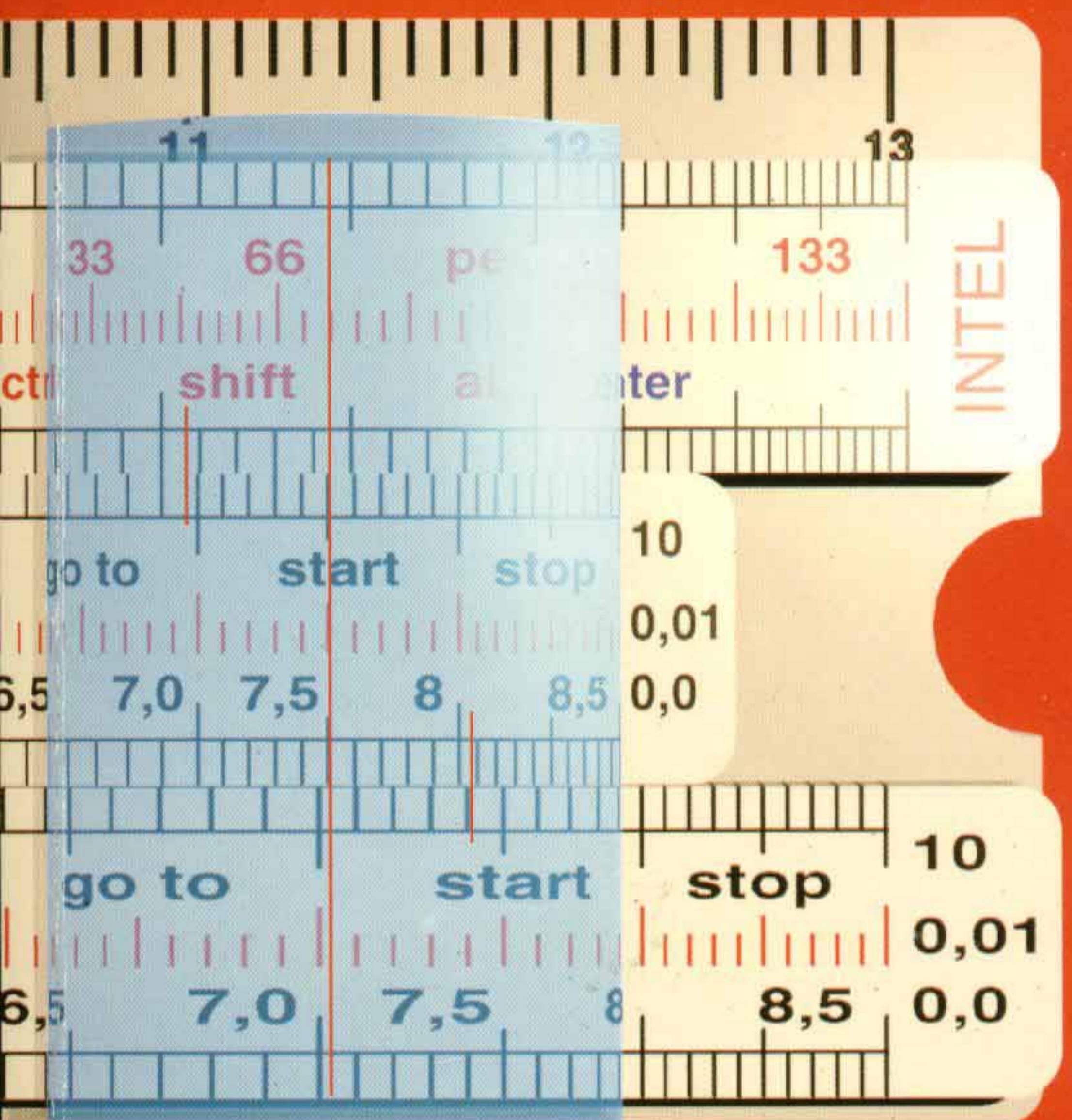


Algorytmy

struktury danych i techniki programowania

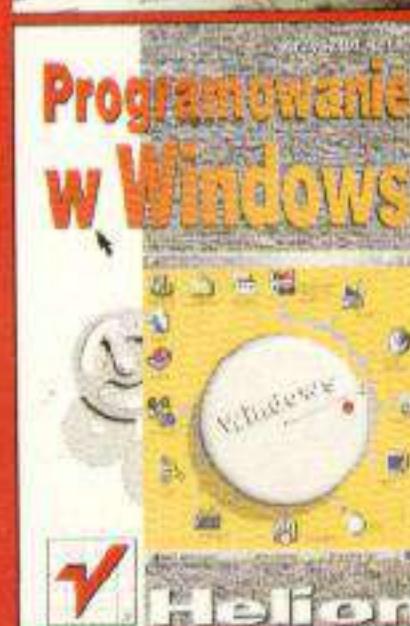
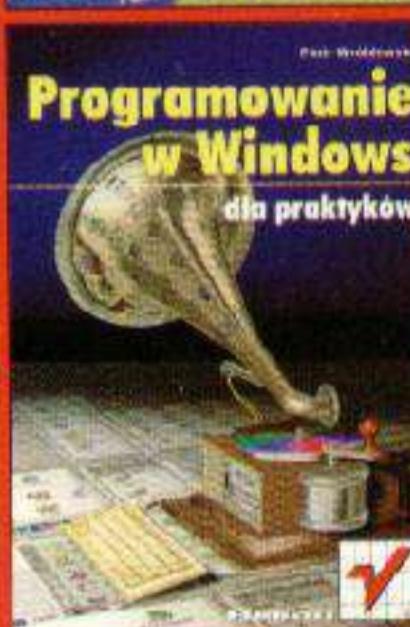
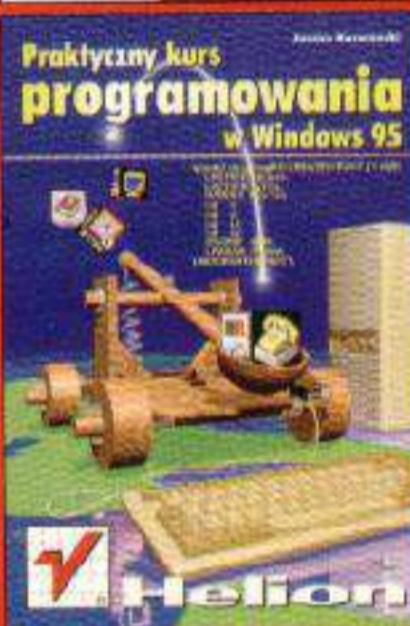
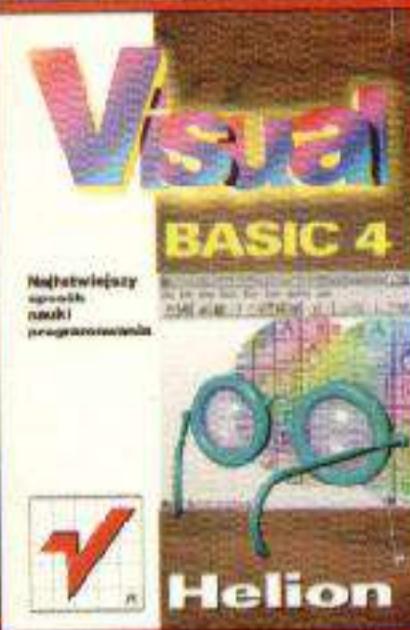
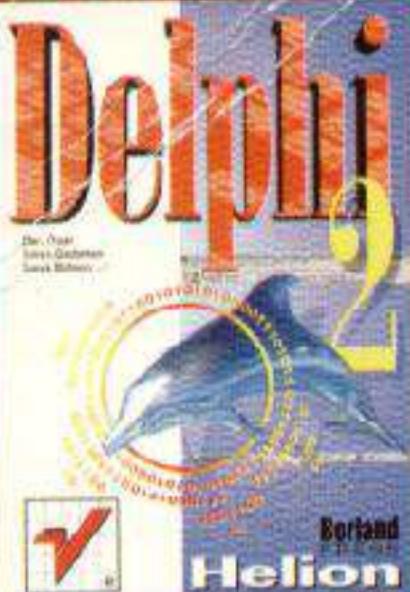
Piotr Wróblewski



Wydanie drugie
poprawione i uzupełnione



helion



Algorytmy

struktury danych i techniki programowania

„Algorytmy, struktury danych i techniki programowania” to nowoczesny podręcznik dla wszystkich osób, które w codziennej pracy programistycznej odczuwają potrzebę szybkiego odszukania pewnych informacji z dziedziny algorytmiki w celu zastosowania ich w swoich programach.

Książka niniejsza została stworzona według zasady:

minimum teorii – maksimum praktyki



Zawiera dyskietkę

Duża liczba zadań i programy znajdujące się na dyskietce powinny umożliwić szybkie zastosowanie w praktyce omawianego materiału.

Techniki rekurencyjne: co to jest rekurencja i jak ją stosować w praktyce? Analiza sprawności algorytmów: kilka prostych metod pozwalających porównywać efektywność algorytmów.

Sortowanie danych: najpopularniejsze procedury sortujące.

Struktury danych: listy, kolejki, zbiory i drzewa w ujęciu praktycznym.

Derekursywacja: jak zamienić program rekurencyjny (czasami bardzo czasochłonny) na jego wersję iteracyjną?

Algorytmy przeszukiwania: przeszukiwanie liniowe, binarne i transformacja kluczowa (ang. hashing).

Przeszukiwanie tekstów: opis najbardziej znanych metod przeszukiwania tekstów (brute-force, K-M.-P., Boyera i Moore'a, Rabina i Karpa).

Zaawansowane techniki programowania: dziel-i-rządź, programowanie dynamiczne, algorytmy żarłoczne (ang. greedy).

Algorytmika grafów: opis jednej z najciekawszych struktur danych występujących w informatyce.

Algorytmy numeryczne: jak zastosować komputery w matematyce do wykonywania obliczeń przybliżonych?

Sztuczna inteligencja: czy komputery mogą myśleć?

Kodowanie i kompresja danych: opis najbardziej znanych popularnych metod kodowania i kompresji danych: systemu kryptograficznego z kluczem publicznym i metody Huffmana.

Zadania: zrób to sam!

ISBN 83-86718-64-1



9 788386 718641

Wydawnictwo Helion

ul. Pszczyńska 89, 44-100 Gliwice, POLAND

✉ 44-100 Gliwice, skr. poczt. 462, POLAND

☎ tel./fax (32) 38-81-54, GSM 0(602) 38-81-54

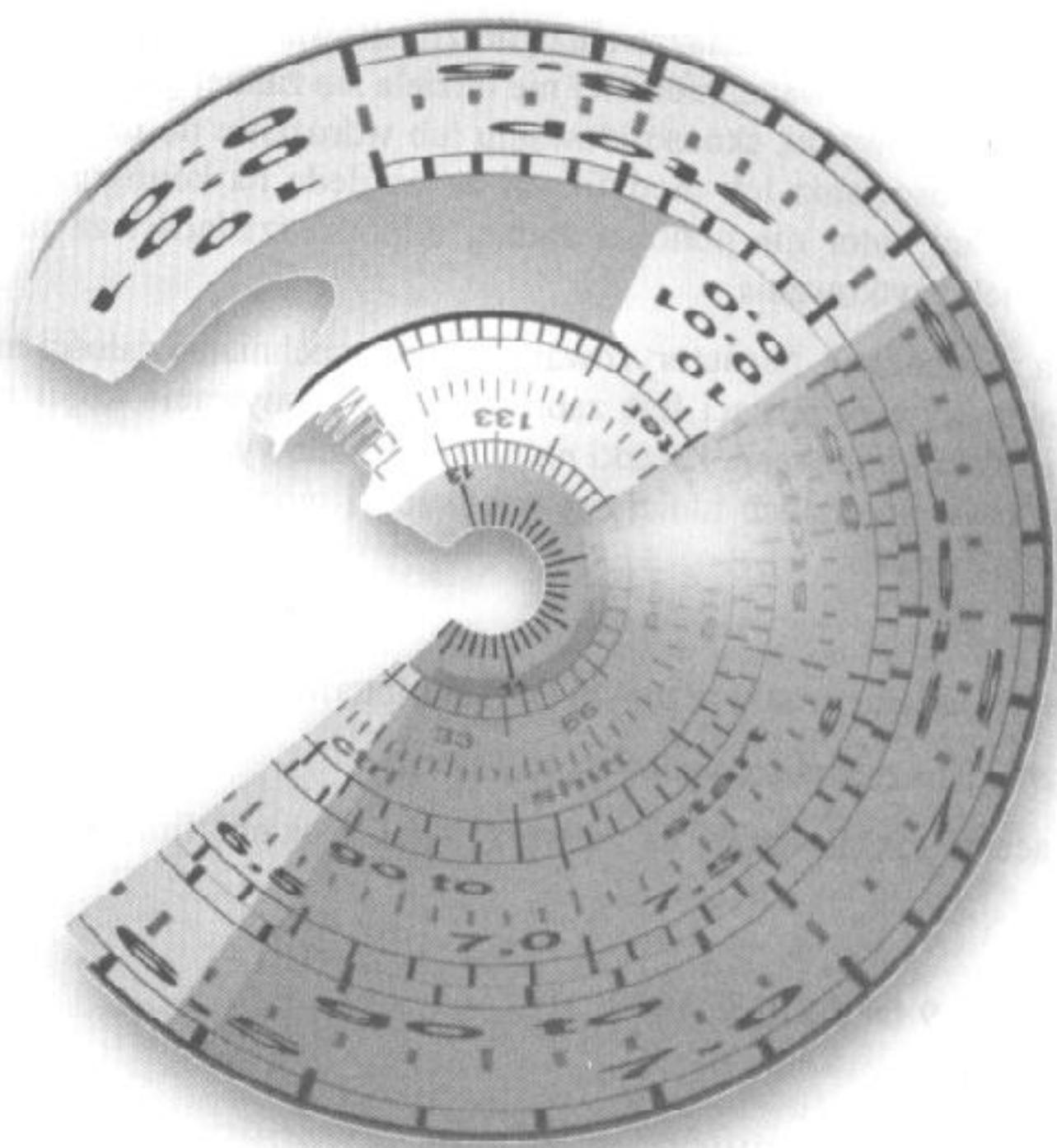
fax (32) 38-81-01, e-mail: helion@helion.com.pl

<http://www.helion.com.pl>

Piotr Wróblewski

Algorytmy struktury danych i techniki programowania

Wydanie drugie poprawione i uzupełnione



Helion

Spis treści

Przedmowa.....	9
Rozdział 1 Zanim wystartujemy.....	17
1.1. Jak to wcześniej było, czyli wyjątki z historii maszyn algorytmicznych.....	19
1.2. Jak to się niedawno odbyło, czyli o tym kto „wymyślił” metodologię programowania.....	21
1.3. Proces koncepcji programów.....	22
1.4. Poziomy abstrakcji opisu i wybór języka.....	23
1.5. Poprawność algorytmów.....	25
Rozdział 2 Rekurencja	29
2.1. Definicja rekurencji	29
2.2. Ilustracja pojęcia rekurencji.....	31
2.3. Jak wykonują się programy rekurencyjne?.....	33
2.4. Niebezpieczeństwa rekurencji	34
2.4.1. Ciąg Fibonacciego.....	35
2.4.2. Stack overflow!.....	36
2.5. Pułapek ciąg dalszy.....	37
2.5.1. Stąd do wieczności	38
2.5.2. Definicja poprawna, ale.....	38
2.6. Typy programów rekurencyjnych.....	40
2.7. Myślenie rekurencyjne.....	42
2.7.1. Spirala.....	42
2.7.2. Kwadraty „parzyste”.....	44
2.8. Uwagi praktyczne na temat technik rekurencyjnych.....	45
2.9. Zadania	47
2.10. Rozwiązania i wskazówki do zadań.....	49
Rozdział 3 Analiza sprawności algorytmów.....	53
3.1. Dobre samopoczucie użytkownika programu.....	54

3.2. Przykład 1: Jeszcze raz funkcja silnia.....	57
3.3. Przykład 2: Zerowanie fragmentu tablicy	61
3.4. Przykład 3: Wpadamy w pułapkę	64
3.5. Przykład 4: Różne typy złożoności obliczeniowej.....	65
3.6. Nowe zadanie: uprościć obliczenia!	68
3.7. Analiza programów rekurencyjnych	68
3.7.1. Terminologia	69
3.7.2. Ilustracja metody na przykładzie	71
3.7.3. Rozkład „logarytmiczny”	72
3.7.3.....	72
3.7.4. Zamiana dziedziny równania rekurencyjnego.....	74
3.7.5. Funkcja Ackermanna, czyli coś dla smakoszy	75
3.8. Zadania	76
3.9. Rozwiązania i wskazówki do zadań.....	78
Rozdział 4 Algorytmy sortowania.....	81
4.1. Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	82
4.2. Sortowanie bąbelkowe, algorytm klasy $O(N^3)$	84
4.3. Quicksort, algorytm klasy $O(N \log_2 N)$	87
4.4. Uwagi praktyczne	90
Rozdział 5 Struktury danych.....	93
5.1. Listy jednokierunkowe.....	94
5.1.1. Realizacja struktur danych listy jednokierunkowej	96
5.1.2. Tworzenie listy jednokierunkowej.....	98
5.1.3. Listy jednokierunkowe – teoria i rzeczywistość	108
5.2. Tablicowa implementacja list	122
5.2.1. Klasyczna reprezentacja tablicowa.....	122
5.2.2. Metoda tablic równoległych	124
5.2.3. Listy innych typów	127
5.3. Stos	128
5.3.1. Zasada działania stosu	128
5.4. Kolejki FIFO	133
5.5. Sterty i kolejki priorytetowe	136
5.6. Drzewa i ich reprezentacje.....	143
5.6.1. Drzewa binarne i wyrażenia arytmetyczne	147
5.7. Uniwersalna struktura słownikowa	152
5.8. Zbiory	159
5.9. Zadania	161
5.10. Rozwiązania zadań	162
Rozdział 6 Dereksywacjia	165
6.1. Jak pracuje kompilator?.....	166
6.2. Odrobina formalizmu... nie zaszkodzi!	169
6.3. Kilka przykładów dereksywacji algorytmów.....	170
6.4. Dereksywacja z wykorzystaniem stosu.....	174
6.4.1. Eliminacja zmiennych lokalnych	175
6.5. Metoda funkcji przeciwnych.....	177
6.6. Klasyczne schematy dereksywacji.....	180

6.6.1. Schemat typu <i>while</i>	181
6.6.2. Schemat typu <i>if... else</i>	182
6.6.3. Schemat z podwójnym wywołaniem rekurencyjnym	185
6.7. Podsumowanie	187
Rozdział 7 Algorytmy przeszukiwania	189
7.1. Przeszukiwanie liniowe	189
7.2. Przeszukiwanie binarne	190
7.3. Transformacja kluczowa	191
7.3.1. W poszukiwaniu funkcji H	193
7.3.2. Najbardziej znane funkcje H	194
7.3.3. Obsługa konfliktów dostępu	197
7.3.4. Zastosowania transformacji kluczowej	204
7.3.5. Podsumowanie metod transformacji kluczowej	204
Rozdział 8 Przeszukiwanie tekstów	207
8.1. Algorytm typu <i>brute-force</i>	207
8.2. Nowe algorytmy poszukiwań	210
8.2.1. Algorytm K-M-P	211
8.2.2. Algorytm Boyera i Moore'a	216
8.2.3. Algorytm Rabina i Karpa	218
Rozdział 9 Zaawansowane techniki programowania	223
9.1. Programowanie typu „dziel-i-rządź”	224
9.1.1. Odszukiwanie minimum i maksimum w tablicy liczb	225
9.1.2. Mnożenie macierzy o rozmiarze $N \times N$	229
9.1.3. Mnożenie liczb całkowitych	232
9.1.4. Inne znane algorytmy „dziel-i-rządź”	233
9.2. Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas	234
9.2.1. Problem plecakowy, czyli niełatwe jest życie turysty-piechura	235
9.3. Programowanie dynamiczne	238
9.4. Uwagi bibliograficzne	243
Rozdział 10 Elementy algorytmiki grafów	245
10.1. Definicje i pojęcia podstawowe	246
10.2. Sposoby reprezentacji grafów	248
10.3. Podstawowe operacje na grafach	249
10.4. Algorytm Roy-Warshalla	251
10.5. Algorytm Floyda	254
10.6. Przeszukiwanie grafów	257
10.6.1. Strategia „w głąb”	257
10.6.2. Strategia „wszerz”	259
10.7. Problem właściwego doboru	261
10.8. Podsumowanie	266
Rozdział 11 Algorytmy numeryczne	267
11.1. Poszukiwanie miejsc zerowych funkcji	268
11.2. Iteracyjne obliczanie wartości funkcji	269
11.3. Interpolacja funkcji metodą Lagrange'a	270
11.4. Różniczkowanie funkcji	272

11.5. Całkowanie funkcji metodą Simpsona.....	274
11.6. Rozwiązywanie układów równań liniowych metodą Gaussa	276
11.7. Uwagi końcowe	279
Rozdział 12 W stronę sztucznej inteligencji.....	281
12.1. Reprezentacja problemów	282
12.2. Gry dwuosobowe i drzewa gier	283
12.3. Algorytm mini-max.....	286
Rozdział 13 Kodowanie i kompresja danych.....	293
13.1. Kodowanie danych i arytmetyka dużych liczb.....	294
13.2. Kompresja danych metodą Huffmana.....	302
Rozdział 14 Zadania różne	309
14.1. Teksty zadań	309
14.2. Rozwiązań.....	312
Dodatek A Poznaj C++ w pięć minut.....	317
Literatura	337
Spis ilustracji	339
Spis tablic	343
Skorowidz	345

Przedmowa

Algorytmika stanowi gałąź wiedzy, która w ciągu ostatnich kilkudziesięciu lat dostarczyła wielu efektywnych narzędzi wspomagających rozwiązywanie różnorodnych problemów przy pomocy komputera. Książka ta prezentuje w miarę szeroki, ale i zarazem pogłębiony wachlarz tematów z tej dziedziny. Ma ona również ukazać Czytelnikowi odpowiednią perspektywę możliwych zastosowań komputerów i pozwolić mu – jeśli można użyć takiej metafory – nie wyważać drzwi tam, gdzie ktoś dawno już je otworzył.

Dla kogo jest ta książka?

Niniejszy podręcznik szczególnie polecam 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 książka dla osób, które zetknęły się już z programowaniem i rozumieją podstawowe pojęcia, takie jak *zmienna, program, algorytm, komplikacja...* – tego typu terminy będą bowiem stanowiły podstawę języka używanego w tej książce.

Co odróżnia tę książkę od innych podręczników?

Przede wszystkim – nie jest to publikacja skierowana jedynie dla informatyków. Liczba osób wykorzystujących komputer do czegoś więcej niż do gier i pisania listów jest wbrew pozorom dość duża. Zaliczyć do tego grona można niewątpliwie studentów kierunków informatycznych, ale nie tylko; w programach większości

studiów technicznych znajdują się elementy informatyki, mające na celu przygotowanie do sprawnego rozwiązywania problemów przy pomocy komputera. Nie wolno pomijać także stale rosnącej grupy ludzi zajmujących się programowaniem traktowanym jako hobby. Uwzględniając tak dużą różnorodność potencjalnych odbiorców tej publikacji duży nacisk został położony na prostotę i klarowność wykładu oraz unikanie niedomówień – oczywiście w takim stopniu, w jakim to było możliwe ze względu na ograniczoną objętość i przyjęty układ książki.

Dlaczego C++?

Niewątpliwe kilka słów wyjaśnienia należy poświęcić problemowi języka programowania, w którym są prezentowane algorytmy w książce. Wybór padł na nowoczesny i modny język C++ którego precyza zapisu i modularność przemawiają za użyciem go do programowania nowoczesnych aplikacji. 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++ zdobywa 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 mini-kurs języka C++, który powinien umożliwić im błyskawiczne opanowanie podstawowych różnic między C++ i Pascalem.

Oczywiście niemożliwe jest szczegółowe nauczenie tak obszernego pojęciowo języka, jakim jest C++, dysponując objętością zaledwie krótkiego dodatku – bo tyle zostało przeznaczone na ten cel. Zamiarem było jedynie przełamanie bariery składniowej, tak aby były zrozumiałe prezentowane listingi. Czytelnik pragnący poszerzyć zasady programowania w C++ może sięgnąć na przykład po [Pohl89], [WF92] lub [Wró94] gdzie zagadnienia te zostały poruszone szczegółowo.

Ambitnym i odważnym programistom można polecić dokładne przestudiowanie [STR92] – dzieła napisanego przez samego twórcę języka i stanowiącego ostatczną referencję na temat C++.

Jak należy czytać tę książkę?

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

Początkującym 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 przy 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 rozwiązaniem zadań w nim zamieszczonych zaleca się dokładne przestudiowanie całego materiału, który obejmują poprzedzające go rozdziały.

Ostrożność nie zawadzi...

