekurencyjnym, 162

Segmentation fault, 34

set.cpp, 144

shaker.cpp, 77

shaker-sort, 76

Shamir A., 268

SI, 253, 254

sieci neuronowe, 256

- nauczanie, 257
- struktura, 257

silnia, 30, 37, 51, 53, 63, 153

silnia(), 53, 153

silnia2(), 37

simpson.cpp, 247

sito Erastotenesa, 289, 291

składowe statyczne klas, 308

słownik węzłów, 218, 231

słowniki, 139

słownikowa metoda kodowania, 284

sortowanie, 73

- algorytm klasy $O(N \log N)$, 77
- algorytm klasy $O(N^2)$, 74, 75
- algorytmy, 73
- bąbelkowe, 75
- Heap Sort, 80
- przez kopcowanie, 80
- przez scalanie, 83
- przez wstawianie, 74
- przez wytrząsanie, 76
- quicksort, 77
- sterta, 131
- systemowe, 86
- szybkie, 77
- wewnętrzne, 73
- zewnętrzne, 73, 84

spiralą, 38

spiralą.cpp, 39

spiralą_win.cpp, 40

SRL, 62

SSS*, 260

Stack overflow, 33, 34, 67

stany, 209

std.cpp, 35

- sterta, 125, 126
 - implementacja, 128
 - kolejka priorytetowa, 127
 - sortowanie, 131
 - sterta.h, 128
 - tworzenie, 127
 - wstawianie elementów, 128, 129
- Sterta, 128
- sterta.h, 128
- stopień kompresji, 277
- stopień wejściowy wierzchołka grafu, 213
- stos, 119
 - operacje, 120
 - pop(), 120, 122
 - push(), 120, 122
 - STOS, 121
 - stos.cpp, 123
 - stos.h, 121
 - zasada działania, 120
- STOS, 121
- stos.cpp, 123
- stos.h, 121
- Strassen V., 196
- strategia gry, 259
- strategia przeszukiwania, 260
- string, 92
- string.cpp, 92
- struktury, 301
- struktury danych, 89, 92
 - drzewa, 131
 - grafy, 211
 - kolejka priorytetowa, 125
 - kolejki FIFO, 123
 - listy jednokierunkowe, 93
 - sterta, 125
 - stos, 119
 - zbiory, 143
- suma grafów, 220
- suma logiczna, 315
- suma modulo 2, 170, 315
- suma modulo R_{\max} , 170
- switch, 301
- symulacja inteligentnego zachowania, 253
- syn, 134
- system ekspercki, 255
 - baza wiedzy, 255, 256
 - kontroler wyводу, 256
 - moduł dialogowy, 256
 - motor wyводу, 256
 - MYCIN, 255
 - sztuczny lekarz, 255
- system kodujący z kluczem publicznym, 269
- system liczbowy, 313
 - dwójkowy, 313, 314

- dziesiętny, 314
- ósemkowy, 317
- szesnastkowy, 317
- tabela konwersji, 317
- system operacyjny, 22
- system uzupełnienia dwójkowego, 314
- systemsort.cpp, 86
- szachy, 260
- szereg rekurencyjny liniowy, 62
- Sztuczna Inteligencja, 253, 254
 - cele, 254
 - drzewa gier, 259
 - grafy stanów, 258
 - reprezentacja problemów, 257
 - sieci neuronowe, 256
 - system ekspercki, 255
- sztuczny lekarz, 255
- szukaj.cpp, 60
- szyfr blokowy, 268
- ścieżka w grafie, 213

T

- tablica dwuwymiarowa, 217
- tablica przesunięć, 183
- tablica różnic centralnych, 245
- tablice, 90, 300
 - implementacja kolejki FIFO, 124
 - implementacja listy, 115
 - listy, 115
 - reprezentacja drzewa, 134
- tablice równoległe, 134
- techniki optymalizacji programów, 68
- techniki programowania, 191
- tekst, 179
- tekst źródłowy, 22
- template, 122
- teoria automatów, 214
- teoria gier, 230
- test Turinga, 254
- this, 305
- tictac.cpp, 260, 262, 263
- transformacja kluczowa, 167, 176
 - funkcje H, 169
 - konflikty dostępu, 171
 - podwójne kluczowanie, 175
 - próbkowanie liniowe, 173
 - strefa podstawowa, 173
 - strefa przepełnienia, 173
 - tablice, 173
 - zastosowania, 176
- traveling salesman problem, 215
- trojkaty.cpp, 46
- true, 89

Turing A. M., 20, 254

tworzenie

drzewa binarne, 135

listy jednokierunkowe, 96

program, 22

tworzenie, 135

txt-1.cpp, 180

typy danych, 89

podstawowe, 89

wbudowane, 89

zakres dozwolonych wartości, 90

typy złożoności obliczeniowej, 59

U

układ równań, 248

unie, 90

union, 90

UNIVAC 1, 20

Uniwersalna Struktura Słownikowa, 139

using namespace, 296

USS, 139

obsługa, 141

reprezentacja słów, 141

wyszukiwanie słów, 142

zasada działania, 140

uss.cpp, 140

V

Visual C++ Express Edition, 323

kompilacja, 326

projekt konsolowy, 325

Win 32 Application Wizard, 325

void, 296

W

warshall.cpp, 222

wartf.cpp, 243

wartości logiczne, 89

warunki końcowe, 25

warunki wstępne, 25

wektory, 118

Welch T., 283

while, 27, 302

wielom.cpp, 271

wielom2.cpp, 272

wielomiany, 271

wielowątkowość, 69

wieża Hanoi, 151, 156, 162, 199

Wirth N., 21

wsk_fun.cpp, 108

wskaźniki, 89, 299

wskaźniki do funkcji, 108

wydajność oprogramowania, 51

wyrazen.cpp, 136

wyrażanie preferencji, 235

wyrażenia arytmetyczne, 133, 135

wysokość drzewa binarnego, 132

wysokość węzła, 132

wywołanie przez wartość, 36

wywołanie terminalne, 150

wyznacznik Vandermonde'a, 244

wzór Simpsona, 247

wzór Stirlinga, 245

X

XOR, 274, 315

xor.cpp, 274

Z

zajętość pamięci, 50

zamiana dziedziny równania rekurencyjnego, 66

zbiory, 143, 217

implementacja, 144

set.cpp, 144

zerowanie fragmentu tablicy, 57

Ziv J., 283

złożoność algorytmów, 49

złożoność obliczeniowa, 49, 65, 68

złożoność praktyczna, 54

złożoność teoretyczna, 54

zmienne, 10, 317

zmienne dynamiczne, 299

zmienne globalne, 154

zmienne lokalne, 154

znaki, 89, 318

znaki.cpp, 91

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