Niemogliwe jest zaprezentowanie wszystkiego, co najważniejsze w dziedzinie algorytmiki, w objętości jednej książki. Jest to niewykonalne z uwagi na rozpiętość dziedziny, z jaką mamy do czynienia. Może się więc okazać, że to, co zostało pomyślano jako logicznie skonstruowana całość, jednych rozczaruje, innych zaś przytłoczy ogromem poruszanych zagadnień. Pragnieniem autora było stworzenie w miarę reprezentacyjnego przeglądu zagadnień algorytmicznych przydatnego dla tych Czytelników, którzy programowanie mają zamiar potraktować w sposób profesjonalny.

Co zostało opisane w tej książce?

Opis poniższy jest w pewnym sensie powtórzeniem spisu treści, jednak zawiera on coś, czego żaden spis treści nie potrafi zaoferować – minimalny komentarz dotyczący zawartości.

Rozdział 1 Zanim wystartujemy

Rozbudowany wstęp pozwalający wziąć „głęboki oddech” przed przystąpieniem do klawiatury...

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.

Rozdział 3 Analiza sprawności algorytmów

Przegląd najpopularniejszych i najprostszych metod służących do obliczania sprawności obliczeniowej algorytmów i porównywania ich ze sobą w celu wybrania „najefektywniejszego”.

Rozdział 4 Algorytmy sortowania

Prezentuje najpopularniejsze i najbardziej znane procedury sortujące.

Rozdział 5 Struktury danych

Omawia popularne struktury danych (listy, kolejki, drzewa binarne etc.) i ich implementację programową. 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 stosuje kilka poznanych już wcześniej metod 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: programowanie dynamiczne, dziel-i-rządź, algorytmy żarłoczne (ang. *greedy*). Te *meta-algorytmy* rozszerzają znacznie 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) rozwiązywanie wielu problemów, które traktowane przy pomocy innych struktur danych wydają się nie do rozwiązania.

Rozdział 11 Algorytmy numeryczne

Kilka ciekawych problemów natury obliczeniowej, ukazujących zastosowanie komputerów w matematyce, do wykonywania obliczeń przybliżonych.

Rozdział 12 Czy komputery mogą myśleć?

Wstęp do bardzo rozległej dziedziny tzw. *sztucznej inteligencji*. Przykład implementacji programowej popularnego w teorii gier algorytmu *Mini-Max*.

Rozdział 13 Kodowanie i kompresja danych

Omówienie popularnych metod kodowania i kompresji danych: systemu kryptograficznego z kluczem publicznym i metody *Huffmanna*. Rozdział zawiera ponadto dokładne omówienie sposobu wykonywania operacji arytmetycznych na bardzo dużych liczbach całkowitych.

Rozdział 14 Zadania różne

Zestaw różnorodnych zadań, które nie zmieściły się w głównej treści książki.

Wersje programów na dyskietce

Programy znajdujące się na dołączonej do książki dyskietce są zazwyczaj pełniejsze i bardziej rozbudowane. Jeśli w trakcie wykładu jest prezentowana jakaś funkcja bez podania *explicit* sposobu jej użycia, to na pewno dyskietkowa wersja zawiera reprezentacyjny przykład jej zastosowania (przykładowa funkcja *main* i komplet funkcji nagłówkowych). Warto zatem podczas lektury porównywać wersje dyskietkowe z tymi, które zostały omówione na kartach książki!

Pliki na dyskietce są w formacie MS-DOS. Programy zostały przetestowane zarówno systemie DOS (kompilator Borland C++), jak i w systemie UNIX (kompilator GNU C++).

Na dyskietce znajdują się zatem pełne wersje programów, które z założenia powinny dać się od razu uruchomić na dowolnym kompilatorze C++ (UNIX lub

DOS/Windows). Jedyny wyjątek stanowią programy „graficzne” napisane dla popularnej serii kompilatorów firmy Borland; wszelkie inicjacje trybów graficznych itp. są tam wykonane według standardu tej firmy.

W tekście znajduje się jednak tabelka wyjaśniająca działanie użytych instrukcji graficznych, tak więc nawet osoby, które nigdy nie pracowały z kompilatorami Borlanda, poradzą sobie bez problemu z analizą programów przykładowych.

Konwencje typograficzne i oznaczenia

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

W szczególności regułą jest, że wszystkie listingi i teksty ukazujące się na ekranie zostały odróżnione od zasadniczej treści książki czcionką Courier:

prog.cpp

Tekst programu znajduje się na dyskietce w pliku *prog.cpp*

Inna konwencja dotyczy odnośników bibliograficznych:

Patrz [Odn93] – odnośnik do pozycji bibliograficznej [Odn93] ze spisu na końcu książki.

Uwagi na marginesie

Książka ta powstała w trakcie 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, jest ciągle w Polsce dość trudno (jeśli w ogóle) dostępna i będzie dla mnie dużą radością, jeśli znajdą się osoby, którym niniejszy podręcznik oszczędzi w jakiś sposób czasu i pieniędzy.

Wstępny wydruk (jeszcze w „pieluchach”) książki został przejrzany i opatrzony wieloma cennymi uwagami przez Zbyszka Chamskiego. Ostateczna wersja książki została poprawiona pod względem poprawności językowej przez moją siostrę, Ilonę. Chciałbym gorąco podziękować im obojgu za wykonaną pracę, licząc jednocześnie, że efekt końcowy ich zbytnio nie zawiódł...

P.W.

Lannion

Wrzesień 1995

Uwagi do wydania 2

W bieżącej edycji książki, wraz z całym tekstem zostały gruntownie przejrzone i poprawione programy przykładowe, jak również rysunki znajdujące się w tekście, które w pierwszym wydaniu zawierały kilka niekonsekwencji. Została zwiększoną czytelność listingów (wyróżnienie słów kluczowych), oraz dołożono trzy nowe rozdziały (*II – IV*). Uzupełnieniu uległy ponadto rozdziały: *10* (gdzie omówione zostało dodatkowo m.in. przeszukiwanie grafów) i *5* (omówiono implementację zbiorów). Licząc, że wniesione poprawki odbiją się pozytywnie na jakości publikacji, życzę przyjemnej i pożytecznej lektury.

P.W.

Czerwiec 1997

Rozdział 1

Zanim wystartujemy

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

ALGORYTM¹:

- skończony ciąg/sekwencja reguł, które aplikuje się na skończonej liczbie danych, pozwalający rozwiązywać zblizone 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 specjalisci zajmujący się historią matematyki odnaleźli najbardziej prawdopodobny źródłosłów: termin ten pochodzi od nazwiska perskiego pisarza-matematyka Abu Ja'far Mohammed ibn Mūsâ al-Khowârizmî² (IX wieku n.e.). Jego zasługą jest dostarczenie klarownych reguł wyjaśniających krok po kroku zasady operacji arytmetycznych wykonywanych na liczbach dziesiętnych.

Słowo algorytm często jest 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 ;

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*.

```

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 oczywiście jedyny algorytm, z którym mieliśmy w swoim życiu do czynienia. Każdy z nas z pewnością umie zaparzyć kawę:

- 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 polecienniem „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:

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 $T(A)$).

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 pośpieszmy do klawiatury, aby wpisywać do pamięci komputera programy spełniające powyższe założenia, popatrzmy na algorytmikę z perspektywy historycznej.

1.1. Jak to wcześniej było, 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 związane z rewolucją techniczną dokonującą się na przestrzeni zaledwie ostatnich dwustu lat. Patrzmy zresztą na kilka charakterystycznych dat z tego okresu:

– 1801 –

Francuz Joseph Marie Jacquard wynajduje krosno tkackie, w którym wzorzec tkaniny był „programowany” na swego rodzaju kartach perforowanych. Proces tkania był kontrolowany przez algorytm zakodowany w postaci sekwencji otworów wybitych w karcie.

– 1833 –

Anglik Charles Babbage częściowo buduje maszynę do wyliczania niektórych formuł matematycznych. Autor koncepcji tzw. *maszyny analitycznej*, zbliżonej do swego poprzedniego dzieła, ale wyposażonej w możliwość przeprogramowywania, jak w przypadku maszyny Jacquarda.

– 1890 –

Pierwsze w zasadzie 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 przedsiębiorstwo Holleritha przekształciło się w 1911 roku w International Business Machines Corp., bardziej znane jako IBM).

– lata 30-te –

Rozwój badań nad teorią algorytmów (plejada znanych matematyków: Turing, Gödel, Markow).

– lata 40-te –

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 I skonstruowany w 1944 roku (jeszcze na przekaźnikach, czyli jako urządzenie elektro-mechaniczne). Jego twórcą był Amerykanin Howard Aiken z uniwersytetu Harvard. Aiken bazował na idei Babbage'a, która musiała czekać 100 lat na swoją praktyczną realizację! W dwa lata później powstaje pierwszy „elektroniczny” komputer ENIAC (jego wynalazcy: J. P. Eckert i J. W. Mauchly z uniwersytetu Pensylwania).

Powszechnie jednak za „naprawdę” pierwszy komputer w pełnym tego słowa znaczeniu uważa się EDVAC zbudowany na uniwersytecie w Princeton. Jego wyjątkowość polegała na umieszczeniu programu wykonywanego przez komputer całkowicie w pamięci komputera. Autorem tej przełomowej idei był matematyk Johannes von Neumann (Amerykanin węgierskiego pochodzenia).

– okres powojenny –

Prace nad komputerami prowadzone są w wielu krajach równolegle. W grę zaczyna wchodzić wejście na obiecujący nowo powstały rynek komputerów (kończy się bowiem era budowania unikalnych „uniwersyteckich” prototypów). Na rynku pojawiają się kalkulatory IBM 604 i BULL Gamma3, 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!

– 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.

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

Zamieszczony w poprzednim paragrafie „kalendarz” został doprowadzony 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 każdym praktycznie biurze i w coraz większej ilości domów prywatnych.

W latach 60-tych zaczęto konstruować pierwsze naprawdę duże systemy informatyczne – w sensie ilości kodu, głównie asemblerowego, 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 były tworzone szybko 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”¹. 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 nawet mówić o kryzysie w rozwoju oprogramowania! W roku 1968 została nawet zwołana konferencja NATO (Garmisch, Niemcy) poświęcona na przedyskutowanie zaistniałej sytuacji. W rok później została utworzona w ramach IFIP (International Federation for Information Processing) 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: „w miejsce 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 metodologii

¹ Źargonowe określenie procesu usuwania błędów z programu.

programowania nie znikły bynajmniej z horyzontu: Dijkstra, Hoare, Floyd, Wirth... (Będą oni jeszcze nie raz cytowani w tej książce!).

Krótką prezentację, 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 warto 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 tyczy się również wielu innych dziedzin wiedzy).

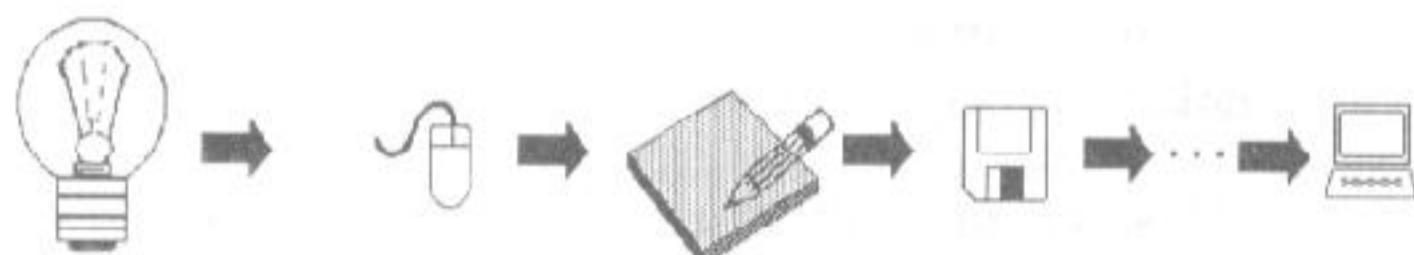
Teraz, gdy już zdefiniowaliśmy sobie głównego bohatera tej książki (bohatera zbiorowego: chodzi bowiem o algorytmy!), przejrzymy kilka sposobów używanych do jego opisu.

1.3. Proces koncepcji programów

W paragrafie poprzednim wyszczególniliśmy kilka cech charakterystycznych, które powinien posiadać algorytm rozumiany jako pojęcie informatyczne. Szczerogólny 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 obrazuje w sposób uproszczony etapy procesu programowania komputerów. Olbrzymia żarówka symbolizuje etap, który jest od czasu do czasu pomijany przez programistów (dodajmy, że typowo z opłakanymi skutkami...) – REFLEKSJĘ.

Rys. 1 - 1.
Etapy konstrukcji programu.



Następnie jest tworzony tzw. tekst źródłowy nowego programu, mający postać pliku tekstowego, wprowadzanego do komputera przy pomocy 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 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 języka¹. Kompilator dokonuje mniej lub bardziej zaawansowanej analizy poprawności i, jeśli wszystko jest w porządku, produkuje tzw. *kod wykonywalny*, zapisany w postaci zrozumiałej przez komputer. Plik zawierający kod wykonywalny może być następnie wykonywany pod kontrolą systemu operacyjnego komputera (który *notabene* także jest zbiorem programów).

Gdzie w tym procesie umiejscowione jest to, co stanowi tematykę książki, którą trzymasz, Czytelniku, w ręku? Otóż z całego skomplikowanego procesu tworzenia oprogramowania zajmiemy 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 rozwiązywać przy pomocy komputera, a następnie omówimy sobie, jak to zadanie wykonać w sposób efektywny. Tworzenie zewnętrznej otoczki programów, czyli tzw. *interfejsu użytkownika* jest w chwili obecnej procesem praktycznie do końca zautomatyzowanym, co wyklucza konieczność poruszania tego tematu w książce.

1.4. 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 asemblera: nieczytelny dla nieprzygotowanego odbiorcy);
- zbliżyć się do człowieka (opis słowny: maksymalny poziom abstrakcji zakładający poziom inteligencji odbiorcy niemożliwy aktualnie do „wbudowania” w maszynę²).

Wybór języka asemblera 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 przez komputer, może nie zadziałać!

¹ W praktyce jest to język angielski.

² Niemowlę radzi sobie bez problemu z problemami, nad którymi biedzą się specjalisci od tzw. *sztucznej inteligencji* usiłujący je rozwiązywać przy pomocy komputerów! (Chodzi o efektywność uczenia się, rozpoznawanie form etc.).

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

- przy pomocy 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 którejś z nich zostanie podyktowany kontekstem omawianych zagadnień. Przykładowo, jeśli dany algorytm jest możliwy do czytelnej prezentacji przy pomocy języka programowania, wybór będzie oczywisty! Od czasu do czasu jednak napotkamy na 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 obstawały za tym wyborem. C++ jest językiem programowania określonym 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 obiektywne cechy tego języka powodują, iż bardzo łatwe jest ukrywanie szczegółów implementacyjnych, rozszerzanie już zdefiniowanych modułów (bez ich kosztownego „przepisywania”), a są to właściwości, którymi nie można pogardzić.

Być może cenne będzie podkreślenie „usługowej” roli, jaką w procesie programowania pełni język do tego celu wybrany. 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 śmiesznym...). Język programowania jest w końcu tylko narzędziem, ulegającym zresztą znacznej (r)ewolucji na przestrzeni ostatnich lat. Pracując nad pewnymi partiami tej książki musiałem zwalczać od czasu do czasu 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.

Zdając sobie sprawę, że C++ może być pewnej grupie Czytelników nieznany, został w dodatku A przygotowany mini-kurs 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++, umożliwia jednak lekturę książki osobom pragnącym z niej skorzystać bez konieczności poznawania nowego języka.

1.5. 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 programów.

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... Mamy tu do czynienia z pośpiechem mającym na celu wyprzedzenie konkurencji, co usprawiedliwia wypuszczanie przez dyrekcje firm niedopracowanych produktów – ze szkodą dla użytkowników, którzy nie mają 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 naukowych¹!

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 ilość 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

¹ Formalne badanie poprawności systemów algorytmicznych jest możliwe przy użyciu specjalnych języków stworzonych do tego celu.

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”... 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.

Dwa typowe sposoby stosowane w praktyce to:

- sprawdzanie stanu punktów kontrolnych przy pomocy *debuggera* (odczytujemy wartości pewnych „ważnych” zmiennych i sprawdzamy, czy zachowują się „poprawnie” dla pewnych „reprezentacyjnych” danych wejściowych²).
- 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 przy pomocy 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 Dijkstra³, 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 umożliwiają *quasi-automatyczne* generowanie algorytmów.

Każdy program „wyprodukowany” przy pomocy tych metod jest automatycznie poprawny – pod warunkiem, że nie został „po drodze” popchniony jakiś błąd. „Wygenerowanie” algorytmu jest możliwa dopiero po jego poprawnym zapisaniu wg schematu:

² 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.

³ Jeśli już jesteśmy przy nim, to warto polecić przynajmniej pobiczoną lekturę [DF89], która stanowi dość dobry wstęp do metodologii programowania.

{warunki wstępne⁴} poszukiwany-program {warunki końcowe}

Możliwe jest przy pewnej dozie doświadczenia 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” pożądanych 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.

⁴ 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 rekursją¹. Mimo iż użycie rekurencji nie jest obowiązkowe², jej zalety są oczywiste dla każdego, kto choć raz spróbował tego stylu programowania. Wbrew pozorom nie jest to wcale mechanizm prosty i wiele jego aspektów wymaga dogłębnej analizy.

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.

2.1. Definicja rekurencji

Pojęcie rekurencji poznamy na przykładzie. Wyobraźmy sobie małe dziecko w wieku lat – przykładowo – pięciu. Dostaje ono od rodziców zadanie zebrania do pudełka wszystkich drewnianych klocków, które „nierozmyślnie” zostały rozsypane na podłodze. Klocki są bardzo prymitywne, są to zwyczajne drewniane sześcianiki, które doskonale nadają się do budowania nieskomplikowanych budowli. Polecenie jest bardzo proste: „Zbierz to wszystko razem i poukładaj tak jak było w pudełku”. Problem wyrażony w ten sposób jest dla dziecka

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

² Programy zapisane w formie rekurencyjnej mogą być przekształcone – z mniejszym lub większym wysiłkiem – na postać klasyczną, zwaną dalej iteracyjną (patrz rozdział 6).

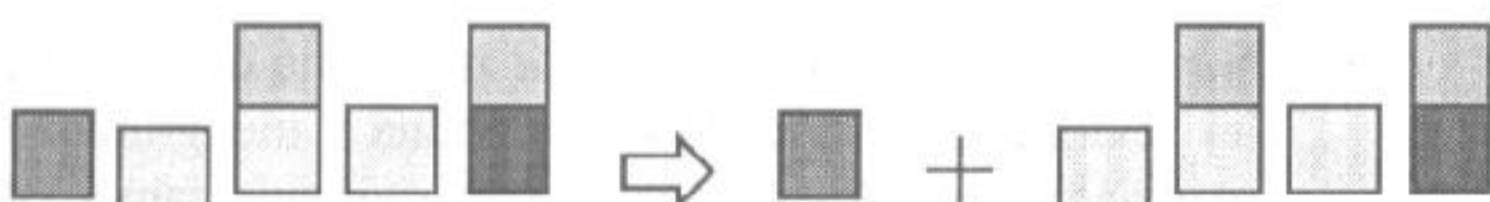
potwornie skomplikowany: klocków jest cała masa i niespecjalnie wiadomo jak się do tego całościowo zabrać. Mimo ograniczonych umiejętności na pewno nie przerasta go następująca czynność: *wziąć jeden klocek z podłogi i włożyć do pudełka*. Małe dziecko zamiast przejmować się złożonością problemu, której 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 minuty.

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 do momentu wyczerpania się klocków...

Spójrzmy na rysunek 2 - 1, który przedstawia w sposób symboliczny tok rozumowania przyjęty przy rozwiązywaniu problemu „sprzątania rozsypanych klocków”.

Rys. 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 cechami, typowymi dla algorytmów rekurencyjnych:

- zakończenie algorytmu jest jasno określone („w momencie gdy na podłodze nie będzie więcej klocków, możesz uznać, że zadanie zostało wykonane”).
- „duży” problem został rozłożony na problem elementarny (który umiejsmy rozwiązać) i na problem 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 opisany powyżej problemie w kategorii algorytmu? Czy w ogóle możemy przypisywać pięcioletniemu dziecku wiedzę, z której ono nie zdaje sobie 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 rozpaczą 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. Pójdzmy dalej za tym śmiałyim 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 w pełni uświadomimy sobie mechanizmy w nich wykorzystywane.

Powyższe wyjaśnienie pojęcia rekurencji powinno być znacznie czytelniejsze niż typowe podejście zatrzymujące się na niewiele mówiącym stwierdzeniu, że „program rekurencyjny jest to program, który wywołuje sam siebie”...

2.2. Ilustracja pojęcia rekurencji

Program, którego analizą będziemy się zajmowali w tym podrozdziale, jest bardzo zbliżony do problemu klocków, z którym spotkaliśmy się przed chwilą. Schemat rekurencyjny zastosowany w nim jest identyczny, jedynie zagadnieniem jest nicco 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 wysoko prawdopodobne, że rozumowałoby ono w sposób następujący:

- Wziąć pierwszy niezbadany element tablicy n -elementowej;
- jeśli aktualnie analizowany element tablicy jest równy x , to:
wypisz „Sukces” i zakończ;

w przeciwnym wypadku

Zbadaj pozostałą część tablicy $n-1$ -elementowej.

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

```
const      n=10;
int       tab[n]={1,2,3,2,-7,44,5,1,0,-3};
void szukaj(int tab[n], 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);
}
```

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

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

Podstawowymi błędami popełnianymi przy konstruowaniu programów rekurencyjnych są:

- 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.

2.3. 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 bardziej od podszewki JAK to działa!”. Pozostaje zatem przyporządkować 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 podstawowej¹!) na lekcjach matematyki dość często używa się tzw. *silni* z n , czyli iloczynu wszystkich liczb naturalnych od 1 do n włącznie. Ten użyteczny symbol¹ zdefiniowany jest w sposób następujący:

$$\begin{aligned}0! &= 1, \\ n! &= n \cdot (n-1)! \text{ gdzie } n \geq 1\end{aligned}$$

Pomińmy jego znaczenie matematyczne, nieistotne w tym miejscu. Nic nie stoi jednak 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);
}
```

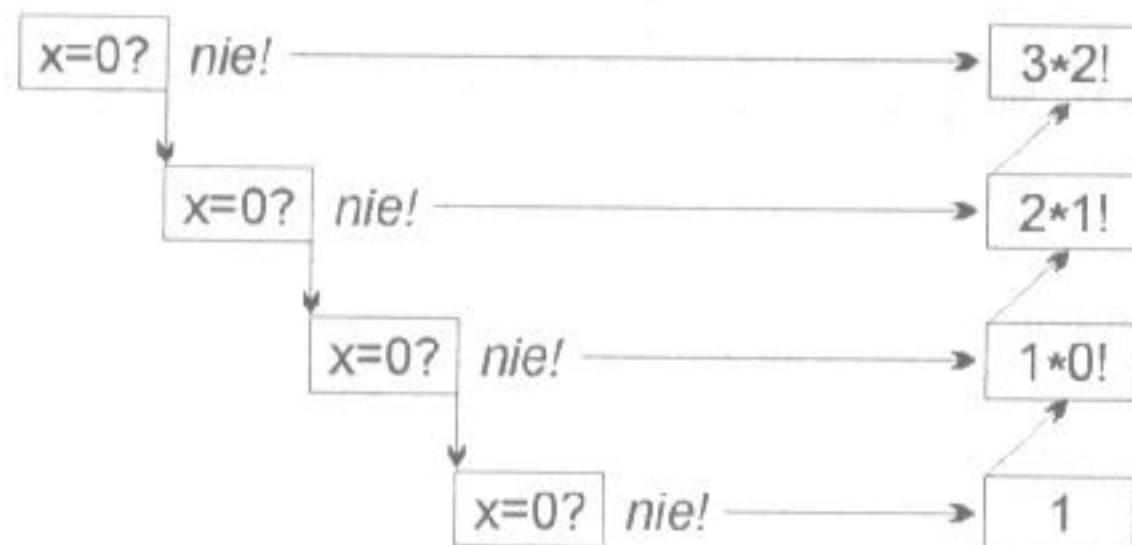
Prześledźmy na przykładzie, jak się wykonuje program, który obliczy $3!$ Rysunek 2 - 2 przedstawia kolejne etapy wywoływania procedury rekurencyjnej i badanie warunku na przypadek elementarny.

Konwencje użyte podczas tworzenia 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.

¹ Oznaczany przez $n!$

Rys. 2 - 2.
Drzewo wywołań
funkcji *silnia(3)*



Czymże są jednak owe tajemnicze *poziomy*, *przekazywanie parametrów*, etc.? Chwilowo te pojęcia mają prawo brzmieć z lekka egzotycznie. Aby zmienić to wrażenie, opiszemy słownic sposob obliczenia *silnia(2)*:

Funkcja *silnia* otrzymuje liczbę 2 jako parametr wywołania i analizuje: „czy 2 równa się 0?”. Odpowiedź brzmi „*Nie*”, zatem funkcja „przyjmuje”, że jej wynikiem jest $2 \cdot \text{silnia}(1)$.

Niestety, wartość *silnia(1)* jest nieznana... Funkcja wywołuje zatem kolejny swój egzemplarz, który zajmie się obliczeniem wartości *silnia(1)*, wstrzymując jednocześnie skalkulowanie wyrażenia $2 \cdot \text{silnia}(1)$. Po tym wywołaniu rekurencyjnym funkcja *silnia* czeka na wynik częściowy, który zostanie „nadesłany” przez jej wywołany niedawno nowy „egzemplarz”.

W praktyce przekazywanie parametrów odbywa się za pośrednictwem stosu, programista jednak ma prawo zupełnie się tym nie przejmować. Fakt, iż parametr zostanie zwrócony za pośrednictwem stosu, niewiele się bowiem różni od przesykania 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 częściowego) wypadałoby jakoś je różnicować. Najprostszą metodą jest dokonywanie tego poprzez wartość *x*, która jest dla nas punktem odniesienia używanym przy określaniu aktualnej „głębokości” rekurencji.

2.4. Niebezpieczeństwa rekurencji

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

2.4.1. Ciąg Fibonacciego

Naszym pierwszym zadaniem jest napisanie programu, który liczyłby elementy tzw. ciągu Fibonacciego. Ten dziwoląg matematyczny, używany do wielu różnych i czasami zaskakujących celów, jest definiowany następująco:

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

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

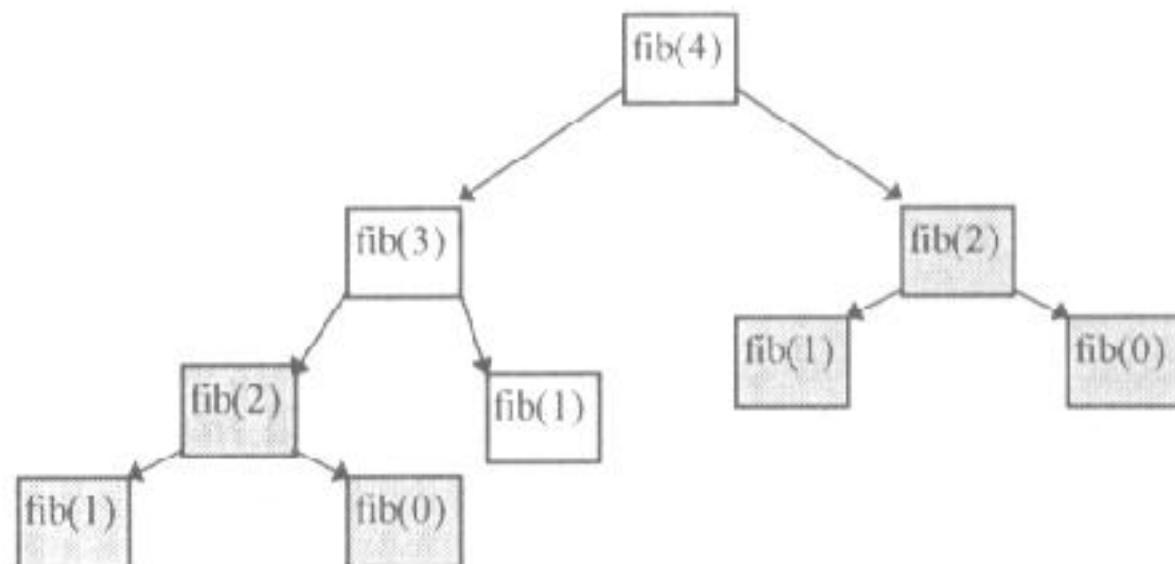
rek3.cpp

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

Spróbujmy prześledzić dokładnie wywołania rekurencyjne. Nieskomplikowana analiza prowadzi do następującego drzewa:

Rys. 2 - 3.

Obliczanie $fib(4)$.



Każde „zaciemnione” wyrażenie stanowi problem elementarny; problem o rozmiarze $n \geq 2$ zostaje „rozbity” na dwa problemy o mniejszym stopniu skomplikowania: $n-1$ i $n-2$.

Skąd się jednak wziął pesymistyczny tytuł tego podrozdziału? Przypatrzymy 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 „zauważać”, w końcu jest to tylko program, który wykonuje to, co mu

¹ Jeśli można sobie pozwolić na tego typu personifikację...

każemy. W rozdziale 9 zostanie omówiona ciekawa technika programowania (tzw. *programowanie dynamiczne*) pozwalająca poradzić sobie z powyższą wadą.

2.4.2. 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 „zawiesić” komputer (przez co rozumiemy powszechnie stan, w którym program nie reaguje na nic i trzeba mu zasalutować trzema klawiszami²) naszym programem? Zdarza się to nawet najbardziej uważnym programistom i stanowi raczej nieodłączny element pracy programistycznej...

Istnieje kilka typowych przyczyn „zawieszania” 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 informacji³ niezbędnych do odtworzenia stanu przed wywołaniem, 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 funkcja MacCarthy’ego, zaprezentowana poniżej:

rek4.cpp

```
unsigned long int MacCarthy(int x)
{
    if (x>100)
```

² Ctrl-ALT-Del w systemie DOS, instrukcja *kill* w systemie Unix...

³ W szczególności wnikać nie będziemy, gdyż tematyka ta nie ma dla nas większego znaczenia w tym miejscu.

```
    return (x-10);
else
    return MacCarthy(MacCarthy(x+11));
}
```

Już na pierwszy nawet rzut oka widać, że funkcja jest jakąś „dziwną”. Kto potrafi powiedzieć w przybliżeniu, jak się przedstawia jej ilość 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...

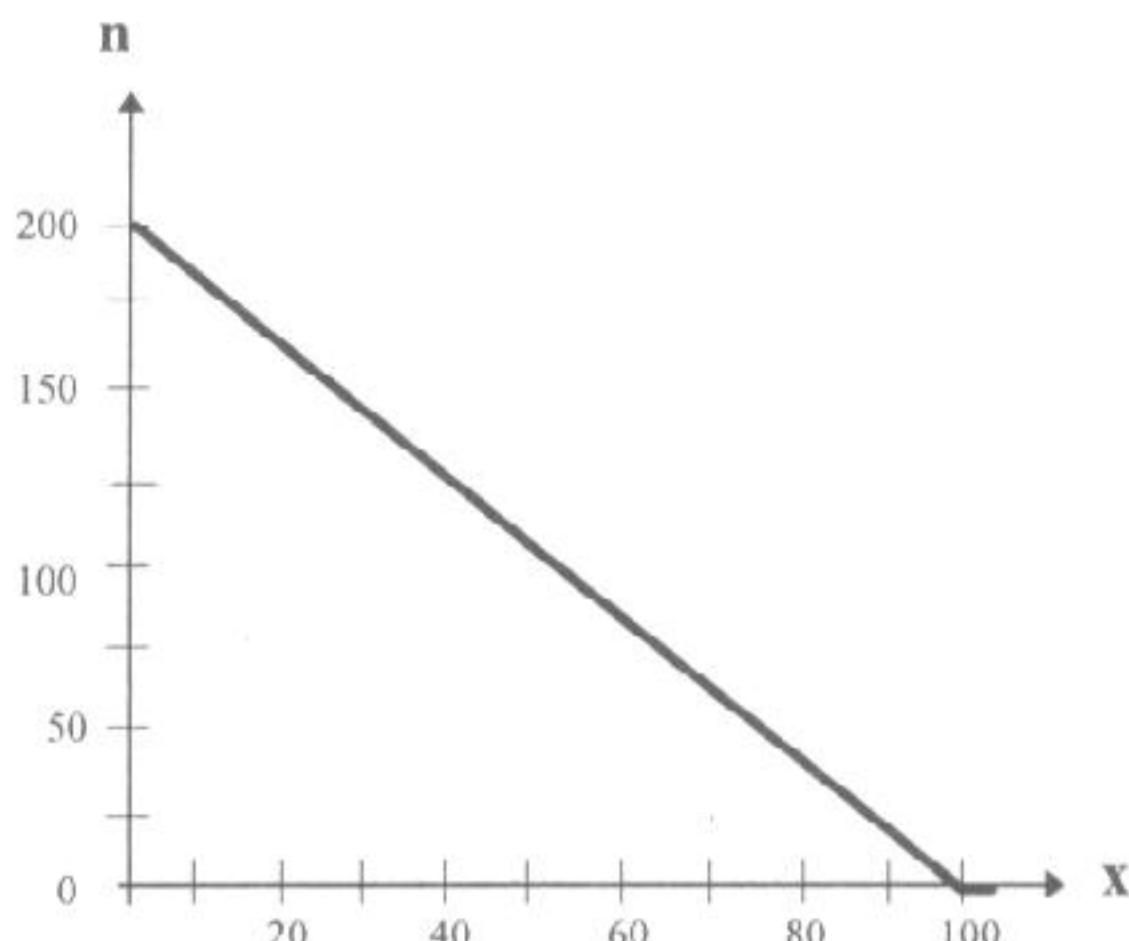
Nie było to wcale takie oczywiste, prawda?

Ćwicz. 2-1

Proszę dokładnie zbadać funkcję MacCarthy'ego w większym przedziale liczbowym, niż ten na rysunku. Jakich niebezpieczeństw można się doszukać?

Rys. 2 - 4.

Ilość wywołań
funkcji Mac-
Carthy'ego
w zależności od
parametru
wywołania.



2.5. 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...

2.5.1. Stąd do wieczności

W wielu funkcjach rekurencyjnych, pozornie dobrze skonstruowanych, może z łatwością ukryć się błąd polegający na spowokowaniu nieskończonej ilości wywołań rekurencyjnych. Taki właśnie zwodniczy przykład jest przedstawiony poniżej:

std.cpp

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

Gdzie jest umiejscowiony problem? Patrząc na ten program trudno dopatrzyć 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 zauważać, 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.

2.5.2. 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 zapętli się, będzie dostarczać oczekiwane wyniki etc.). Niestety jest to wiara dość naiwna i niczym nie uzasadniona. Matematyk bowiem jest w stanie zrobić wszystko związane ze „swoją” 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:

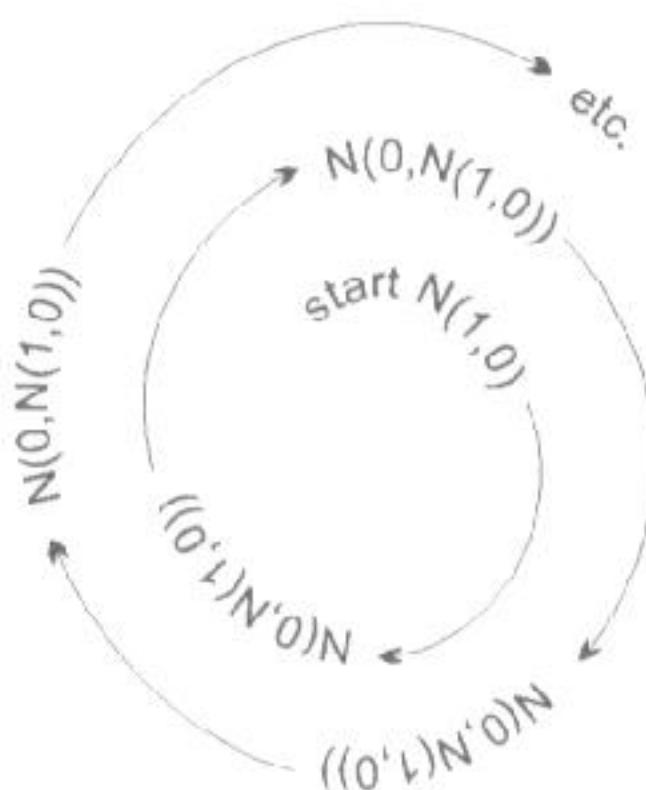
```
int N(int n, int p)
{
    if (n==0)
        return 1;
    else
        return N(n-1, N(n-p, p));
}
```

Można przeprowadzić dowód matematyczny¹, 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++?

Otoż regułą w jego przypadku jest to, iż wszystkie parametry funkcji rekurencyjnej są ewaluowane 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; zobaczymy, jak to się odbędzie w przypadku naszej funkcji, np. dla $N(1,0)$ (patrz rysunek 2 - 5).

Rys. 2 - 5.
Nieskończony ciąg
wywołań rekuren-
cyjnych.



¹ Patrz [Kro89].

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.

2.6. 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 linii kodu użytego na ich realizację. Z tym ostatnim jest ściśle związana łatwość odnajdywania ewentualnych błędów. Wróćmy jednak do tematu.

Zauważaliśmy 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. Podzielimy metody rekurencyjne, tytułem uproszczenia, na dwa podstawowe typy:

- rekurencja „naturalna”;
- rekurencja „z parametrem dodatkowym”¹.

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

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

rek5.cpp

```
unsigned long int silnia1(unsigned long int x)
{
```

¹ Pozostaniemy na moment przy tej nieprecyzyjnej nazwie; ten typ rekurencji powróci nam jeszcze w rozdziale 6 – w innym jednakże kontekście.

```
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 funkcji nie jest być może oczywiste, ale wystarczy wziąć kartkę i ołówek, aby przekonać się na kilku przykładach, że wykonuje ona swoje zadanie. Osobom nie znającym dobrze C++ należy się niewątpliwie wyjaśnić konstrukcji funkcji *silnia2*. Otóż dowolna funkcja w C++ może posiadać parametry domyślne. Dzięki temu funkcja o nagłówku:

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

może być wywołana na dwa sposoby:

- określając wartość drugiego parametru, np. *FunDom(12,5)*: w tym przypadku *k* przyjmuje wartość 5;
- nie określając 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. Jednak jakie istotne wzgłydy 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 nadanie 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 domyśla się już zapewne, że program rekurencyjny „z parametrem dodatkowym” robi to wszystko nieco wydajniej. Ponieważ parametr dodatkowy służy do przekazywania elementów wyniku końcowego, dysponując nim 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!

2.7. 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...).

2.7.1. Spirala

Zastanówmy się, jak można narysować rekurencyjnie jednym „pociągnięciem” kreski rysunek 2 - 6.

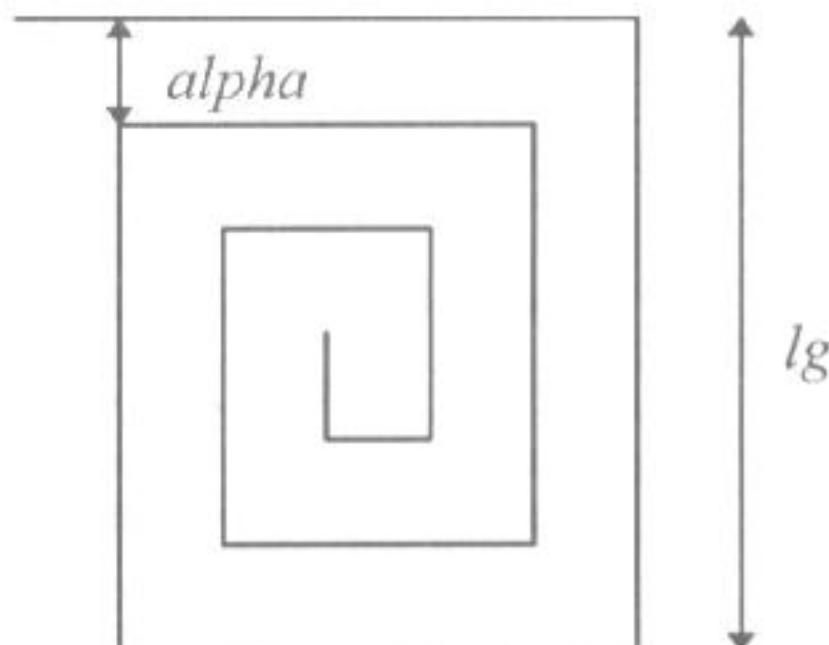
Parametrami programu są :

- odstęp pomiędzy liniami równoległymi: *alpha*;
- długość boku rysowanego w pierwszej kolejności: *lg*.

Algorytm iteracyjny byłby również nieskomplikowany (zwykłą pętlą), ale założmy, że zapomnimy 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.

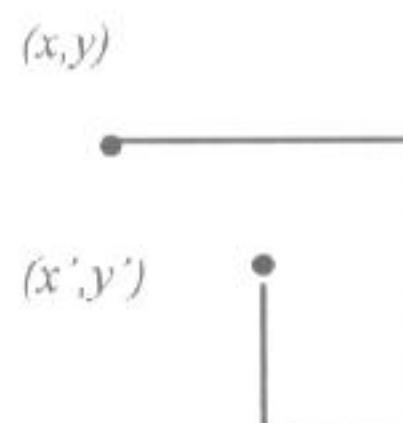
Rys. 2 - 6.
Spirala narysowana rekurencyjnie.



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, obarczoną oczywiście pewnymi warunkami gwarantującymi jej poprawne zakończenie.

Elementarny przypadek rozwiązania prezentuje rysunek 2 - 7.

Rys. 2 - 7.
Spirala narysowana rekurencyjnie – szkie rozwiązania.



Jedna z kilku możliwych wersji programu, który realizuje to, co zostało wyżej opisane, jest przedstawiona poniżej.

W celu ułatwienia lektury programu zamieszczone zostały również objaśnienia instrukcji graficznych.

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);
    }
}

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

```

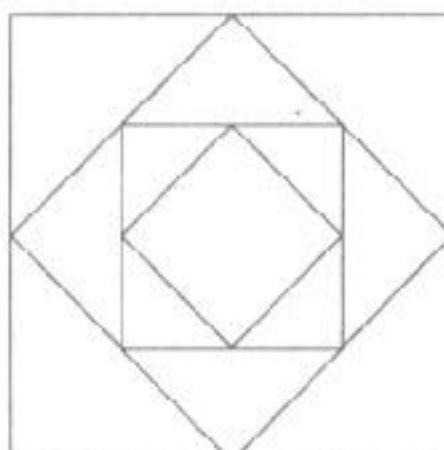
Tabela 2 - 1.
Objaśnienia
instrukcji
graficznych.

FUNKCJA	ZASTOSOWANIE
<i>lineto(x,y)</i>	kreśli odcinek prostej od pozycji bieżącej do punktu (x, y)
<i>moveto(x,y)</i>	przesuwa kurSOR graFicznY do punktu (x, y)
<i>getmaxx()</i>	zwraca maksymalną współrzędną poziomą (zależy o rozdzielcości trybu graficznego)
<i>getmaxy()</i>	zwraca maksymalną współrzędną pionową (j. w.)
<i>getx()</i>	zwraca aktualną współrzędną poziomą
<i>gety()</i>	zwraca aktualną współrzędną pionową

2.7.2. Kwadraty „parzyste”

Zadanie jest podobne do poprzedniego: jak jednym pociągnięciem kreski narysować figurę przedstawioną na rysunku 2 - 8?

Rys. 2 - 8.
Kwadraty
„parzyste” ($n=2$).



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 rekurencyjnego:

kwadraty.cpp

```
void kwadraty(int n, double lg, double x, double y)
{
    // n = parzysta ilość kwadratów
    // x,y = punkt startowy
    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);
        lineto(x, y);
    }
}

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

2.8. 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. 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, nic 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 i/lub szybki program iteracyjny;
- algorytm rekurencyjny jest *niestabilny* (np. dla pewnych wartości parametrów wejściowych może się zapętlić lub dawać „dziwne” wyniki).

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 nieobliczalnego.

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

2.9. 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 wzgłydy praktyczne i prostota.

Zad. 2-1

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

Zad. 2-2

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:

- $t[0], t[1] \dots t[n/2-1], t[n/2], t[n/2+1] \dots t[n-1]$
- jeśli $x=t[n/2]$, to element x został znaleziony¹;
- 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.

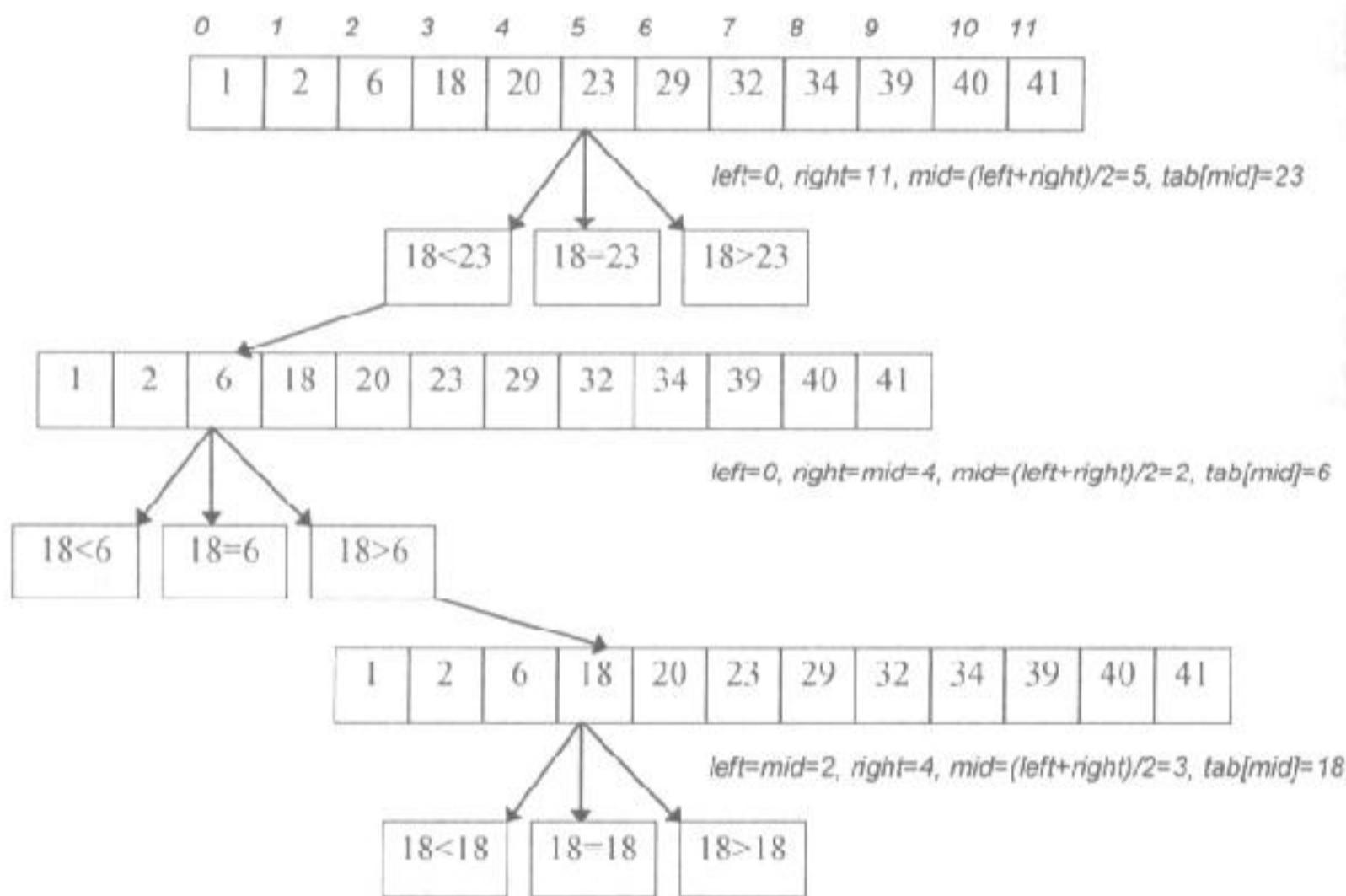
¹ W C++ dzielenie całkowite „obcina” wynik do liczby całkowitej (odpowiednik *div* w Pascalu).

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;

Rys. 2 - 9.

Przeszukiwanie binarne na przykładzie.



- **mid** indeks elementu środkowego obserwowanego aktualnie obszaru tablicy.

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 etapach². Warto zauważyć, że to samo zadanie, rozwiązywane za pomocą przeglądania od lewej do prawej elementów tablicy, zostało 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.

² Za „etap” będziemy tu uważali moment testowania, czy dana liczba jest ta, której poszukujemy.

Zad. 2-3

Napisać 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:

$$\begin{aligned}13 : 2 &= 6 + 1, \\6 : 2 &= 3 + 0, \\3 : 2 &= 1 + 1, \\1 : 2 &= 0 + 1,\end{aligned}$$

$0 \Rightarrow$ koniec algorytmu.

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:

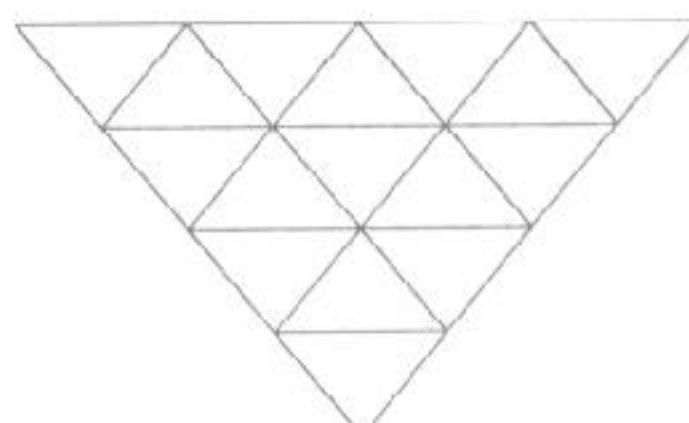
Pyt. 1 Jak wykorzystać rekurencję do odwrócenia kolejności wypisywania cyfr?

Pyt. 2 Czy istnieje łatwe rozwiązanie tego zadania, wykorzystujące rekurencję z „parametrem dodatkowym”?

Zad. 2-4

Spróbuj napisać funkcję, która wymalowuje rekurencyjnie „dywanik” przedstawiony na rysunku 2-10:

Rys. 2-10.
Trójkąty narysowane rekurencyjnie.



2.10. Rozwiązania i wskazówki do zadań

Zad. 2-1

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
    }
}

void main()
{
    int tabl[8]={1,2,3,4,5,6,7,8};
    odwroc(tabl,0,7);           // przykładowe wywołanie
    for(int i=0;i<8;i++)
        cout << tabl[i];       // sprawdzamy efekt...
}
```

Zad. 2-2

Poniżej przedstawiona jest wyłącznie wersja rekurencyjna programu. Jestem przekonany, że Czytelnik odkryje bez trudu analogiczne rozwiązanie iteracyjne¹:

binary_s.cpp

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

¹ Lub zajrzy do rozdziału 7...

Zad. 2-3

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 przeanalizujmy raz jeszcze działanie algorytmu bazującego na dzieleniu przez podstawę systemu liczbowego (tutaj 2). Liczba x jest dzielona przez dwa, co daje nam liczbę $\lfloor x \text{ div } 2 \rfloor$ plus reszta. Owa reszta to oczywiście $\lfloor x \text{ mod } 2 \rfloor$ 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 rozpisać tworzenie reprezentacji dwójkowej pewnej liczby x , używając już właściwych dla C++ operatorów:

$$[x]_2 = [x \% 2][x / 2]_2$$

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

$$[x]_2 = \begin{array}{l} \text{przypadek elementarny} \\ \overbrace{[x \% 2]}^{\text{wywołanie rekurencyjne}} \end{array} \quad \begin{array}{l} \text{wywołanie rekurencyjne} \\ \overbrace{[x / 2]_2}^{\text{przypadek elementarny}} \end{array}$$

Jeśli w powyższym algorytmie każemy komputerowi wpierw napisać liczbę $\lfloor x / 2 \rfloor$ dwójkowo, a dopiero potem $\lfloor x \% 2 \rfloor$ (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 więcej nauczają niż lektura gotowych rozwiązań.

Zad. 2-4

Oto jedno z możliwych rozwiązań:

```
trojkaty.cpp
```

```
void trojkaty(double n, double lg, double x, double y)
{
    // n = ilość 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);
    }
}

void main()
{
    // inicjuj tryb graficzny
    moveto(getmaxx()/2, getmaxy()-10);
    trojkaty(6, getmaxx()/2, getx(), gety());
    getch();
    // zamknij tryb graficzny
}
```

Rozdział 3

Analiza sprawności algorytmów

Podstawowe kryteria pozwalające na wybór właściwego algorytmu zależą głównie od kontekstu, w jakim zamierzamy go używać. Jeśli chodzi nam o społeczne używanie 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. Ktoś z „zewnętrz”, dostający do ręki dyskietkę z programem w postaci wynikowej (tzn. jako plik binarny), jest w niktym stopniu – jeśli w ogóle! – zainteresowany estetyką „wewnętrzną” programu, klarownością 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... Taki punkt widzenia jest często spotykany i programista, który zapomni go uwzględnić, ryzykuje wylotowanie 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 swój produkt *najtaniej* i sprzedać go jak *najdrożej*, natomiast drugi chciałby za niewielką sumę dostać coś *najwyższej jakości*...

Upraszczając dla potrzeb naszej dyskusji wyżej zaanonsowaną problematykę, możemy wyróżnić dwa podstawowe kryteria oceny programu. Są to:

- sposób komunikacji z operatorem;
- szybkość wykonywania podstawowych funkcji programu.

W rozdziale tym zajmiemy się wyłącznie aspektem sprawnościowym wykonywania programów, problem komunikacji – jako zbyt obszerny – zostawiając może na inną okazję.

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. Głównym kryterium doboru zaprezentowanych narzędzi matematycznych była ich prostota. Nie każdy programista jest matematykiem i zamiennianie tej książki w podręcznik analizy matematycznej nie było bynajmniej celem autora.

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

3.1. Dobre samopoczucie użytkownika programu

Zanurzając się w problematykę analizy sprawnościowej programów, możemy wyróżnić min. 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*¹).

Z uwagi na znaczne potanie pamięci RAM w ostatnich latach to drugie kryterium straciło już praktycznie na znaczeniu². Co innego jest z pierwszym!

¹ Ang. Brak pamięci – zachowaj swoje dane: dość częsty komunikat w pewnym przeklamowanym edytorze tekstów dla systemu MS-Windows.

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 nieużywalne w komputerach osobistych. Ponadto należy sobie zdać sprawę, że obsługa skomplikowanych struktur danych jest na ogół dość czasochłonna – jedno kryterium oddziaływało zatem na drugie!

Wcale nie jest aż tak dobrze z szybkością współczesnych komputerów³, aby przestać się tym zupełnie przejmować. Bo cóż z tego, że komputer $xxxDX$ jest 12 razy szybszy od $yyySX$, 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 następujące pytania:

- Jakiego komputera użył?
- Jaka jest częstotliwość pracy zegara taktującego procesor?
- Czy program był jedynym wykonującym się wówczas w pamięci? Jeśli nie, to jaki miał priorytet?
- 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*, nie mająca nic wspólnego ze szczególnymi natury, nazwijmy to, „sprzętowej”.

Parametrem decydującym najczęściej o czasie wykonania określonego algorytmu jest *rozmiar danych*⁴, z którymi ma on do czynienia. Pojęcie *rozmiaru danych* ma wielorakie znaczenie: dla funkcji sortującej tablicę będzie to po prostu rozmiar tablicy, natomiast dla programu obliczającego wartość funkcji silnia będzie to 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

³ Oczywiście mam na myśli komputery osobiste.

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

jest intuicyjnie bardzo proste, dalej będziemy używać właśnie tego nieprecyzyjnego określenia w miejsce rozwlekłych wyjaśnień cytowanych powyżej.

Powróćmy jeszcze do przykładu przytoczonego na samym początku tego rozdziału. Nieprzygotowany Czytelnik widząc stwierdzenia: „czas wykonania programu równy 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. Zawiera ona krótkie zestawienie czasów wykonania algorytmów przy następujących założeniach:

- niech elementarny czas wykonania wynosi jedną mikrosekundę;
- niech pewien algorytm A ma złożoność obliczeniową Θ równą $n!$ Wówczas dla danej wejściowej o rozmiarze x i klasy algorytmu $n!$ czas wykonania programu jest proporcjonalny do $x!$).

Przy powyższych założeniach można otrzymać zacytowane w tabelce wyniki⁵ – dość szokujące, zwłaszcza jeśli spojrzymy na ostatnie jej pozycje.

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

*Tabela 3 - 1.
Czasy wykonania
programów dla
algorytmów różnej
klasy.*

	10	20	30	40	50	60
n	0,000 01 s	0,000 02 s	0,000 03 s	0,000 04 s	0,000 05 s	0,000 06
n^2	0,000 1 s	0,000 4 s	0,000 09 s	0,001 6 s	0,002 5 s	0,003 6 s
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	$200 \cdot 10^6$ w	$1,3 \cdot 10^{13}$ w
$n!$	3,6 s	768 w	$8,4 \cdot 10^{16}$ w	$2,6 \cdot 10^{32}$ w	$9,6 \cdot 10^{48}$ w	$2,6 \cdot 10^{66}$ w

Aby dobrze zrozumieć mechanizmy obliczeniowe używane przy analizie złożoności algorytmów, zgłębimy wspólnie kilka charakterystycznych przykładów obliczeniowych. Nowe pojęcia związane z obliczaniem złożoności obliczeniowej algorytmów.

⁵ Oznaczenia: s-sekunda, w-wiek.

mów zostaną wprowadzone na reprezentatywnych przykładach, co wydaje się lepszym rozwiązaniem niż zacytowanie suchych definicji.

3.2. Przykład 1: 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:

$$\begin{aligned}0! &= 1, \\n! &= n * (n-1)! \text{ gdzie } n \geq 1\end{aligned}$$

Odpowiadająca tej formule funkcja w C++ miała 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:

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

Powyższe wzory należy odczytać w sposób następujący: dla danej wejściowej równej zero czas wykonania funkcji, oznaczany jako $T(0)$, równy 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 $silnia(100)$... 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 wy-

konania 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 ilość 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. Niewiele bowiem się zmieni, 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?

Tę poszukiwaną funkcję będziemy zwać złożonością teoretyczną¹ 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 pierwszym pozwolimy sobie pominąć stałą 1 i wynik nie ulegnie znaczającej zmianie. W równaniu drugim o wiele ważniejsza jest funkcja kwadratowa niż liniowa zależność od n ; podobnie jest w równaniu trzecim, w którym dominuje funkcja 2^n .

Tabela 3 - 2.
Złożoność teoretyczna
algorytmów – przykłady.

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

Pojęcie funkcji O jest jednak kluczowe, zatem dla ciekawskich warto przytoczyć formalną definicję matematyczną. W tym celu przypomnijmy następujące oznaczenia znane z podręczników analizy matematycznej:

- N, \mathbb{R} i są zbiorami liczb odpowiednio naturalnych i rzeczywistych (wraz z zerem);
- Plus (+) przy nazwie zbioru oznacza wykluczenie z niego zera (np. N^+ jest zbiorem liczb naturalnych dodatnich);
- \mathbb{R}^+ będziemy oznaczać zbiór liczb rzeczywistych dodatnich lub zero;
- Znak graficzny \mapsto oznacza przyporządkowanie;
- Znak graficzny \forall należy czytać jako: *dla każdego*;

¹ Lub klasą algorytmu – określenie zresztą znacznie częściej używane.

- Znak graficzny \exists należy czytać jako: *istnieje*;
- 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 $T: N \mapsto \mathbb{R}^*$ możemy zdefiniować jako:

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

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 nie znane 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.: $T(n) = 5n^3 + 3n^2 + 2 \in O(n^3)$ (dobieramy doświadczalnie” $c=1$ i $n_0=0$, wówczas $n^3 \in 5n^3 + 3n^2 + 2 \in O(n^3)$). W sposób zbliżony można przeprowadzić dowody wielu podobnych zadań.

Funkcja O jest wielkością, której można używać w równaniach matematycznych.

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 „niweluje” wpływ wszelkich współczynników o wartościach stałych.

Przypomnijmy elementarny wzór podający zależność pomiędzy logarytmami o różnych podstawach:

$$\log_b x = \frac{\ln x}{\ln b}.$$

W obliczeniach wykonywanych przez programistów zdecydowanie króluje podstawa 2, bowiem jest wygodnie zakładać, że np. rozmiar tablicy jest wielokrotnością liczby 2 etc.

Następnie na podstawie takich założeń częstokroć wyliczana jest złożoność praktyczna i z niej dedukowana jego klasa, czyli funkcja O . 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! Na podstawie bowiem zacytowanego wyżej wzoru możemy z łatwością zauważyć, że logarytmy o odmiennych podstawach różnią się pomiędzy sobą tylko pewnym współczynnikiem stałym, który zostanie „pochłonięty” przez O na podstawie własności

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

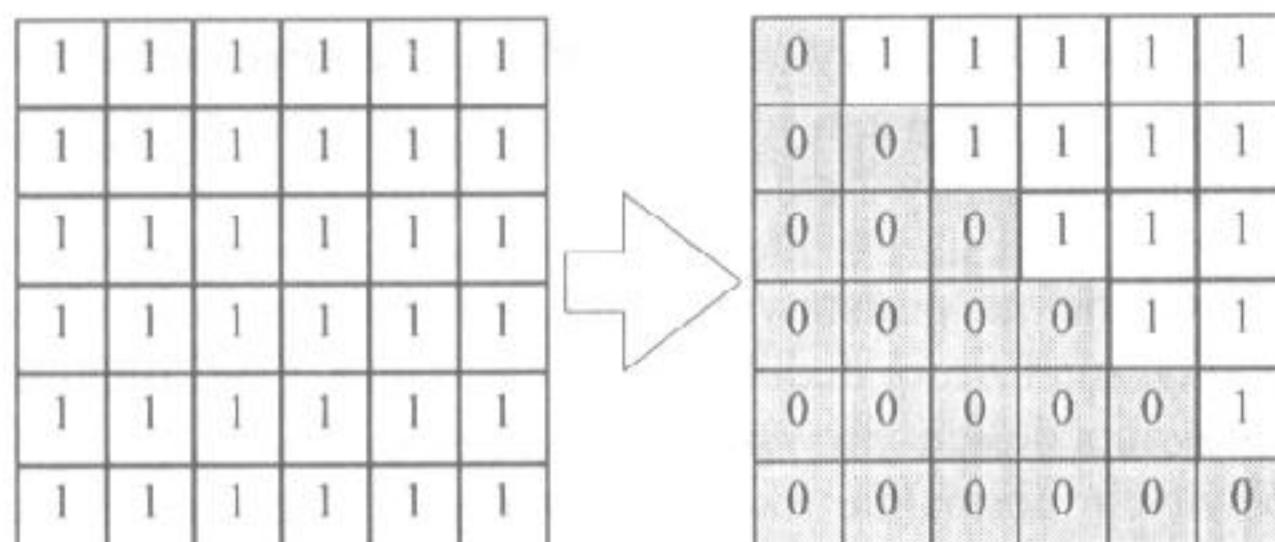
Z tego właśnie względu w literaturze mówi się, że „algorytm $A \in O(\log N)$ ”, a nie „ $A \in O(\log_2 N)$ ”.

Popatrzmy jeszcze na inny aspekt stosowania O -notacji. Założymy, 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 ”.

3.3. Przykład 2: Zerowanie fragmentu tablicy

Rozwiążemy teraz następujący problem: jak wyzerować fragment tablicy (tzn. macierzy) poniżej przekątnej (wraz z nią)? Ideę przedstawia rysunek 3 - 1.

Rys. 3 - 1.
Zerowanie
tablicy.



Funkcja wykonująca to zadanie jest bardzo prosta:

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

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 $n+1$ razy¹.

¹ Warto zauważyć, że istniejące w C++ pętle łatwo dają się sprowadzić do odmiany pętli zacytowanej powyżej.

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:

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

... 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 samo 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.

3.4. Przykład 3: 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. Założymy, że otrzymujemy ten „wyrwany z kontekstu” fragment kodu i musimy się zająć jego analizą:

```

const N=10;
int t[N];
funkcja_ad_hoc()
{
    int k, i;
    int suma=0;                                // t_a
    while (i<N)                                // t_c
    {
        while (j<=t[i])                        // t_c
        {
            suma=suma+2;                      // t_a
            j=j+1;                            // t_a
        }
        i=i+1;                                // t_a
    }
}

```

Uprośćmy nieco problem zakładając, że:

- najbardziej czasochłonne są instrukcje porównania, wszelkie inne ignorujemy zaś jako nie mające większego wpływu na czas wykonania programu.
- zamiast pisać explicite t_c wprowadzimy pojęcie czasu jednostkowego wykonania instrukcji, oznaczając go przez t_c .

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 poradzimy 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 nieznajomość 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 w 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 ilości założeń, przeprowadził analizę statystyczną zadania i doprowadził do ostatecznego wyniku w satysfakcjonującej nas postaci.

3.5. Przykład 4: Różne typy złożoności obliczeniowej

Postawmy ponownie przed sobą następujące zadanie: należy sprawdzić, czy pewna liczba x znajduje się w tablicy o rozmiarze n . Zostało już ono rozwiązane w rozdziale 2, spróbujmy teraz napisać iteracyjną wersję tej samej procedury. Nie jest to czynność szczególnie skomplikowana i sprowadza się do ułożenia następującego programu:

szukaj.cpp

```
const n=10;
int tab[n]={1,2,3,2,-7,44,5,1,0,-3};

int szukaj(int tab[],int x)
// funkcja zwraca indeks poszukiwanego elementu x
int pos=0;
while ((pos<n) && (tab[pos]!=x))
    pos++;
if (pos<n)
```

```

    return pos;
else                                //element został znaleziony
    return -1;                         // porażka poszukiwań
}

```

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 ilość 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 trafiliśmy 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 uwzględnić konfigurację danych, która w przypadku przeszukiwania tablicy ma 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 przyjmijmy, 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 $D_{n,n}$ 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:

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

Otrzymujemy zatem wyrażenie:

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

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

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

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 programistie pewne granice, w których może on go umieścić. Czy podobnie możemy rozpatrywać przypadek *średni*?

Jak łatwo zauważać, 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.

3.6. 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 ułatwiają nam to zadanie, pozwalając niejednokrotnie określić natychmiastowo 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 (pętle 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 fragmentu programu:

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

Wybieramy instrukcję *suma=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 $O(n^2)$.

3.7. Analiza programów rekurencyjnych

Większość programów rekurencyjnych nie da się niestety rozważyć przy użyciu metody poznanej w przykładzie znajdującym się w §3.2. Istotnie, zastosowana tam metoda rozwiązywania równania rekurencyjnego, polegająca na rozpisaniu jego składników i dodaniu układu równań stronami, 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 przedstawał. Osoby szczególnie zainteresowane stroną matematyczną powinny dotrzeć bez kłopotu do odpowiedniej literatury (patrz uwagi zamieszczone we wstępie rozdziału).

3.7.1.Terminologia

Lektura kilku kolejnych paragrafów wymaga od nas poznania terminologii, którą będziemy się dość często posługiwać. Pomimo „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 gwoli formalności niż w jakimś praktycznym celu.

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:

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

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

Otrzymane powyżej współczynniki λ , 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. Jego postać zależy od formy, jaką przybiera reszta $u(n,m)$. Oto możliwe przypadki:

- Jeśli $u(n,m)=0$, to:

$$RS=0.$$

- Jeśli $u(n,m)$ jest wielomianem stopnia m i zmiennej n oraz I (jeden) *nie jest* rozwiązaniem RC , wówczas:

$$RS=Q(n,m),$$

gdzie: $Q(n,m)$ jest pewnym wielomianem stopnia m i zmiennej n , o współczynnikach nieznanych (do odnalezienia).

- Jeśli $u(n,m)$ jest wielomianem stopnia m i zmiennej n oraz I 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.

3.7.2. Ilustracja metody na przykładzie

Spójrzmy jeszcze raz na **Przykład z §3.2**. Otrzymaliśmy wtedy następujące równanie:

$$T(0) = 1,$$

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

Spróbujmy je rozwiązać nowo poznaną metodą.

ETAP 1 Poszukiwanie równania charakterystycznego:

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

ETAP 2 Pierwiastek równania charakterystycznego:

Jest to oczywiście $r=1$.

ETAP 3 Równanie ogólne:

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

ETAP 4 Poszukiwanie równania szczególnego:

Wiemy, że $u(n,m)=1$ (jeden jest wielomianem stopnia zero!). Ponadto I 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 to jest to identyczne z poprzednim rozwiązaniem¹.

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 niskim znaczeniu praktycznym.

3.7.3. Rozkład „logarytmiczny”

Z rozdziału poprzedniego pamiętamy zapewne zadanie poświęcone przeszukiwaniu binarnemu. Jedną z możliwych wersji funkcji² 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
            // element znaleziony
            return -1;           // element nie odnaleziony
    else
    {
        int mid=(left+right)/2;
        if(tab[mid]==x)
            return mid;          // element znaleziony!
        else
            if(x<tab[mid])
                return szukaj_rec(tab,x,left,mid);
            else
                return szukaj_rec(tab,x,mid,right);
    }
}
```

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

¹ Jeśli dwie metody prowadzą do takiego samego, prawidłowego wyniku, to istnieje duże prawdopodobieństwo, iż obie są dobre...

² Innej niż poprzednio zaproponowana.

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

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

Widać już, że powyższy układ nijak się ma 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 całe 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”...

Załóżmy, 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 tablicy 3 - 3.

Tabela 3 - 3.

Czasy wykonania programów dla algorytmów różnej klasy.

	Klasa algorytmu	Uwagi
$a > d(b)$	$T(n) \in O(n^{\log_b a})$	
$a < d(b)$	$T(n) \in O(n^{\log_b d(b)})$	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 w liczeń bazujących na następujących założeniach:

- n jest potągią 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 powyższym wzorom z łatwością rozwiązana. Spróbujmy dla przykładu skończyć zadanie dotyczące przeszukiwania binarnego. Jak pamiętamy, otrzymaliśmy wówczas następujące równania:

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

Patrząc na zestaw podanych powyżej wzorów widzimy, że nie jest on zgodny ze „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:

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

3.7.4. 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:

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

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

3.7.5. Funkcja Ackermanna, czyli coś dla smakoszy

Gdyby małe dzieci знаły się odrobinę na informatyce, to rodzice na pewno by je straszyli 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:

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);
}
```

Pytanie dotyczące tego programu brzmi: jaki jest powód komunikatu *Stack overflow!* (ang. przepłynięcie stosu) podczas próby wykonania go? Komunikat ten sugeruje jednoznacznie, iż podczas wykonania programu nastąpiła znacząca ilość 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 Ackermannova 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:

$$\forall n \geq 1, A(n,3) = 2^{\overbrace{2}^2} \Bigg\}^n.$$

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

$$A(1,4) = 2.$$

$$A(2,4) = 2^2 = 4.$$

$$A(3,4) = 2^{2^2} = 65536.$$

$$A(4,4) = 2^{\overbrace{2}^2} \Bigg\}^{65536}.$$

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

3.8. Zadania

Zad. 3-1

Proszę rozważyć problem prawdziwości lub fałszu poniższych równań:

a) $T(n^3) \in O(n^3);$

- b) $T(n^2) \in O(n^3)$;
- c) $T(2^{n+1}) \in O(2^n)$;
- d) $T((n+1)!) \in O(n!)$;
- e) $T(n) \in O(n) \Rightarrow \{T(n)\}^2 \in O(n^2)$;
- f) Twój własny przykład?

Zad. 3-2

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

```
int fib(int n)
{
    if (n==0)
        return 1;
    else
        if (n==1)
            return 1;
        else
            return fib(n-1)+fib(n-2);
}
```

Proszę określić, jakiej klasy jest to funkcja.

Zad. 3-3

Proszę przeanalizować jeden ze swoich programów, taki, 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 „sporną” instrukcję przypisania „wyrzuca się” przed pętlę, przyspieszając jej działanie. Podobnie odpowiednio układając kolejność pewnych obliczeń, można wykorzystywać częściowe wyniki, będące rezultatem pewnego bloku instrukcji, w dalszych blokach – pod warunkiem oczywiście, 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.

Zad. 3-4

Proszę rozwiązać następujące równanie rekurencyjne:

$$\begin{aligned} u_n &= u_{n-1} - u_n \cdot u_n - 1 \quad (\text{dla } n \geq 1), \\ u_0 &= 1. \end{aligned}$$

3.9. Rozwiązania i wskazówki do zadań

Zad. 3-1

Równanie rekurencyjne ma postać:

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

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

ETAP 3 Równanie ogólne:

Z teorii wyłożonej wcześniej wynika, że równanie ogólne ma postać $RO = Ar_1^n + Br_2^n$ – zostawmy je chwilowo w tej formie.

ETAP 4 Poszukiwanie równania szczególnego:

Wiemy, że $u(n,m)=2$ i jest to wielomian stopnia zero. Z teorii wynika, że musimy odnaleźć również jakiś wielomian stopnia zerowego, czyli mówiąc po ludzku: pewną stałą c . Równanie szczególne jest rozwiązaniem równania rekurencyjnego, zatem możemy je podstawić w miejsce $T(n)$, $T(n-1)$ i $T(n-2)$ (Tutaj n nie gra żadnej roli!). Wynikiem tego podstawienia będzie oczywiście $c=2+c+c \Rightarrow c=-2$.

ETAP 5 Poszukiwanie ostatecznego rozwiązania:

Poszukiwanym rozwiązaniem jest suma $RO + RS$:

$$T(n) = RO + RS = (Ar_1^n + Br_2^n) + (-2) = Ar_1^n + Br_2^n - 2.$$

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 ($T(0)=1$ i $T(1)=2$). Po wykonaniu podstawienia otrzymamy:

$$1 = A + B - 2,$$

$$2 = Ar_1 + Br_2 - 2.$$

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

Zad. 3-4

Oto szkic rozwiązania:

Załóżmy, 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:

$$\begin{aligned} v_n &= v_{n-1} + 1, \\ v_0 &= 1. \end{aligned}$$

Jego rozwiązaniem jest oczywiście $v_n = n+1$ (patrz §3.2.) 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 przedżej czy później z tym zagadnieniem musi mieć do czynienia. Opisy metod sortowania będą dotyczyć wyłącznie tzw. sortowania wewnętrznego, używającego wyłącznie pamięci głównej komputera¹. Po sporych wahaniach zdecydowałem się jednak nie podejmować problematyki tzw. sortowania zewnętrznego². Sortowanie zewnętrzne dotyczy sytuacji, z którą większość Czytelników być może nigdy się nie zetknie w praktyce programowania: ilość danych do sortowania jest tak olbrzymia, że niemożliwa do umieszczenia w pamięci w celu posortowania ich przy pomocy jednej z wielu metod sortowania „wewnętrznego”.

Dlaczego przyjmuję tak optymistyczną hipotezę? W chwili obecnej jest zauważalne systematyczne tanieanie nośników pamięci RAM i dysków twardych. Proces ten jest nieodwracalny i jedyne, o co się można spierać, to stopień jego zaawansowania.

Mój pierwszy prywatny komputer osobisty typu IBM PC XT miał 1MB RAM i dysk twardy 20MB. Od tego momentu minęło zaledwie kilka lat: dzisiaj większość programów nie chciałaby zwyczajnie wystartować z tak małą ilością pamięci, a objętość 20MB jest wystarczająca na instalację pojedynczego programu... Coraz głośniej zaczyna być o bazach danych całkowicie rezydujących w pamięci (dla zwiększenia sprawności), rzecz niewyobrażalna kilka lat temu w praktyce. W konsekwencji takiego *status quo* zdecydowałem się nie poświęcać sortowaniu zewnętrznemu specjalnej uwagi. Osoby zainteresowane odnajdą szczegółowe informacje na przykład w [AHU87], [FGS90], [Knu75], [Sed92] – mam nadzieję, że większość Czytelników odniesie się do tego posunięcia ze zrozumieniem.

¹ Ang. internal sorting.

² Ang. external sorting.

Potrzeba sortowania danych jest związana z typowo ludzką chęcią gromadzenia i/lub porządkowania. Darujmy sobie jednak pasjonującą dyskusję na temat socjalnych aspektów sortowania i skoncentrujmy się raczej na zagadnieniach czysto algorytmicznych...

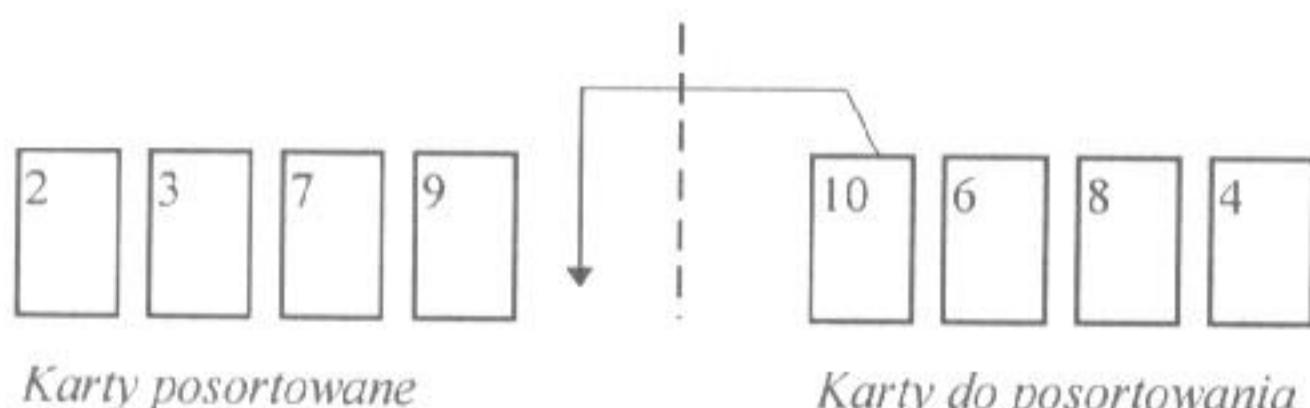
Istotnym problemem w dziedzinie sortowania danych jest ogromna różnorodność algorytmów wykonujących to zadanie. Początkujący programista często nie jest w stanie samodzielnie dokonać wyboru algorytmu sortowania najodpowiedniejszego do konkretnego zadania. Jedno z możliwych podejść do tematu polegałoby 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 pośrednich obserwacji wynika, że programiści raczej używają dobrze sprawdzonych „klasycznych” rozwiązań, takich jak np. sortowanie przez *wstawianie*, sortowanie *szylkie*, sortowanie *bąbelkowe*, niż równie dobrych (jeśli nie lepszych) rozwiązań, które służą głównie jako tematy artykułów czy też przyczynki do badań porównawczych z dziedziny efektywności algorytmów.

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.

4.1. 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:

Rys. 4 - 1.
Sortowanie przez
wstawianie (I).

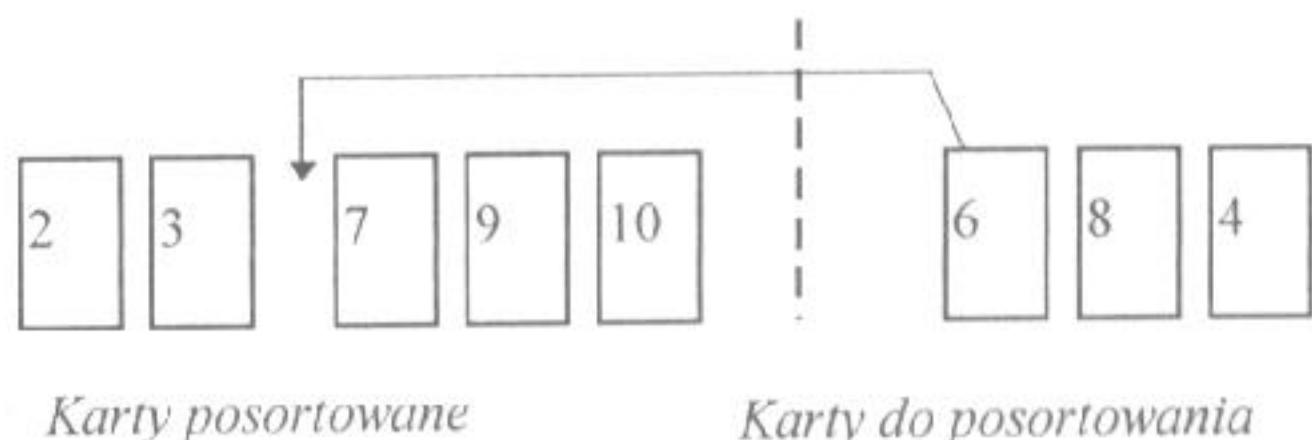


Idea tego algorytmu opiera się na następującym niezmienniku: w danym momencie trzymamy w ręku karty posortowane³ oraz karty pozostałe do posortowania. W celu kontynuowania procesu sortowania bierzemy pierwszą z brzegu kartę ze sterty nieposortowanej i wstawiamy ją na właściwe miejsce w pakiecie już wcześniej 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'.

Rys. 4 - 2.

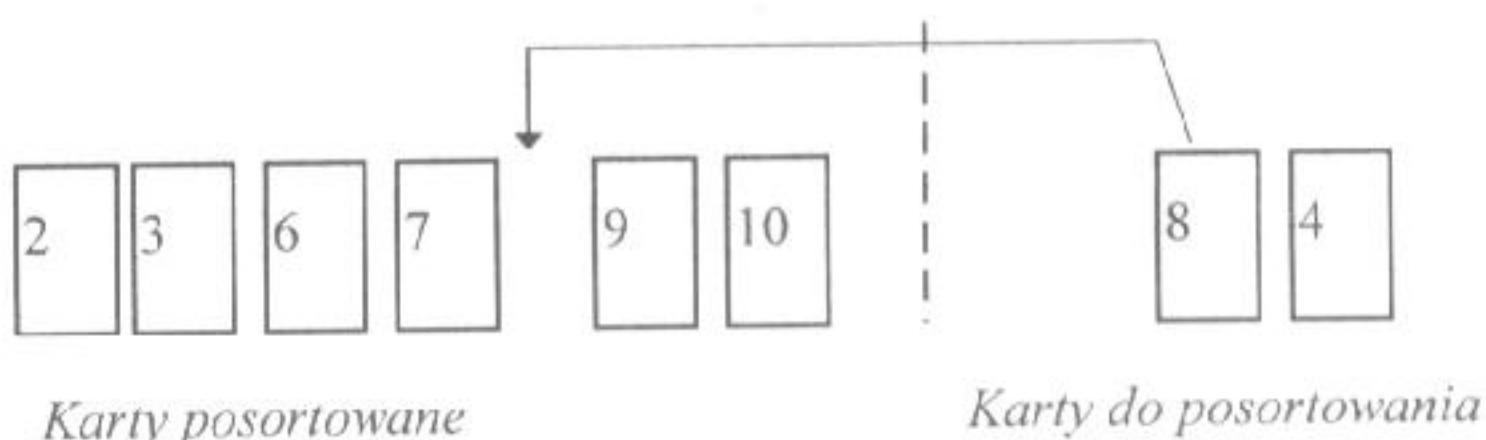
Sortowanie przez wstawianie (2).



Tuż po poprawnym ułożeniu szóstki otrzymujemy „rozdanie” z rysunku 4 - 3.

Rys. 4 - 3.

Sortowanie przez wstawianie (3).



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

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 programu:

³ Na samym początku algorytmu możemy mieć puste ręce, ale dla zasady twierdzimy wówczas, że trzymamy w nich zerową ilość kart.

insert.cpp

```

void InsertSort(int *tab)
{
    for(int i=1; i<n; i++)
    {
        int j=i;           // 0..i-1 są posortowane
        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 §5.1.3).

4.2. 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:

bubble.cpp

```

void bubble(int *tab)
{
    for(int i=1; i<n; i++)
    {
}

```

```

for (int j=n-1; j>i; j--)
    if (tab[j]<tab[j-1])
        // zamiana tab[j-1] z tab[j]
        int tmp=tab[j-1];
        tab[j-1]=tab[j];
        tab[j]=tmp;
    }
}

```

Przeanalizujmy dokładnie sortowanie bąbelkowe pewnej 7-elementowej tablicy. Na rysunku 4 - 4 element „zaciemiony” 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!

Rys. 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 konfiguracje 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])
            {
                zamiana(tab[j-1], tab[j]);
                k=j;
            }
        left=k+1;
        for(j=left; j<right; j++)
            if(tab[j-1]>tab[j])
            {
                zamiana(tab[j-1], tab[j]);
                k=j;
            }
        right=k-1;
    }while (left<right);
}
```

4.3. Quicksort, algorytm klasy $O(N \log_2 N)$

Jest to słynny algorytm¹, 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 robi... 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 właściwie procedurę, natomiast rekurencja zapewnia „sklejenie” wyników cząstkowych 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 dzielony² wg klucza symbolicznie przedstawionego na rysunku 4 - 5.

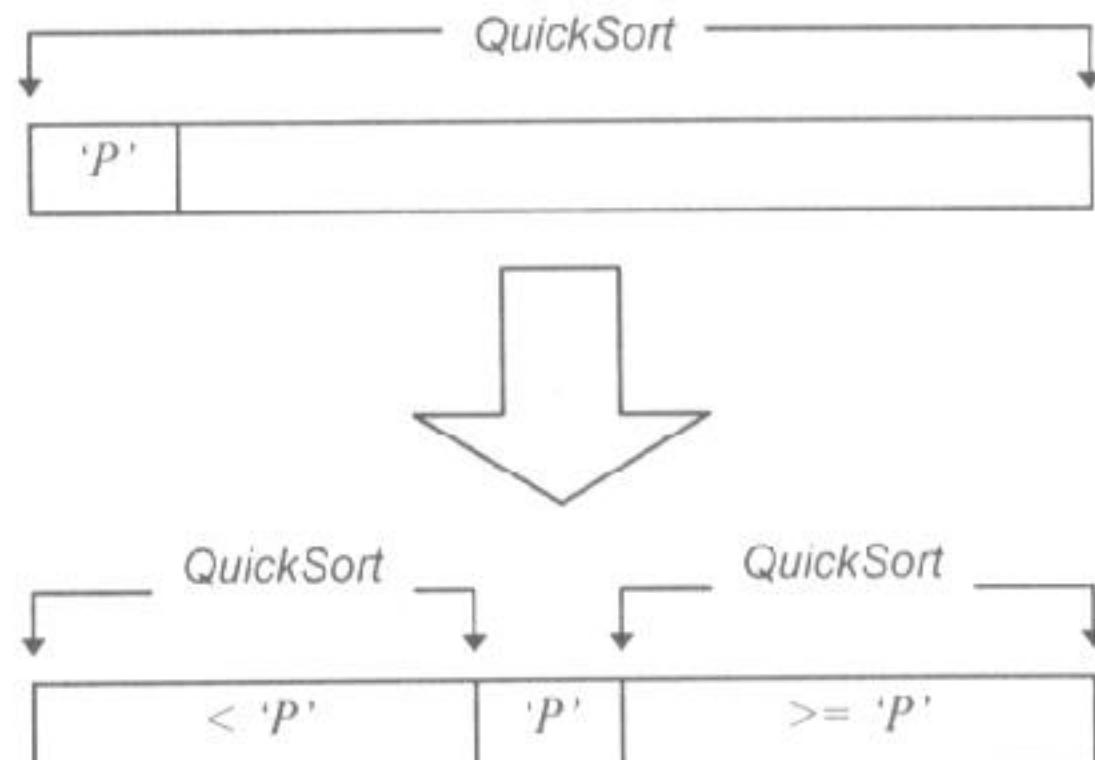
Kolejnym etapem jest zaaplikowanie procedury *Quicksort* na „lewym” i „prawym” fragmencie tablicy, czego efektem będzie jej posortowanie. To wszystko!

Rys. 4 - 5.
Podział tablicy
w metodzie Quicksort.

<i>element < 'P'</i>	<i>element osiowy 'P'</i>	<i>element >= 'P'</i>
-------------------------	---------------------------	--------------------------

Na rysunku 4 - 6 są przedstawione symbolicznie dwa główne etapy sortowania metodą *Quicksort* (P oznacza tradycyjnie komórkę tablicy służącą za „oś”).

Rys. 4 - 6.
Zasada działania pro-
cedury Quicksort.



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

² Elementy tablicy są fizycznie przemieszczane, jeśli zachodzi potrzeba.

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

Przedstawiona powyżej metoda sortowania charakteryzuje się olbrzymią prostotą, wyrażoną najdoskonalej przez zwięzły zapis samej procedury:

```
void Quicksort(int *tab, int left, int right)
{
    if (left < right)
    {
        int m;
        // Podziel tablicę względem elementu osiowego:
        // P=pierwszy element, czyli tab[left];
        // Pod koniec podziału element 'P' zostanie
        // przeniesiony do komórki numer '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: *Quicksort*-ów 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 nie-wątpliwie doszukać się w rozwiązaniu przedstawionym w [Ben92].

Pomysł opiera się na zachowaniu dość prostego niezmiennika w aktualnie „rozdzielanym” fragmencie tablicy (patrz rysunek 4 - 7).

Rys. 4 - 7.
Budowa niezmiennika
dla algorytmu
Quicksort.

P	$<P$		$>=P$		<i>Fragment niezbadany</i>
<i>left</i>		<i>m</i>		<i>i</i>	<i>right</i>

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 przemijający tablicę od **left** do **right**;
- m poszukiwany indeks komórki tablicy, w której umieścimy element osiowy.

Przemianowanie 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. W tym celu podczas przemieszczania indeksu i sprawdzamy prawdziwość niezmiennika $tab[i] > P$. Jeśli jest on fałszywy, to poprzez inkrementację i wymianę wartości $tab[m]$ i $tab[i]$ przywracamy „porządek”. Gdy zakończymy ostatecznie przeglądanie tablicy w pogoni za komórkami, które nie chciały się podporządkować niezmiennikowi, zamiana $tab[left]$ i $tab[m]$ doprowadzi do oczekiwanej sytuacji, przedstawionej wcześniej na rysunku 4 - 7.

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

qsort.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])
                zamiana(tab[++m],tab[i]);
        zamiana(tab[left],tab[m]);
        qsort(tab,left,m-1);
        qsort(tab,m+1,right);
    }
}
```

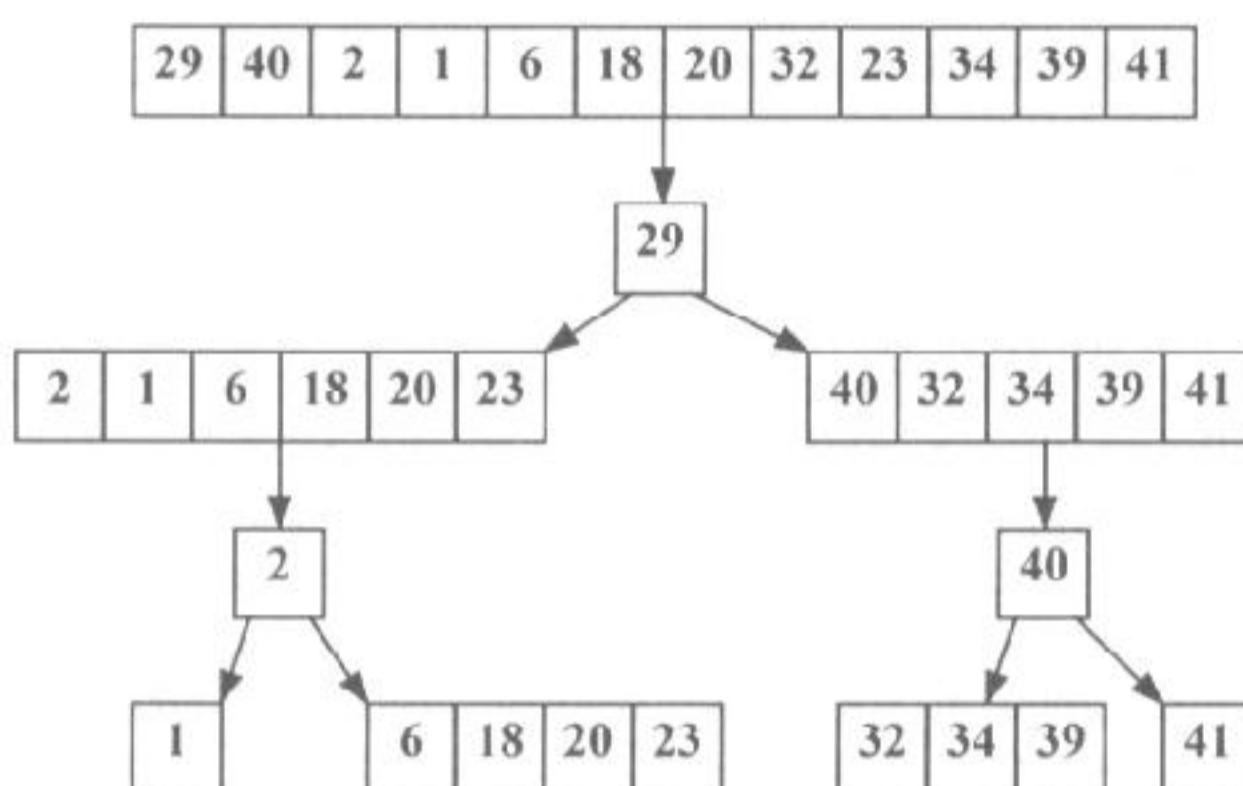
W celu dobrego zrozumienia działania algorytmu spróbujmy posortować nim „ręcznie” małą tablicę, np. zawierającą następujące liczby:

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.

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* stanie dobrą przykład techniki programowania zwanej „dziel i rządź”, która zostanie dokładniej omówiona w rozdziale 9.

Rys. 4 - 8.
Sortowanie me-
dą Quicksort
na przykładzie.



Tutaj zapowiem jedynie, że chodzi o taką dekompozycję problemu, aby uzyskać zysk czasowy wykonywania programu (jak i przy okazji uproszczenie rozwiązywanego zadania). Algorytm *Quicksort* spełnia te oba założenia wręcz wzorcowo!

4.4. Uwagi praktyczne

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

- do sortowania małych ilości 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. Powód jest dość prosty: trzymając się dobrze znanych metod mamy większą pewność, iż nie popełnimy jakiegoś nadprogramowego błędu.

Podczas programowania warto również uważnie czytać, czy w bibliotekach standardowych używanego kompilatora nie ma już zaimplementowanej funkcji sortującej. Przykładowo w kompilatorze gcc 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: ilość 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 ilustracji 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*):

quick-gcc.cc

```
int comp(const void *x, const void *y)
{
    int xx=*(int*)x;                                // jawnia konwersja z typu
    int yy=*(int*)y;                                // 'void*' do 'int'

    if(xx < yy)      return -1;                      // < 0
    if(xx == yy)   return 0;                          // = 0
    else           return 1;                          // > 0
}

const n=12;
int tab[n]={40,29,2,1,6,18,20,32,34,39,23,41};

void main()
{
    qsort(tab,n,sizeof(int),comp);
    for (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));
}

void 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 „kota 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!

Rozdział 5

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, możliwość jego łatwej modyfikacji, czytelność zapisu algorytmów i... dobre samopoczucie programisty.

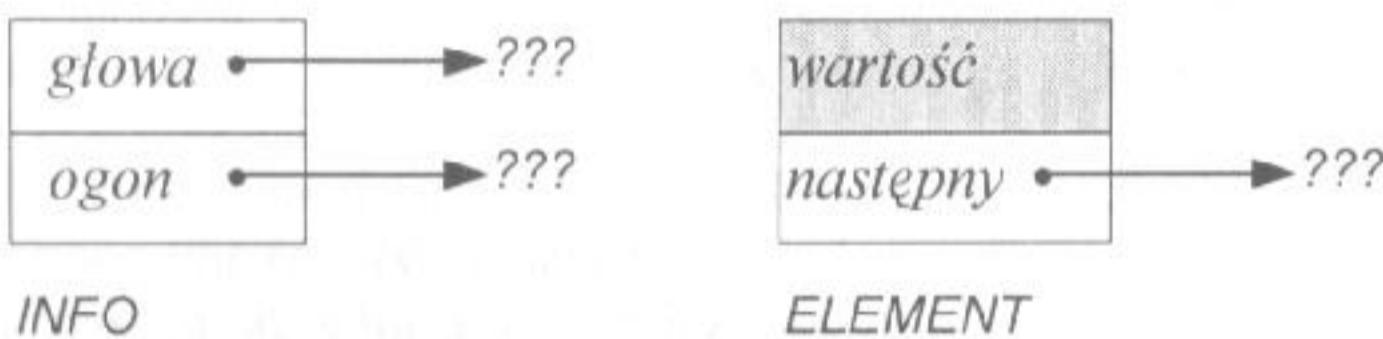
Każdy, kto poznał jakkolwiek język programowania, został niejako zmuszony do opanowania zasad posługiwania się tzw. typami podstawowymi. Przykładowo w C++ mamy do dyspozycji typy: *int, long, float, char*, typy wskaźnikowe etc. Mogą one posłużyć jako elementy bazowe rekordów, tablic, unii, które już zasługują na miano struktur danych – na tyle jednak prymitywnych, iż nie będą one stanowić przedmiotu naszych głębszych rozważań. Prawdziwa przygoda rozpoczyna się dopiero, gdy dostajemy do ręki tzw. *listy, drzewa binarne, grafy*... Wraz z nimi rozszerzają się znacznie możliwości rozwiązania programowego wielu ciekawych zagadnień; zwiększa się wachlarz potencjalnych zastosowań informatyki. Listy ułatwiają tworzenie elastycznych baz danych, drzewa binarne mogą posłużyć do analizy symbolicznej wyrażeń arytmetycznych, grafy¹ ułatwiają rozwiązanie wielu zagadnień z dziedziny tzw. *sztucznej inteligencji* – możliwości jest doprawdy bardzo dużo. 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. Zapraszam zatem do lektury.

¹ Materiał dotyczący grafów został, ze względu na jego znaczenie i rozmiar, wyodrębniony w rozdziale 10.

5.1. Listy jednokierunkowe

Lista jednokierunkowa jest oszczędną pamięciowo strukturą danych, pozwalającą grupować dowolną – ograniczoną tylko ilością dostępnej pamięci – liczbę elementów: liczb, znaków, rekordów... Jest to duża zaleta w porównaniu z tablicami, których rozmiar co prawda może być 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 50.000 razy pamięci na rekordy 4 bajtowe niż zarezerwowanie miejsca na tablicę zajmującą 200 KB! (W rzeczywistości lista, która pozwala zapamiętać 200 KB informacji zajmuje w pamięci oprócz owych „gołych” 200 KB pewną dodatkową pamięć. Z każdym rekordem jest związane dodatkowe pole na wskaźnik do kolejnego rekordu listy – patrz rysunek 5 - 1.

Rys. 5 - 1.
Typy rekordów
używanych pod-
czas programowa-
nia list.



Do budowy listy jednokierunkowej używane są dwa typy „komórek” pamięci. Pierwszy jest zwykłym rekordem natury informacyjnej, zawierającym dwa wskaźniki: do *poczatku* i do *konca* listy. Drugi typ komórek jest również rekordem, jednakże ma on już charakter roboczy. Zawiera bowiem *pole wartości* i *wskaźnik na następny element listy*. W typowych opisach struktur listowych nie wzmiankuje się zazwyczaj rekordu informacyjnego (nie jest on elementem struktury danych) – jest to oczywisty błąd. Kosztem kilku bajtów pamięci² 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 информацию o kolejności przybywania danych!).

Pola: *głowa*, *ogon* i *następny* są wskaźnikami³, natomiast *wartość* może być czymkolwiek – liczbą, znakiem, rekordem etc. W przykładach znajdujących się

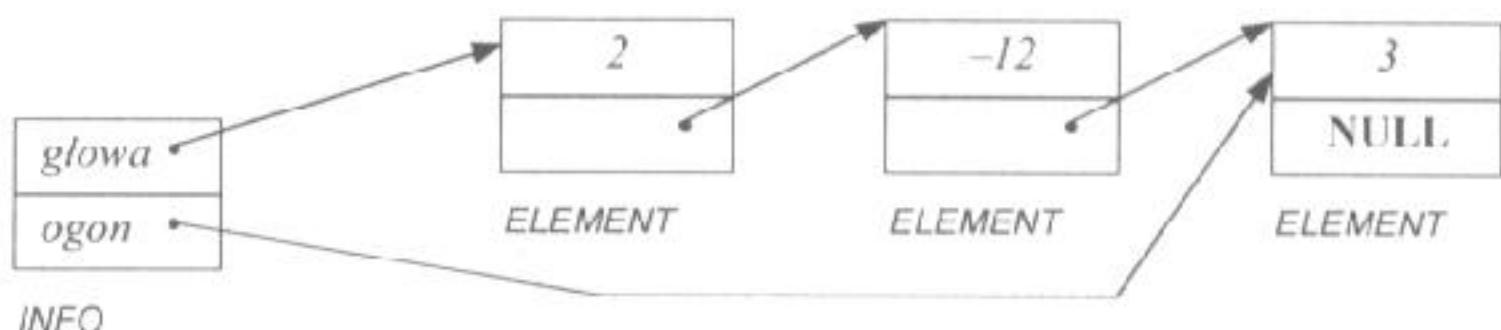
² W IBM PC zmienna wskaźnikowa zajmuje 4 lub 8 bajtów w zależności od użytego modelu pamięci.

³ Fakt „wskaazywania” na coś jest symbolizowany dalej przez „strzałki”.

w tej książce dla uproszczenia operuje się głównie wartościami typu *int*, 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. Na rysunkach znajdujących się w tej książce, wartość NULL będzie od czasu do czasu zaznaczana jako *0000h* – adres pamięci równy zero. Warto jednak pamiętać, że w ogólnym przypadku NULL *nie jest* bynajmniej równa 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 poprzez wartość specjalną NULL. Spójrzmy na rysunek 5 - 2 przedstawiający listę złożoną z trzech elementów: 2, -12, 3.

Rys. 5 - 2.
Przykład listy
jednokierunkowej
(1).

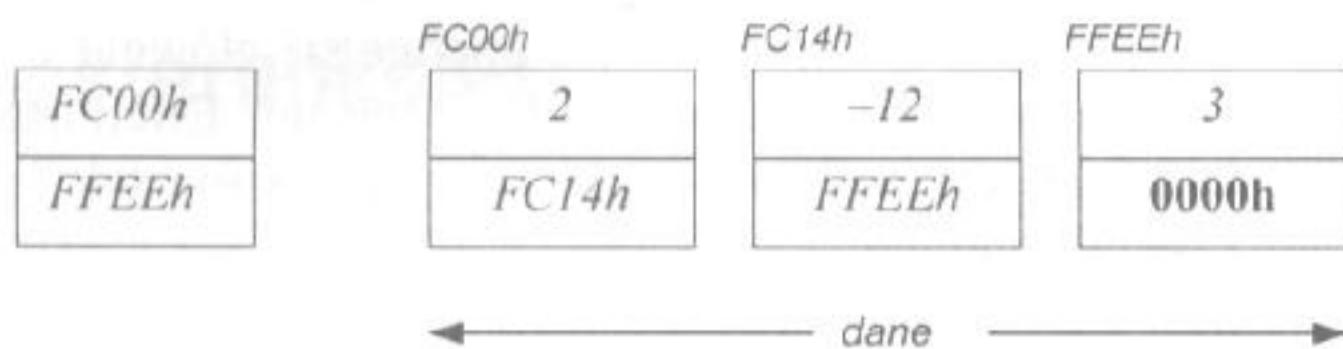


Rysunek 5 - 3 jest dokładnym odbiciem swojego poprzednika – z tą tylko różnicą, że w miejscu strzałek symbolizujących „wskaazywanie” są użyte konkretne wartości liczbowe adresów komórek pamięci. Heksadecymalna liczba umieszczona nad rekordem jest adresem w pamięci komputera, pod którym zostało mu przydzielone miejsce przez standardową procedurę *new*⁴.

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

⁴ Ze względów historycznych warto może przypomnieć, że w klasycznym języku C trzeba było w celu przydzielenia pamięci używać funkcji bibliotecznych *calloc* i *malloc*. W C++ instrukcja *new* robi dokładnie to samo, lecz o wiele czytelniej, i stanowi już element języka.

Rys. 5 - 3.
Przykład listy jednokierunkowej (2).



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

```

adres_tmp=info.głowa
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 (patrz uwagi zawarte w rozdziale 1). Oczywiście, nawet jeśli prezentacja algorytmu zostanie dokonana w pseudo-kodzie, to wersja dyskietkowa będzie zawierała w pełni kompilowalne wersje w C++.

5.1.1. 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

```

typedef struct rob
{
    int wartość;
    struct rob *następny; // wskaźnik do następnego elementu
} ELEMENT;
class LISTA
{

```

```
public:  
    int pusta()           // czy lista jest pusta?  
    {  
        return (inf.glowa==NULL);  
    }  
  
    // sumuje dwie listy:  
    LISTA friend& operator +(LISTA&, LISTA&);  
  
    void wypisz();        // wypisuje elementy listy  
  
    int szukaj(int x);    // szuka elementu x  
  
    void dorzuc1(int x);  // dorzuca x bez sortowania  
  
    void dorzuc2(int x);  // dorzuca x z sortowaniem  
  
    void zeruj();         // zerowanie listy  
    { inf.glowa=inf.ogon=NULL; }  
  
    LISTA& operator --(); // usuwa ostatni element listy  
  
    LISTA()               // konstruktor  
    { inf.glowa=inf.ogon=NULL; }  
  
    ~LISTA()              // destruktor  
    { while (!pusta()) (*this)--; }  
private:  
    typedef struct          // struktura informacyjna  
    {  
        ELEMENT *glowa;  
        ELEMENT *ogon;  
        } INFO;  
    INFO inf;                // rekord informacyjny  
    };                      // 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*...).

Klasa LISTA nie jest zbyt rozbudowana, jednak zawiera kilka rozwiązań, które wymagają dość szczegółowego komentarza. Kwestią otwartą pozostaje wybór ewentualnego utajnienia typów danych; programista musi sam podjąć odpowiednie decyzję mając na uwadze takie aspekty, jak: *sens* ujawniania/ukrywania atrybutów, parametry „sprawnościowe” metod etc. Propozycje przedstawione w tym rozdziale w żadnym razie nie pretendują do miana rozwiązań wzorcowych – takie bowiem nie istnieją wobec nieskończonej w zasadzie ilości 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 wzmiankowane jedynie poprzez swoje nagłówki.

5.1.2. Tworzenie listy jednokierunkowej

Najwyższa już pora na przedstawienie sposobu dołączania elementów do listy. Posłuży nam do tego celu kilka funkcji, o mniejszym lub większym stopniu skomplikowania. Na początek zdefiniowaliśmy miniaturową funkcję usługową *pusta*, która pomimo swej prostoty ma szansę być dość często używana w praktyce. Z uwagi na małe rozmiary funkcja ta została zdefiniowana wprost w ciele klasy. 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 zwiększenie 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
```

```
#include "lista.h"
void LISTA::dorzuc1(int x)
{
    // dorzucanie elementu bez sortowania
    ELEMENT *q=new ELEMENT;           // tworzenie nowej komórki
    q->wartosc=x;
    q->nastepny=NULL;
    if (inf.glowa==NULL)              // lista pusta?
        inf.glowa=inf.ogon=q;
    else
    {
        (inf.ogon)->nastepny=q;
        inf.ogon=q;
    }
}
```

Działanie funkcji *dorzuc1* jest następujące: w przypadku listy pustej 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 (wskażwanej 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)
```



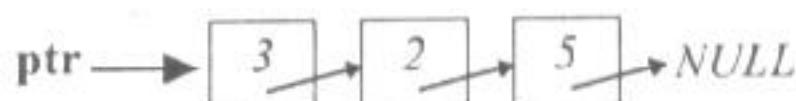
W tym i dalszych przykładach przyjmowane jest założenie, że przydział pamięci ZAWSZE kończy się sukcesem. W rzeczywistych programach jest to przypuszczenie dość niebezpieczne i warto sprawdzać, czy istotnie po użyciu instrukcji *ELEMENT *q=new ELEMENT* wartości *q* nie zostało przypisane NULL! Z uwagi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

Kod ten może być zilustrowany schematem z rysunku 5 - 4.

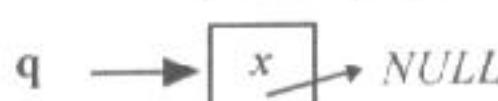
Rys. 5 - 4.

DŁĄCZANIE ELEMENTU NA JEGO POCZĄTEK.

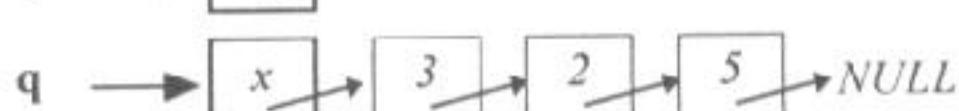
a) lista pierwotna



b) nowa komórka



c) zmodyfikowana lista



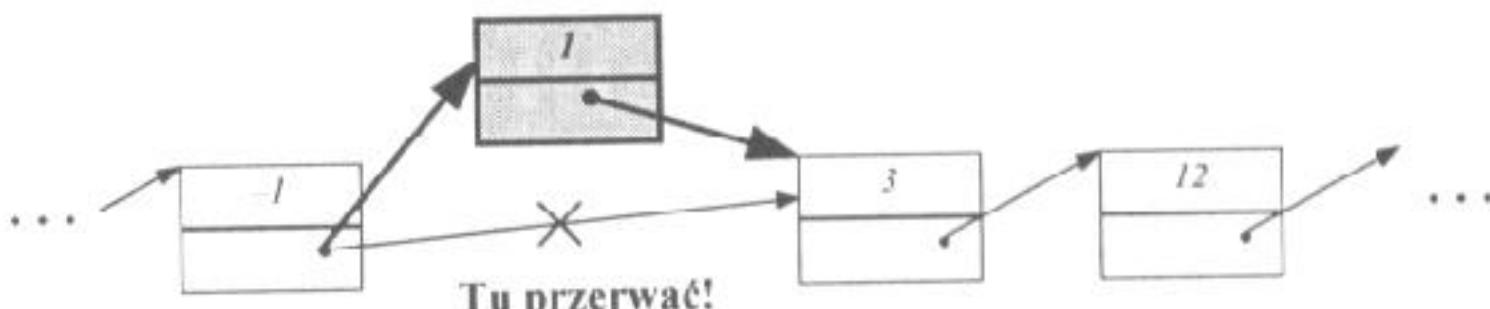
Sposób podany powyżej jest poprawny, ale pamiętajmy, że dokładając nowe elementy zawsze na początek listy tracimy istotną czasami informację na temat kolejności nadchodzenia elementów!

- 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.

Rys. 5 - 5.

DŁĄCZANIE ELEMENTU LISTY Z SOR-

TOWANIEM.



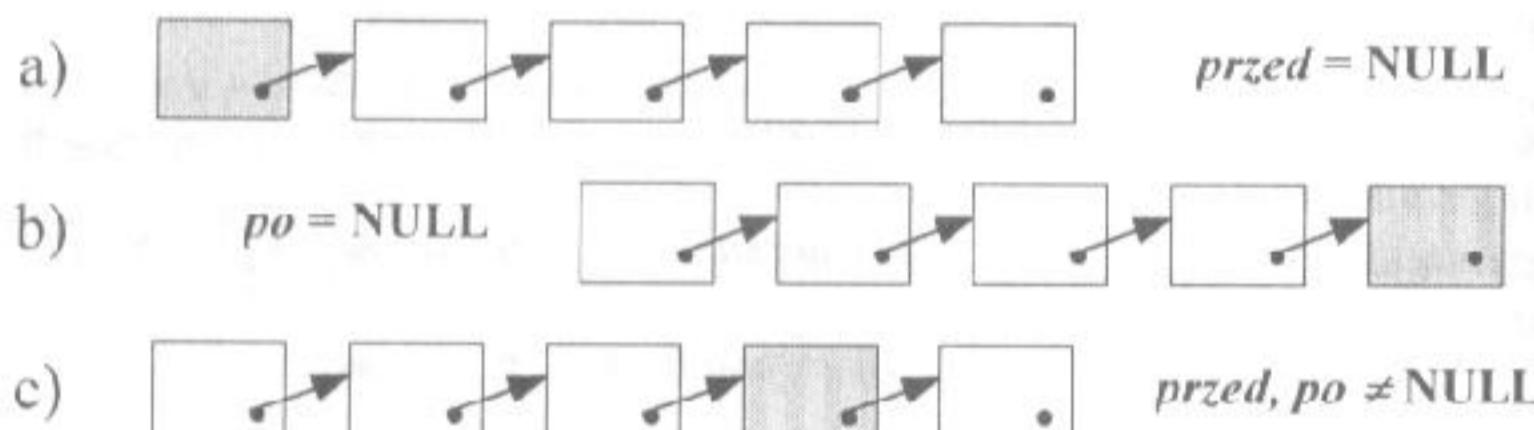
Nowy element (narysowany pogrubioną kreską) może zostać wstawiony na początek (a), koniec (b) listy, jak i również gdzieś w jej środku (c). W każdym

z tych przypadków w istniejącej liście trzeba znaleźć miejsce wstawienia, tzn. zapamiętać dwa wskaźniki: element, *przed* który 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 powiedziane, niestety realizacja jest 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).

Rys. 5 - 6.
Wstawianie nowego elementu do listy – analiza przypadków.



W zależności od ich wystąpienia zmieni 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
    // (ver.2 - z "sortowaniem")
    // tworzymy nowy element listy:
    ELEMENT *q=new ELEMENT;
    q->wartosc=x;    // wypełniamy jego zawartość

    // 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;
enum{SZUKAJ,ZAKONCZ} stan=SZUKAJ;
//zmienna wyliczeniowa
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;// zapamiętają "miejsce"
            // wstawiania
            }
        } // analiza rezultatu poszukiwań:
    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 (1);
    q=q->nastepny;
}
return (0);
}
```

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";
}

```

Do usuwania *ostatniego* elementu listy zatrudniliśmy przeddefiniowany operator dekrementacji.

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 elementu listy. Poniżej jest zamieszczony pełny tekst funkcji wykonującej to zadanie.

```

LISTA& LISTA::operator --()
{ // parametrem domyślnym jest sam obiekt
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 elementu listy...
        temp=temp->nastepny;
    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 peł-

noprawny obiekt – może aktywować dowolną metodę swojej klasy, czyli przykładowo sprawdzić swoją zawartość przy pomocy funkcji *wypisz*.

Przy okazji omawiania operatora dekrementacji spójrzymy 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 destruktora, 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 jawnego destruktora, który znając⁵ 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() { while (!pusta()) (*this)--; }
```

Jest to zwykła pętla *while*, która tak długo usuwa elementy z listy, aż stanie się ona pusta. Mimo tego, 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

⁵ *De facto* to my go znamy i dzielimy się tą cenną wiedzą z destruktorem... Rozsiane tu i ówdzie personifikacje są nie do uniknięcia w tego typu opisach!

dla danego obiektu wykonanie na sobie operacji „dekrementacji”. Obiekt ulegający z pewnych powodów destrukcji (typowe przypadki zostały wzmiankowane wcześniej) wywoła swój destruktory, który zaaplikuje na sobie tyle razy funkcję dekrementacji, aby całkowicie zwolnić pamięć wcześniejszej 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*, aktywując na rzecz tej ostatniej metodę *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;
    }

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

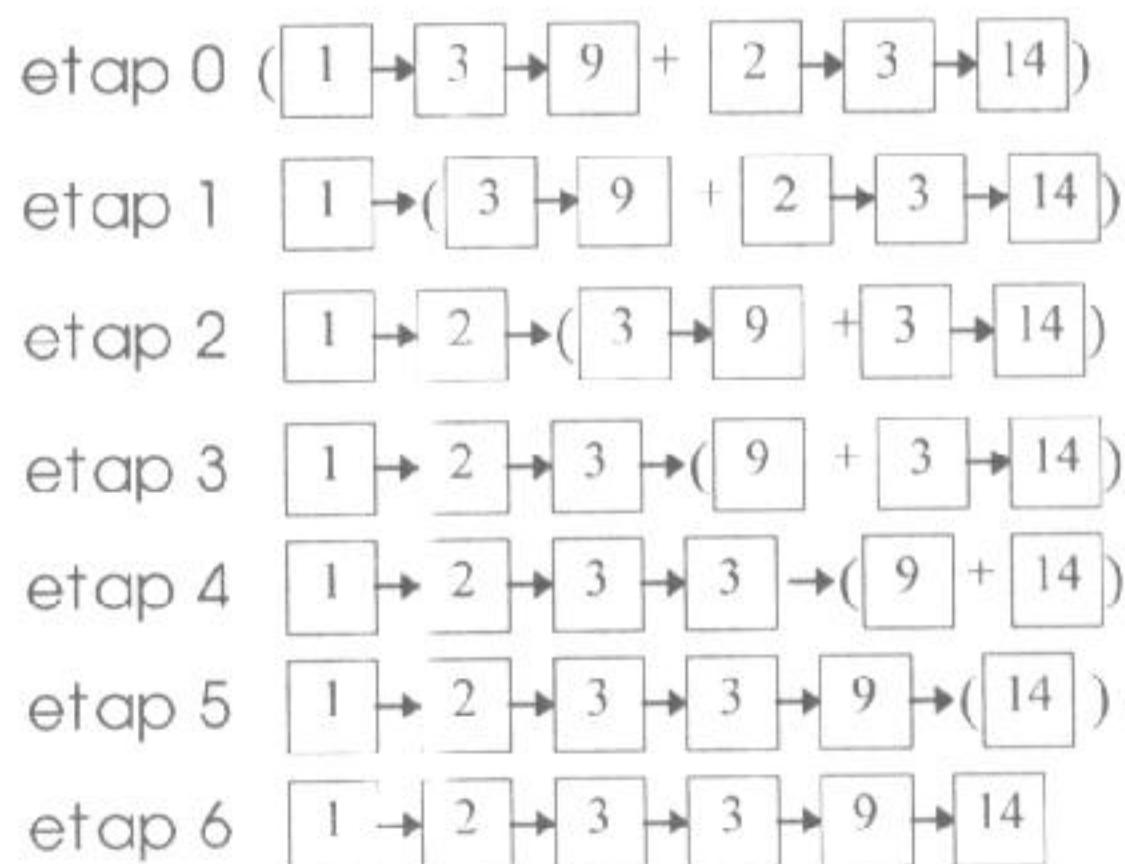
Czy jest to najlepsza metoda? Chyba nie, z uwagi chociażby na niepotrzebne dublowanie danych. Ideałem byłoby posiadanie metody, która wykorzystując fakt, iż listy są już posortowane⁶, dokona ich zespolenia ze sobą (tzw. *fuzji*) używając wyłącznie istniejących komórek pamięci, bez tworzenia nowych. Inaczej mówiąc, będziemy zmuszeni do manipulowania wyłącznie wskaźnikami i to jest jedyne narzędzie, jakie dostaniemy do ręki!

⁶ Zakładamy tym samym użycie podczas tworzenia listy metody *dorzuc2*.

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 w kierunku wartości niemalejących).

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 najmniejszy 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).

Rys. 5 - 7.
Fuzja list
na przykładzie.



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 jawnny – na przykład poprzez swój adres.

Fuzję list wykonamy w dwóch etapach. Wpierw przygotujemy prostą funkcję, która otrzymując dwie posortowane listy a i b , zwróci jako wynik listę, która

będzie ich fuzją. Rekurencyjny zapis tego procesu jest bardzo prosty i zbliżony stylem do rozwiązywania problemów listowych w takich językach jak *Lisp* lub *Prolog*:

```
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 procedurze *fuzja*, która będąc „zaprzyjaźnioną” z klasą LISTA, może dowolnie manipulować prywatnymi komponentami list *x* i *y*, które zostały jej przekazane w wywołaniu.

```
void fuzja(LISTA &x, LISTA &y)
{
    // listy 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.

```
void main()
{
    LISTA l1, l2;
    const 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(); // wypisz l1
    int tab2[n]={9,6,77,1,7,4};
```

```
cout << "L2 = ";
    for (i=0; i<n; 12.dorzuc2(tab2[i++]));
12.wypisz(); // wypisz l1
cout << "Efekt poszukiwań liczby 14 w liście l1: "
    << l1.szukaj(14) << endl;
cout << "Efekt poszukiwań liczby 0 w liście l1: "
    << l1.szukaj(0) << endl;
cout<<"Oto lista będąca sumą dwóch poprzednich\nL3= ";
LISTA 13=l1+l2;
13.wypisz();
cout << "Listy L1 i L2 pozostały bez zmian:\nL1 = ";
l1.wypisz();
cout << "L2 = ";
12.wypisz();
cout<<"Lista L1 bez dwóch ostatnich elementów:\nL1= ";
(l1--)--.wypisz();
cout << "Efekt fuzji L1 z L2:\n";
fuzja(l1,12);
cout << "L1 = ";
    11.wypisz();
cout << "L2 = ";
    12.wypisz();
11.dorzuc2(80);11.dorzuc2(8);
cout << "dorzućmy 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 liście l1: 1
Efekt poszukiwań liczby 0 w liście 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)
dorzućmy do L1 liczby 80 i 8
L1 = -11 1 2 4 4 5 6 7 8 9 77 80
```

5.1.3. Listy jednokierunkowe – teoria i rzeczywistość

Oprócz pięknie brzmiących rozważań teoretycznych istnieje jeszcze twarda rzeczywistość, w której... mają wykonywać się nasze pieczołowicie przygotowane programy⁷.

Spójrzmy obiektywnie na listy jednokierunkowe pod kątem ich wad i zalet (patrz tabela 5-1).

Tabela 5 - 1.
Wady i zalety list
jednokierunkowych.

wady	zalety
nienaturalny dostęp do elementów	małe zużycie pamięci
nielatwe sortowanie	elastyczność

Przeanalizujmy szczególnie uważnie zagadnienie sortowania danych będących elementami listy. Wyobrażamy 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), to 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... Czy należy w takim przypadku dysponować dwiema wersjami tych list – co „pożera” cenną pamięć komputera – czy też może zdecydujemy się na sortowanie listy w pamięci? Jednak uwaga: to drugie rozwiązanie zajmie z kolei cenny czas procesora!

⁷ Wbrew wszelkim przesłankom nie jest to definicja systemu operacyjnego...

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.

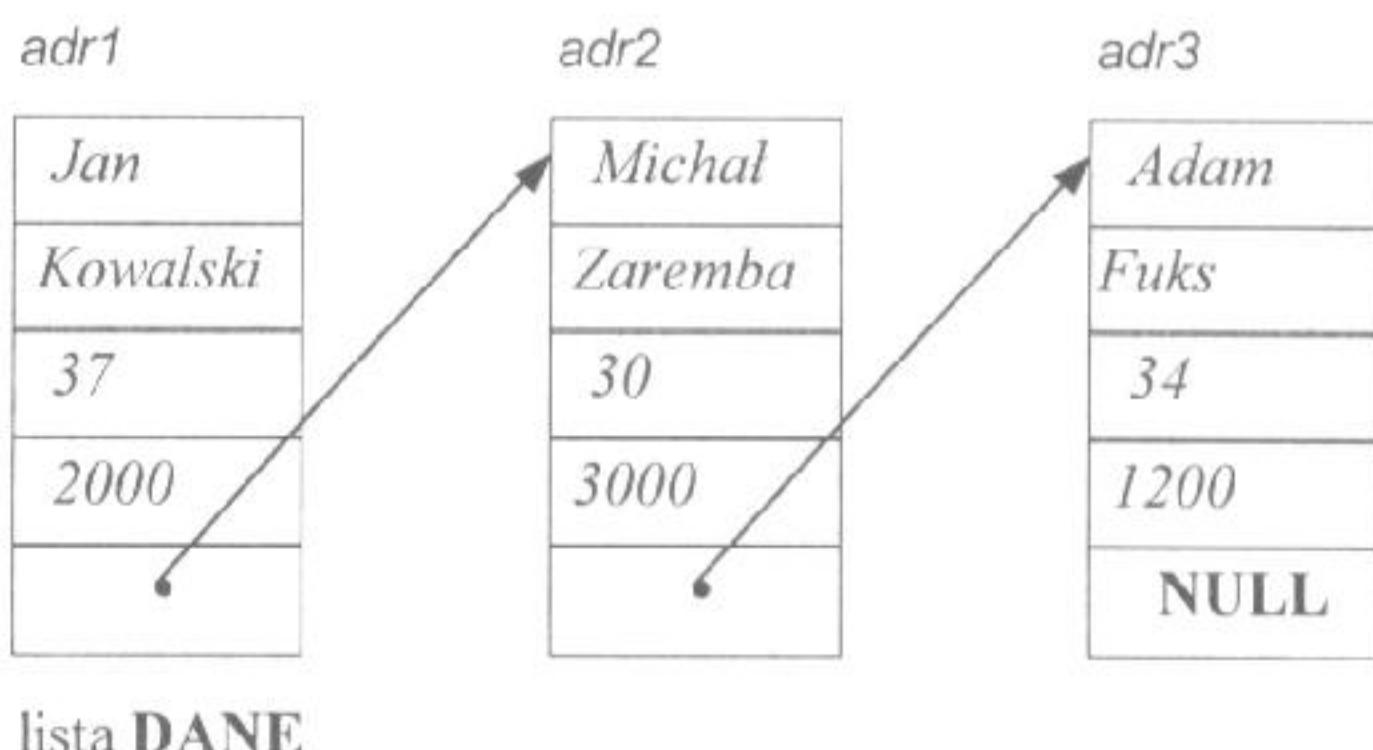
Pomysł polega na uproszczeniu i na skomplikowaniu zarazem tego, co poznaliśmy wcześniej. Uproszczenie polega na tym, że *rekordy zapamiętywane w liście nie są w żaden sposób wstępnie sortowane*. Inaczej mówiąc, do zapamiętywania możemy użyć odpowiednika jakże prostej funkcji *dorzuc1* ze strony 98. Słowo „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 zblizone 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.

Rys. 5 - 8.
Sortowanie listy
bez przemieszcza-
nia jej elementów
(1).

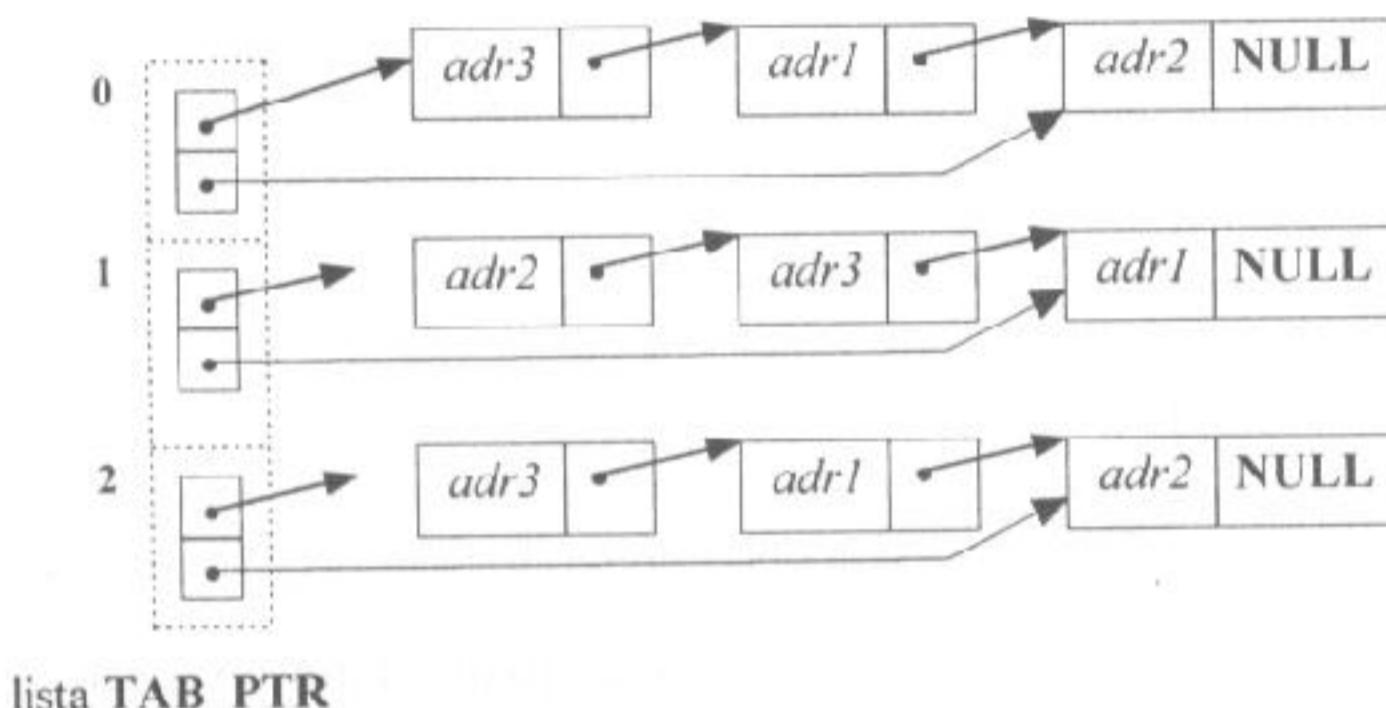


Zawiera on listę o nazwie DANE, zbudowaną z kilku rekordów, które stanowią początek miniaturowej bazy danych o pracownikach pewnego przedsiębiorstwa. Przyjmijmy dla uproszczenia, że jedyne istotne informacje, które chcemy zapamiętać, to: imię, nazwisko, pewien kod i oczywiście zarobek. Na rysunku są 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.

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).

Rys. 5 - 9.
Sortowanie listy
bez przemieszcza-
nia jej elementów
(2).



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 (poznanie już poprzednio), których pola *głowa* i *ogon* umożliwiają dostęp do trzech list wskaźników. Każda z tych list jest posortowana wg innego kryterium sortowania.

Przykładowo 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), podobnie 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 pokaźny objętościowo listing).

Struktury danych prezentują się w nowej wersji następująco:

```
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;             // wskaźników
    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 algorytmów bazowych związanych z listami jednokierunkowymi; jej zastosowanie praktyczne było w związku z tym raczej nikłe.

Obecnie prezentowana wersja struktury listy jednokierunkowej charakteryzuje się bardzo dużą elastycznością użytkowania i to właśnie ona winna służyć jako klasa bazowa w ewentualnej hierarchii dziedziczenia (o ile Czytelnik w istocie będzie w ogóle potrzebował mechanizmów dziedziczenia).

Oto nowa wersja klasy LISTA:

lista2.h

```
const n=5;
const n2=2;                      // ilość kryteriów sortowania
class LISTA
{
public:
    LISTA();                         //konstruktor
    ~LISTA();                        //destruktor
    void dorzuc(ELEMENT *);         // dołącz nowy element q
    void wypisz(char);            // wypisz zawartość listy

    // usuń element, który jest zgodny z wzorcową komórką
    // podaną jako parametr:
    int usun(ELEMENT*, int(*decyzja) (ELEMENT*, ELEMENT*));
private:
typedef struct                   // struktura informacyjna
{                                     // danych
    ELEMENT *glowa;
    ELEMENT *ogon;
}INFO;
```

```

INFO info_dane;           // rekord informacyjny listy danych
typedef struct rob_ptr_inf // struktura informacyjna
// listy wskaźników
{
    LPTR *glowa;
    LPTR *ogon;
}LPTR_INFO;
LPTR_INFO inf_ptr[n2];      // tablica kryteriów (na
                           // rysunku jest to TAB_PTR)
// kilka prywatnych funkcji (omówione w tekście):

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 1200zl”) i wyszukiwania miejsca na wstawienie nowego elementu różniły się nieznacznie. Od tego spostrzeżenia do gotowej realizacji programowej jest już tylko jeden krok. Aby go pokonać, musimy dobrze zrozumieć zasady operowania wskaźnikami do funkcji⁸ w C++, bowiem ich użycie pozwoli na 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 ukłon 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;
}

```

⁸ Miłośnicy i znawcy języka LISP mogą opuścić ten paragraf...

```
int wzor(int x,int(*fun)(int))
{
    return fun(x);
}
void main()
{
    cout << "Wzór 1:" << wzor(10,do_2) << endl;
    cout << "Wzór 2:" << wzor(10,do_4) << endl;
}
```

Funcja *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: nazwa funkcji jest jednocześnie wskaźnikiem do niej. Bezpośrednią konsekwencją jest dość naturalny sposób użycia, pozwalający na uniknięcie typowych dla C++ operatorów * (gwiazdka) i & (operator adresowy).

Inny przykład: 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), które umożliwią nam takie zmodyfikowanie wskaźników, aby cała lista była widziana jako posortowana. W tym celu zmuszeni jesteśmy do użycia pętli *while* poznanej na stronie 101:

```
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: napiszmy 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 strona 112), umawiając się, że pole *glowa* 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 jest/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 przyjmiemy 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:

```

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->glowa=res->ogon=NULL;
    if (inf->glowa==NULL)
        return(res); // lista pusta!
else
{
    LPTR *przed, *pos;
    przed=NULL;
    pos=inf->glowa;
    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 sie w poszukiwaniach
        przed=pos;
        pos=pos->nastepny;
}

```

```
        }
        res->glowa=przed;
        res->ogon=pos;
        return (res);
    }
}
```

- wskaźnik *inf* do struktury informacyjnej listy wskaźników; adres początku znajduje się w polu *glowa*, 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 *decyzja* do funkcji porównawczej, która zostanie włożona do instrukcji *if* w pętli *while*.

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 pozostały kryteriom poszukiwań – wg zarobków, imienia, etc. Jeśli poszukiwanie zakończy się sukcesem, to w polu *ogon* zostanie zwrotny adres fizycznie istniejącego rekordu, który odpowiadał wzorcowi naszych poszukiwań. W przypadku gdyby element taki nie istniał, powinny zostać zwrocone 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 zastosowań funkcji niech będzie dołączanie nowego elementu do listy. Trzeba wówczas stworzyć nowy rekord, prawidłowo wypełnić jego pola i dołączyć na koniec listy danych. 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*, którewróci funkcja *odszukaj_wsk*.

Zastosowanie funkcji *odszukaj_wsk* jest, jak widać, bardzo wszechstronne. Ta ka 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 equal1(ELEMENT *q1, ELEMENT *q2)
{
    // czy rekordy q1 i q2 mają identyczne nazwiska?
    return (strcmp(q1->nazwisko, q2->nazwisko)==0);
}

int equal2(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 porządkowania listy, 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 poszukiwania/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 stronie 97 mieliśmy okazję zapoznać się z funkcją *pusta* informującą, czy lista danych coś zawiera. Nic nie stoi na przeszkodzie, aby do kompletu dodać jej kolejną wersję, badającą w analogiczny sposób listę wskaźników:

```

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 *pusta*, zestaw funkcji decyzyjnych i uniwersalną funkcję *odszukaj_wsk*, możemy pokusić się o napisanie brakującej procedury *dorzuc1*, która będzie służyła do dodawania nowego rekordu do listy danych z jednoczesnym sortowaniem list wskaźników. Założymy, 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)
{
    // rekordłączamy 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;
    }
    //łączamy wskaźnik do rekordu do listy
    // posortowanej alfabetycznie:
    dorzuc2(0,q,alfabetycznie);
    //łączamy 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 101, choć różni się od tamtej doprawdy 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) // lista pusta
    {
        inf_ptr[nr].glowa=inf_ptr[nr].ogon=wsk;
        wsk->nastepny=NULL;
    }else //szukamy miejsca na wstawienie
    {
        LPTR *przed,*po;
        LPTR_INFO *gdzie;
        gdzie=odszukaj_wsk(&inf_ptr[nr],q,decyzja);
        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<n2; 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 gwoli 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 co 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 nie odnaleziony
        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;
    }
    przed->nastepny=q->nastepny;
    delete q;
    return 1;
}

```

```

    }
    if (pos!=q)
        return(0);           // element nie odnaleziony?
    if (pos==info_dane.glowa) // usuwamy z poczatku listy
    {
        info_dane.glowa=pos->nastepny;
        delete pos;
    }else
        if(pos->nastepny==NULL) // usuwamy z konca 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; szczegółowy przykład zastosowania może mieć zatem 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
    // wypisanie wskaźników (są to adresy), lecz
    // informacji na które one wskazują
    LPTR *q=inf->glowa;
    while (q != NULL)
    {
        cout << setw(9)<<q->adres->nazwisko<< " zarabia "
            << setw(4)<<q->adres->zarobek<<"zl \n";
        q=q->nastepny;
    }
}

```

```
    }
    cout << "\n";
}

void LISTA::wypisz(char kryterium)
{
    if (kryterium == 'a') // alfabetycznie
        wypisz1(&inf_ptr[0]);
    else
        wypisz1(&inf_ptr[1]);
}
```

Funkcja *main* testuje wszystkie nowo poznane mechanizmy:

```
void 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 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 alfabetyczne ***\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,equal2) <<endl;
    delete f;
    cout<< "*** Baza danych posortowana alfabetyczne ***\n";
    l1.wypisz('a');
    cout<< "*** Baza danych posortowana wg zarobków ***\n";
    l1.wypisz('z');
}
```

Uruchomienie programu powinno dać następujące wyniki:

```
*** Baza danych posortowana alfabetyczne ***
    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 alfabetyczne ***
    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
```

5.2. Tablicowa implementacja list

Programowanie w C++ zmusza niejako programistę do dobrego poznania operacji na dynamicznych strukturach danych, sprawnego żonglowania wskaźnikami etc. To jest uwaga natury ogólnej, natomiast trzeba zauważać również, iż 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 *rzecz jasna* ponosi „chłopiec do bicia”, czyli sam język programowania. Tymczasem, podobnie zresztą jak i w życiu, to samo można zrobić wieloma sposobami – 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 1000 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 1000! Jest to jednak cena nie do uniknięcia, placimy ją za prostotę realizacji.

5.2.1. Klasyczna reprezentacja tablicowa

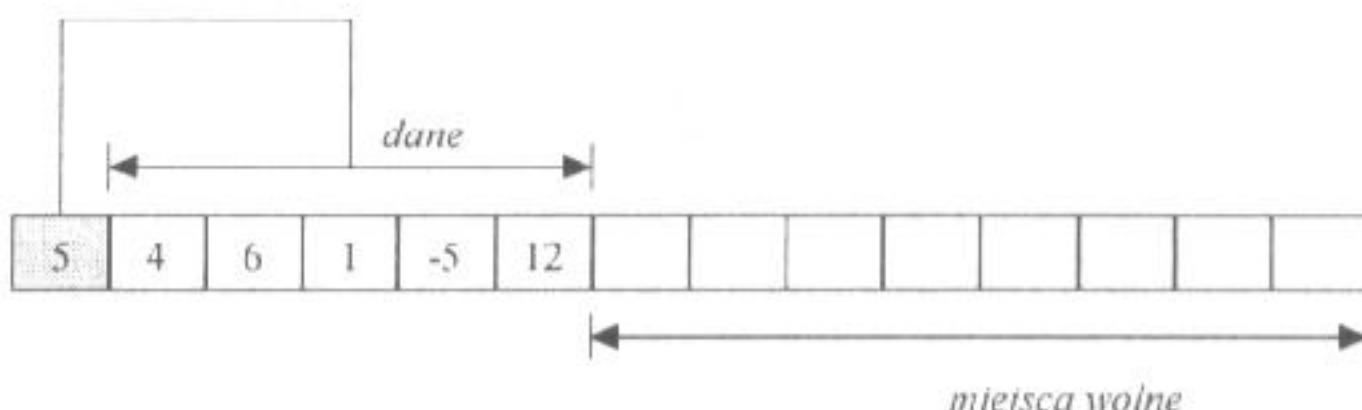
Jedną z najprostszych metod zamiany tablicy na listę jest umówienie się co do sposobu interpretacji jej zawartości. Jeśli powiemy sobie głośno (i nie zapomnimy zbyt szybko o tym), że *i*-temu indeksowi tablicy będzie odpowiadać *i*-ty element listy, to problem mamy prawie z głowy. To „prawie” wynika z tego, że trzeba się umówić, ile maksymalnie elementów zechcemy zapamiętać na liście. Oprócz

tego konieczne będzie wybranie jakiejś zmiennej do zapamiętywania aktualnej ilości 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:

Rys. 5 - 10.

Tablicowa
implementacja
listy.



Programowa realizacja jest bardzo prosta – deklaracja klasy nie zawiera żadnych niespodzianek:

lista_tab.cpp

```
const int MaxTab=201;           // 200 możliwych elementów
class ListaTab
{
    int tab[MaxTab];           // tab[0] zarezerwowane!
public:
    ListaTab()                // konstruktor klasy
    {tab[0]=0;}                // usuń element z pozycji k
    void UsunElement(int k);   // wstaw element x na koniec listy
    void WstawElement(int x);  // wstaw element x na pozycję k;
    void WstawElement(int x, int k);
    void WypiszListe();
};
```

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 potrzeb będzie mógł je sobie stworzyć.

Co jednak z *dolaczaniem* 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 element x na koniec listy
    if(tab[0]<MaxTab-1)
        tab[++tab[0]]=x;
    }

void ListaTab::WstawElement(int x,int k)
//wstawiamy element x 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 „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 według jakiegoś innego klucza – Czytelnik może odpowiednio rozbudować funkcje i metody w ramach nieskomplikowanego ćwiczenia.

5.2.2. Metoda tablic równoległych

W poprzednio poznanej implementacji list przy pomocy zwykłej tablicy przypisaliśmy na sztywno i -temu elementowi tablicy i -ty element listy. W prostych zastosowaniach może to wystarczyć w zupełności, jednak rozwiązanie takie o wiele bardziej jest 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 nieuchronne?

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 *następny*, które służy do odszukiwania elementu „następnego” na liście. Dobrze znane i klasyczne wręcz rozwiązań. 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”.

Rys. 5 - 11.

Metoda „tablic równoległych” (1).

int następny;
char info;

rekord bazowy

3	5	4	2	1	-1	?	?
E	O	K	T	K	?	?	?

Przykładowa tablica rekordów z danymi

Pierwszy element tablicy (tzn. ten z pozycji 0) pełni rolę wskaźnika początku listy. Jest to zatem zmienna typu *głowa*. Jeśli oznaczmy tablicę jako *t*, to *t[0].następny* 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 następnie, co następuje po ‘K’, musimy odczytać *t[3].następny*. Jest to 2 i tam też jest umieszczona kolejna litera słowa „KOTEK” – etc. Koniec listy jest zaznaczany umownie poprzez wartość -1 w polu *następny*.

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 (patrz 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 formułując, dobrze by było dysponować osobną tablicą na dane i osobną na wskaźniki. Dlaczego jednak nie pójść dalej i nie używać kilku tablic na wskaźniki?! Zbliżyłybyśmy się wówczas do wersji zaprezentowanej na rysunku 5 - 8, otrzymując jednak o wiele prostsze w realizacji zadanie.

Na rysunku 5 - 12 jest przedstawiona mini-baza danych zgrupowana w wyodrębnionej tablicy danych.

Rys. 5 - 12.

Metoda „tablic równoległych” (2).

	DANE			L1	L2	L3
	0	1	2			
0						
1						
2	Kowalski	37	2000	3	1	3
3	Zaremba	30	3000	1	4	1
4	Fuks	34	1200	2	3	4
5						
6						
7						

Obok tablicy danych możemy zauważać 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 począwszy od komórki *dane[2]* w góre. 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 zauważyc, 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.

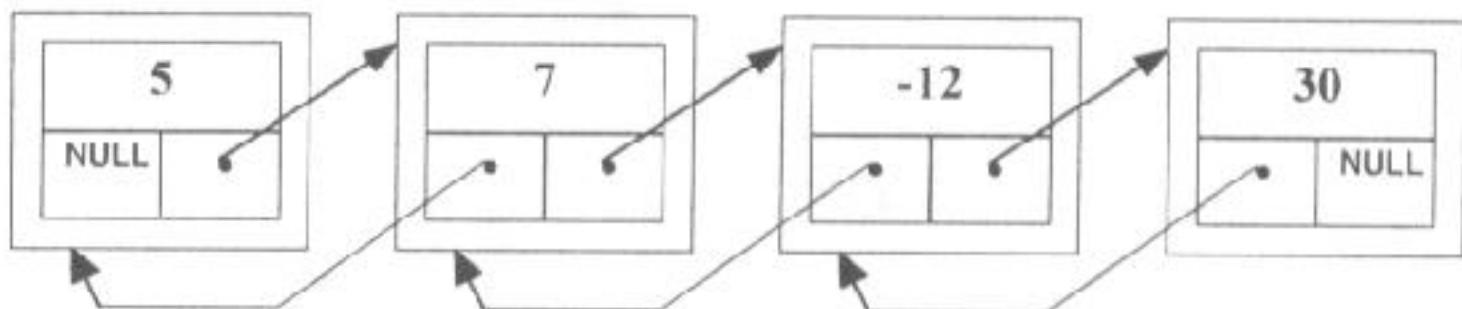
Analogicznie postępując możemy „odkryć”, że *L2* jest listą posortowaną wg kodów 2-cyfrowych, a *L3* – wg zarobków. Tablicowa reprezentacja list, w której nastąpiło oddzielenie danych od wskaźników, pozwala na zapamiętanie w tym samym obszarze pamięci kilku list jednocześnie – o ile oczywiście ich elementy 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 (patrz np. klasy z grupy *TArray* w systemie Borland C++). Samodzielne zdefiniowanie takiej klasy jest jednak czasochłonne – zwłaszcza jeśli zamierzamy zadbać o jej uniwersalność.

5.2.3. Listy innych typów

Listy jednokierunkowe są bardzo wygodne w stosowaniu i zajmują stosunkowo mało pamięci. Tym niemniej operacje na nich niekiedy zajmują dużo czasu. Zauważyło ten fakt sporo ludzi i tym sposobem zostały wymyślone inne typy list, np.:

lista dwukierunkowa – komórka robocza zawiera wskaźniki do elementów: *poprzedniego* i *następnego*:

Rys. 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 *nastepny*. Lista dwukierunkowa jest dość „kosztowna”, jeśli chodzi o zajętość pamięci, tym niemniej czasami ważniejsza jest szybkość działania od ewentualnych strat pamięci.

Struktura wewnętrzna listy dwukierunkowej jest oczywista:

```

typedef struct rob
{
    int wartosc;
    struct rob *nastepny;
    struct rob *poprzedni;
} ELEMENT;
  
```

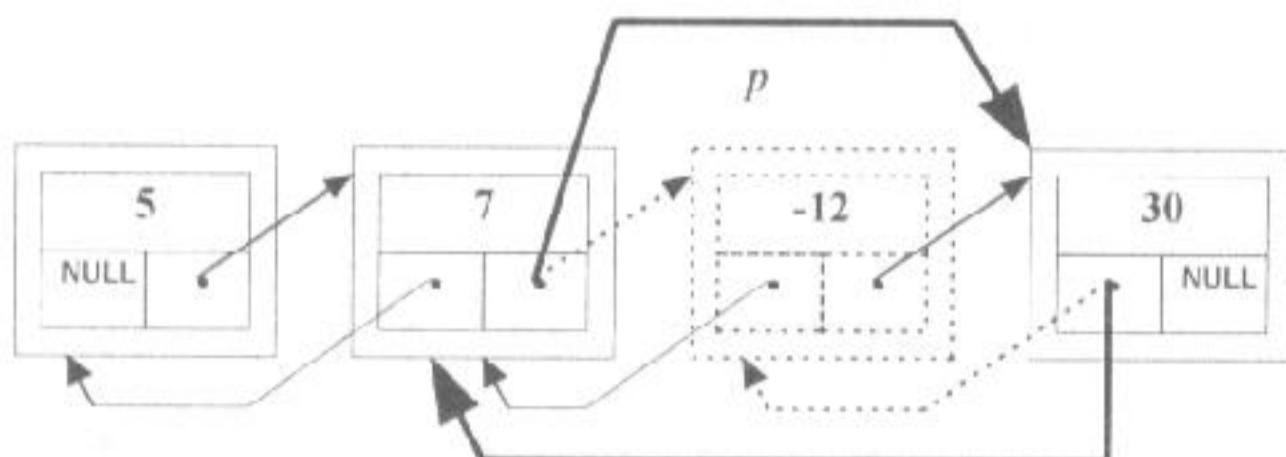
Załóżmy teraz, że podczas przeglądania elementów listy zapamiętaliśmy wskaźnik pozycji bieżącej *p*. (Przykładowo szukaliśmy elementu spełniającego pewien warunek i na wskaźniku *p* nasze poszukiwania zakończyły się sukcesem). Jak usunąć element *p* z listy? Jak pamiętamy z paragrafów poprzednich, do 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):

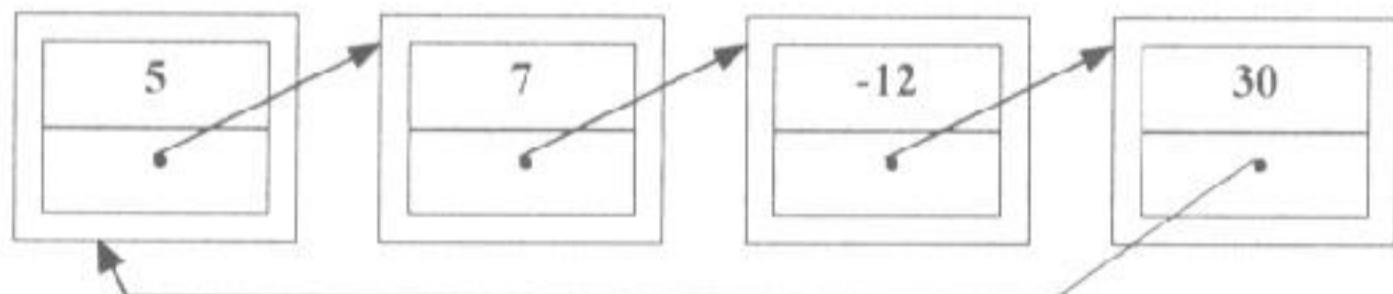
Rys. 5 - 14.
Usuwanie danych
z listy dwukierunkowej.



lista cykliczna – patrz rysunek 5 - 15 – jest zamknięta w pierścień; wskaźnik ostatniego elementu wskazuje „pierwszy” element.

- Pewien element określany jest jako „pierwszy” raczej umownie i służy wyłącznie do wejścia w „magiczny krąg” wskaźników listy cyklicznej...

Rys. 5 - 15.
Lista cykliczna.



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.

5.3. Stos

Stos jest kluczową strukturą danych w informatyce. To zdanie brzmi bardzo groźnie, lecz chciałbym zapewnić, że nie kryje się za nim nic strasznego. Krótko mówiąc jest to struktura danych, która ułatwia rozwiązywanie wielu problemów natury algorytmicznej i w tę właśnie stronę wspólnie będziemy zdążać. Zanim dojdziemy do zastosowań stosu, spróbujmy go jednak zaimplementować w C++!

5.3.1. 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 przy pomocy

angielskiego skrótu **LIFO**: Last-In-First-Out, co w wolnym tłumaczeniu oznacza „ostatni będą pierwszymi”. Do odkładania danych na wierzchołek stosu służy zwykłowo funkcja o nazwie *push(X)*, gdzie *X* jest daną pewnego typu. Może to być dowolna zmienna prosta lub złożona: liczba, znak, rekord...

Podobnie, aby pobrać element ze stosu, używa się funkcji o nazwie *pop(X)*, która załadowuje zmienną *X* daną zjętą z wierzchołka stosu. Obie te podstawowe funkcje oprócz swojego głównego zadania, które zostało wzmiankowane wyżej, zwracają jeszcze kod błędu¹. Jest to stała typu całkowitego, która informuje programistę, czy czasem nie nastąpiła sytuacja anormalna, 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 dwiema 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.

Dla dokładniejszego zobrazowania zasady działania stosu proszę prześledzić kilka operacji dokonanych na nim i efekt ich działania – patrz rysunek 5 - 16.

*Rys. 5 - 16.
Słos i podstawowe
operacje na nim.*

Courier	= kod programu
<i>pochylona kursywa</i>	= komentarz
<code>s=pop(c); s=push('A'); s=push('L');</code>	<code>s=pop(c); s=push('B'); s=push('C');</code>
<i>s=StosPusty</i>	<i>s=OK</i>
<i>c=??</i>	<i>c='L'</i>

Rysunek przedstawia stos służący do zapamiętywania znaków. Stałe symboliczne *StosPusty*, *OK* i *StosPelny* są zdefiniowane przez programistę w module zawierającym deklarację stosu. Wyrażają się one w wartościach typu *int* (co akurat nic ma specjalnego znaczenia dla samego stosu...). Nasz stos ma pojemność dwóch elementów, co jest oczywiście absurdalne, ale zostało przyjęte na użytek naszego przykładu, aby zilustrować efekt przepełnienia.

¹ Nie jest to bynajmniej obowiązkowe!

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 błędą.

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 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 klasy², 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=300;
const int STOS_PELNY=3;
const int STOS_PUSTY=2;
const int OK=1;
template <class TypPodst> class STOS
{
    TypPodst t[DLUGOSC_MAX+1];
    stos=t[0]...t[DLUGOSC_MAX]
    int szczyt; // szczyt = pierwsza WOLNA komórka
public:
    STOS() { szczyt=0; } // konstruktor
    void clear() { szczyt=0; } // zerowanie stosu
    int push(TypPodst x);
    int pop (TypPodst &w);
    int StanStosu();
}; // koniec definicji klasy STOS

```

Nasz stos będzie mógł potencjalnie służyć do przechowywania danych wszelkiego rodzaju, z tego też powodu celowe wydało się zadeklarowanie go w postaci tzw. *klasy szablonowej*, co zostało zaznaczone przez słowo kluczowe *template*.

² Czyli dokonamy tzw. *hermetyzacji*.

Idea klasy szablonowej polega na stworzeniu wzorcowego kodu, w którym typ pewnych danych (zmiennych, wartości zwracanych przez funkcje...) 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 w 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, to otrzymujemy tego rodzaju dziwolągi³:

```
template <class TypPodst> int STOS<TypPodst>::  
push(TypPodst x)  
{  
// element x zostanie położony na stos  
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* (jaki by nie był jego typ) jest zapisywany na szczyt 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 zapełniony!

Metoda *pop* wykonuje odwrotne zadanie, zdejmowany ze stosu element jest zapamiętywany w zmiennej *w* (przekazanej w wywołaniu przez referencję); zmiana *szczyt* jest oczywiście dekrementowana – pod warunkiem że stos nie był pusty (z próżnego to nawet i programista nie... naleje?):

```
template <class TypPodst> int STOS<TypPodst>::  
pop(TypPodst &w)
```

³ Oczywiście, zawsze można się pocieszać, że ewentualnie mogłyby 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, które wchodzą w grę podczas programowania, stąd też każdy zabieg, który służy uczynienia 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 obiektywym*...).

```

{
// "w" zostanie "załadowane" wartością zdjętą ze stosu
if (szczyt>0)
{
    w=t[--szczyt];
    return (OK);
} else
    return (STOS_PUSTY);
}

```

Od czasu do czasu może zajść potrzeba zbadania stanu stosu bez wykonywania na nim żadnych operacji. Użyteczna może być wówczas następująca funkcja:

```

template <class TypPodst> int STOS<TypPodst>::  

StanStosu()  

{  

// zwraca informację o stanie stosu  

switch(szczyt)  

{  

    case 0 :return (STOS_PUSTY);  

    case DLUGOSC_MAX+1 :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: *STOS<char*> s* – podobnie dzieje się w przypadku każdego innego typu danych:

stos.cpp

```

#include "stos.h"  

#include <iostream.h>  
  

char* tab1[3]={"ala","ma","kota"};  

float tab2[3]={3.14, 2.12, 100};  
  

void main()
{
// deklarujemy jeden stos do przechowywania tekstów:  

STOS<char*> s1;
// deklarujemy jeden stos do przechowywania liczb:  

STOS<float> s2;
cout << "Odkładam na 1 stos: ";
for(int i=0; i<3; i++)
{
}

```

```
    cout << tab1[i] << " ";
    s1.push(tab1[i]);
}
cout << "\nOdkładam na 2 stos: ";
for(i=0; i<3;i++)
{
    cout << tab2[i] << " ";
    s2.push(tab2[i]);
}
for(i=0; i<3;i++)
{
    char *z;
    float f;
    s1.pop(z);
    s2.pop(f);
    cout << "\nZdejmuję parami dane ze stosów: (" 
        << z << ", "<<f<<") \n";
}
```

Oto wyniki naszego programu:

```
Odkładam na 1 stos: ala ma kota
Odkładam na 2 stos: 3.14 2.12 100
Zdejmuję parami dane ze stosów: (kota,100)
Zdejmuję parami dane ze stosów: (ma,2.12)
Zdejmuję parami dane ze stosów: (ala,3.14)
```

5.4. Kolejki FIFO

Kolejki typu *FIFO* (ang. **F**irst **I**n **F**irst **O**ut, co w wolnym tłumaczeniu oznacza: *kto był pierwszy, ten i pierwszym pozostanie¹*), będą kolejnym omawianym typem danych. 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.

¹ Zasada obsługi ogonka ludzi przed kasą sklepową.

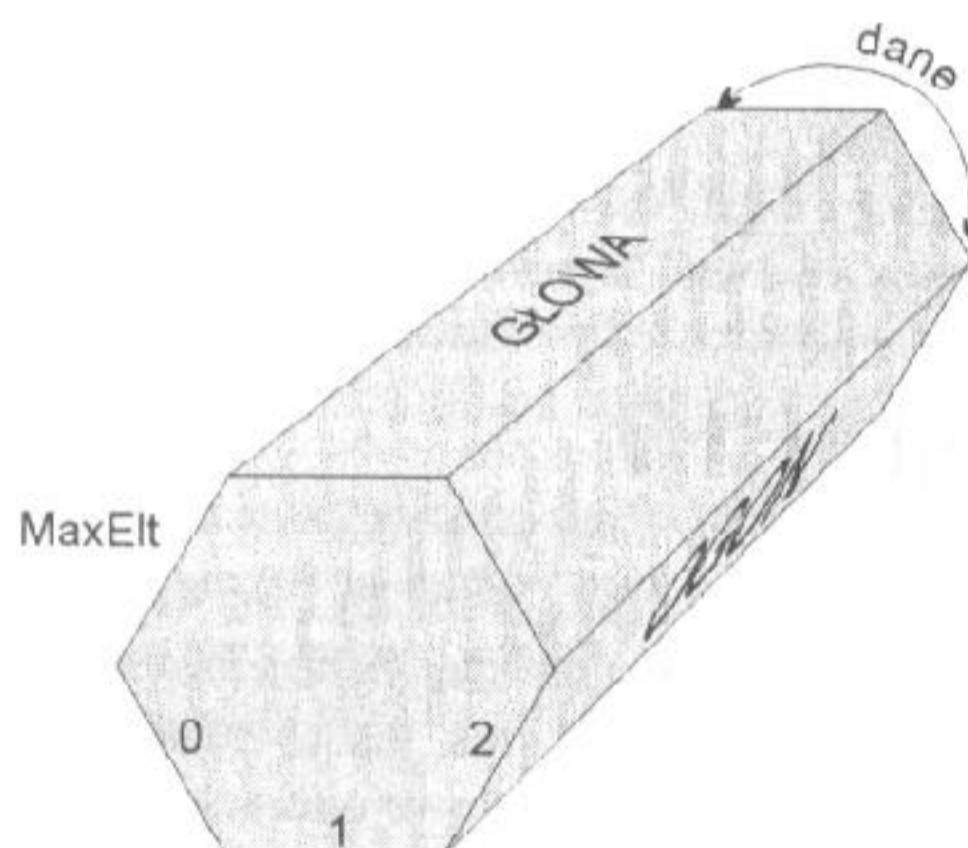
Jak to zwykle bywa, możliwych implementacji kolejek jest co najmniej kilka. Realizacja efektywna „czasowo” za pomocą list jednokierunkowych jest zbliżona do tej wzmiankowanej przy okazji omawiania stosu. Nie będzie ona stanowiła przedmiotu naszej dyskusji, ograniczymy się jedynie do prostej implementacji tablicowej (zbliżonej zresztą ideowo do tablicowej realizacji stosu).

Tablicowa implementacja kolejki *FIFO* jest wyjaśniona na rysunku 5 - 17.

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 zawińie 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 umówić ponadto jak interpretować stwierdzenie, że kolejka jest pusta?

Rys. 5 - 17.
Tablicowa realiza-
cja kolejki *FIFO*.



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

```
template <class TypPodst> class FIFO
{
    TypPodst *t;
    int głowa, ogon, MaxElt;
```

```

public:

FIFO(int n)
{
    // konstruktor kolejki o rozmiarze n
    MaxElt=n;
    glowa=ogon=0;
    t=new TypPodst[MaxElt+1]; // przydziął pamięci
}

void wstaw(TypPodst x)
{
    // wstawia nowy element x do kolejki
    t[ogon++]=x;
    if(ogon>MaxElt)
        ogon=0;
}

int obsluz(TypPodst &w)
{
    // obsługuje 1-go klienta z kolejki
    if (glowa==ogon) // kolejka pusta
return -1;          // informacja o błędzie operacji
    w=t[glowa++];
    if(glowa>MaxElt)
        glowa=0;
    return 1;           // sukces operacji
}
int pusta()          // czy kolejka jest pusta?
{
    if (glowa==ogon)
        return 1;      // kolejka pusta
    else
        return 0;      // coś jest w kolejce
}
};

```

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 korzystanie w praktyce z nowej struktury danych
kolejka.cpp

```

#include <iostream.h>
#include "kolejka.h"

static char *tab[]=
{"Kowalska", "Fronczak", "Becki", "Pigwa"};
void main()
{
    FIFO<char*> kolejka(5); // kolejka 5-osobowa
}

```

```

for(int i=0; i<4;i++)
    kolejka.wstaw(tab[i]);
for(i=0; i<5;i++)
{
    char *s;
    int res=kolejka.obsluz(s);
    if (res==1)
        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 zjawiły się na samym początku. Tak też jest w naszym przykładzie, o czym najdubitniej świadczą wyniki wykonania programu:

```

Obsłużony klient:Kowalska
Obsłużony klient:Fronczak
Obsłużony klient:Becki
Obsłużony klient:Pigwa
Kolejka pusta!

```

5.5. Sterty i kolejki priorytetowe

W paragrafach poprzednich mieliśmy okazję zapoznać się m.in. z dwiema strukturami danych stanowiącymi swoje skrajności „ideowe”:

- *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:

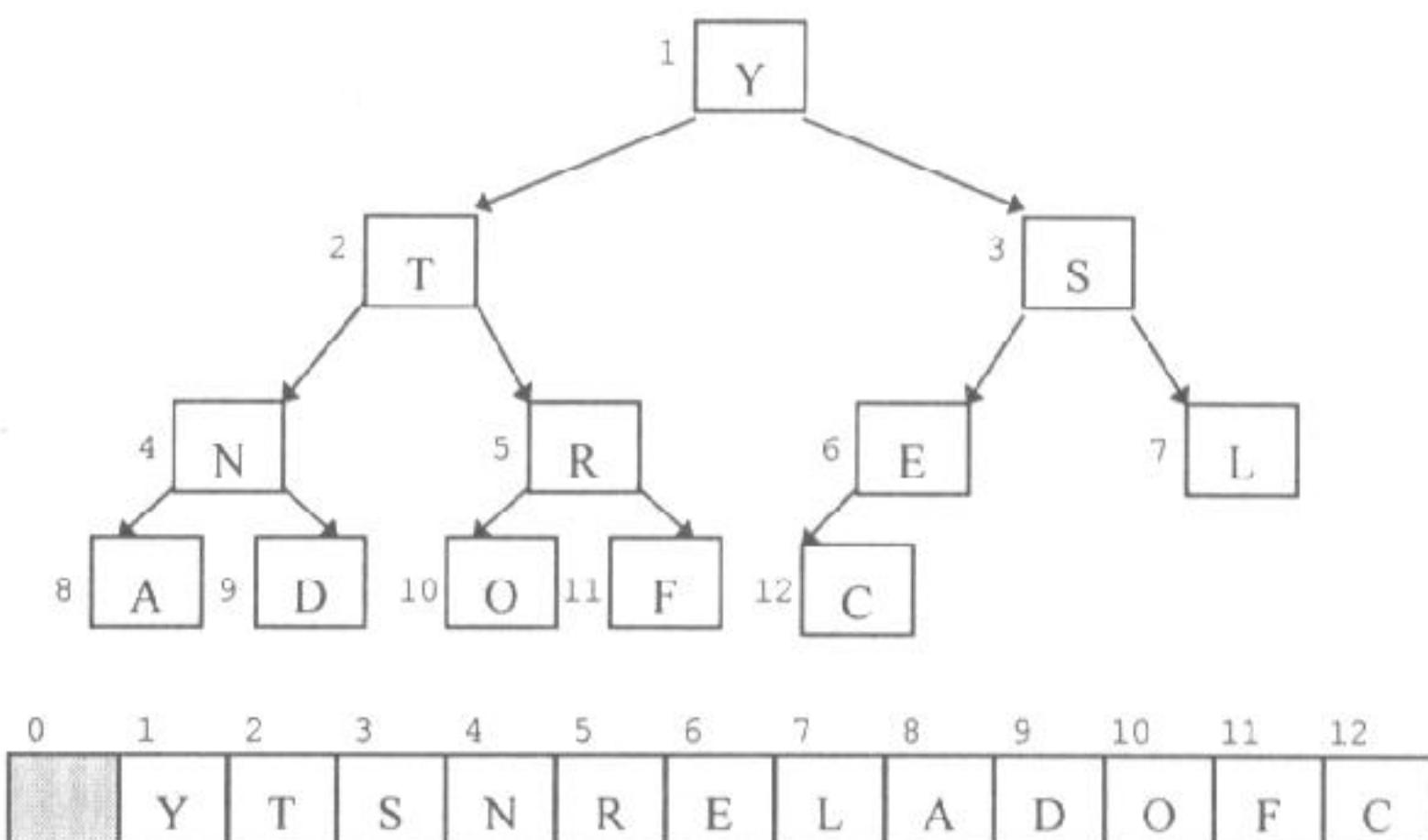
- zwykłym wstawianiu nowego elementu;
- usuwaniu największego elementu¹.

¹ 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

Jednym z najłatwiejszych sposobów realizacji kolejek priorytetowych jest użycie struktury danych zwanej *stertą*². 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 „dowieszamy” maksymalnie po dwa nowe aż do ulokowania wszystkich 12-elementów. Można to oczywiście wyrazić nieco bardziej formalnie, ale zapewniam, że zdecydowanie mniej zrozumiale.

Rys. 5 - 18.
Sterta i jej tablicowa implementacja.



Liniowy porządek wypełniania drzewa automatycznie sugeruje sposób jego składowania w tablicy³:

- „wierzchołek” (czyli *de facto* korzeń, bo drzewo jest odwrócone)=1;
- „lewy” potomek i -tego węzła jest „schowany” pod indeksem $2*i$;

ujemnej. W naszych przykładach dla prostoty ograniczymy się tylko do przypadku składowania liczb całkowitych.

² Ang. *heap* – inna spotykana polska nazwa to *stóg*.

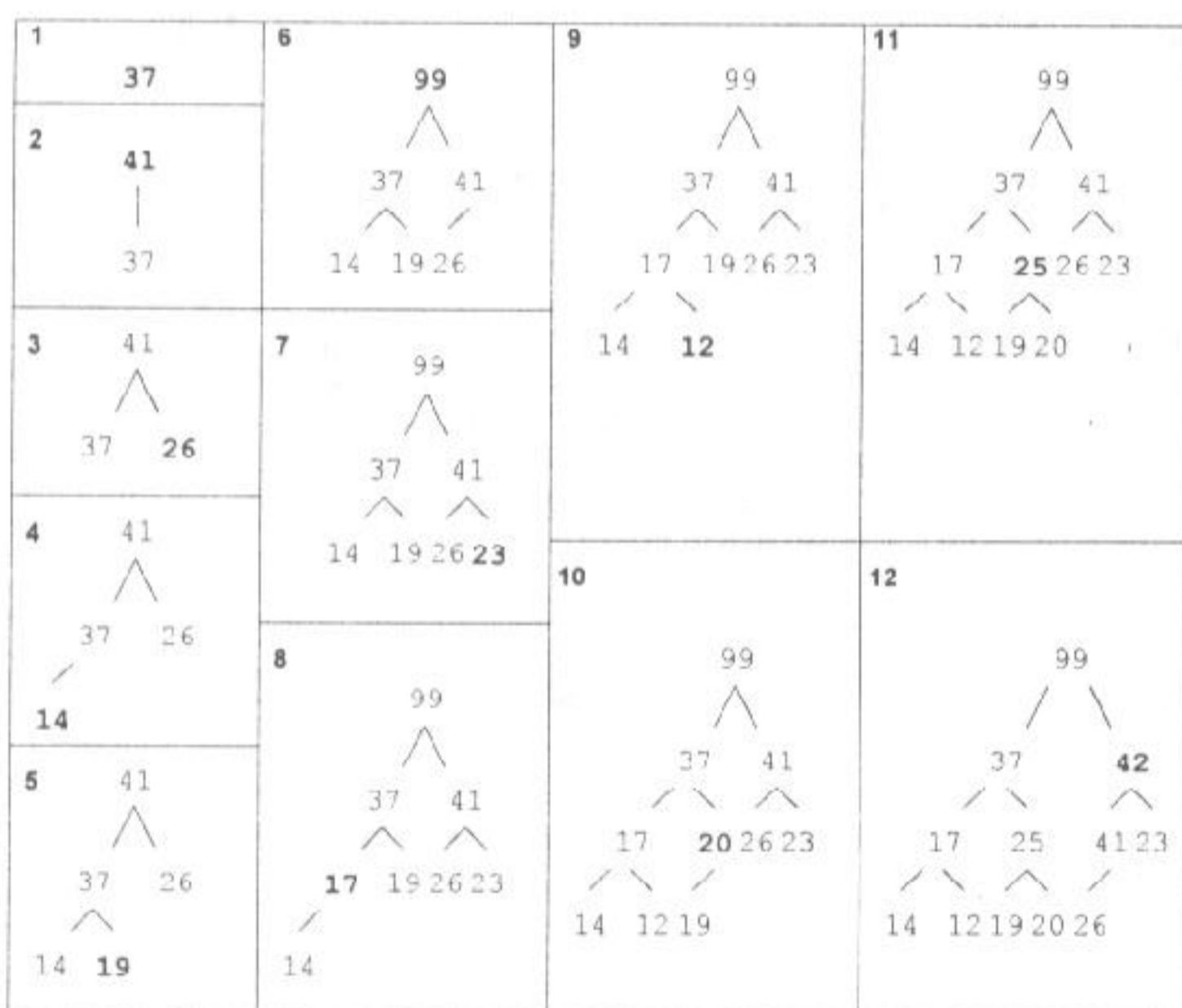
³ Zerowa komórka tablicy nie jest używana do składowania danych.

- „prawy” potomek i -tego węzła jest „schowany” pod indeksem $2*i+1$.

Uwaga: dany węzeł może mieć od 0 do 2 potomków.

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 większa⁴ od wartości węzłów jego dwóch potomków – jeśli oczywiście istnieją. Sposób organizacji drzewa (jak również w konsekwencji tablicy) ułatwia operacje wstawiania i usuwania elementów. Możemy bowiem nowy element bez problemu „dopisać” na koniec tablicy (co oczywiście zburzy nam ład wcześniej tam panujący), następnie za pomocą dość prostych modyfikacji tablicy przywrócić z powrotem tablicy (drzewu) własności sterty. Patrzmy na przykładzie, w jaki sposób jest konstruowana sterta ze zbioru elementów: 37, 41, 26, 14, 19, 99, 23, 17, 12, 20, 25 i 42 – dodłączanych sukcesywnie do drzewa. Cały proces jest pokazany na rysunku 5 - 19.

Rys. 5 - 19.
Konstrukcja sterty
na przykładzie.



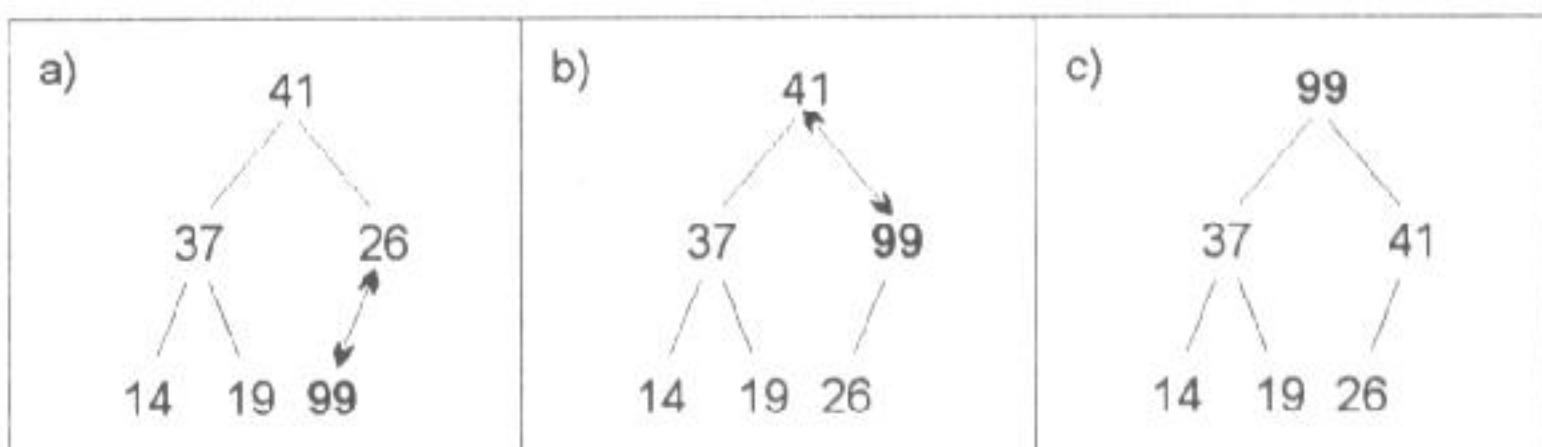
Na rysunku tym widzimy gdzie wędruje nowy – zaznaczony wytłuszczoną czcionką – element. Poprzez porównanie z etapem poprzednim łatwo zauważamy modyfikacje struktury drzewa. Założymy, że dokładamy na koniec drzewa

⁴ Spotyka się również implementacje, w których jest to wartość nie większa.

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 sterły: potomek węzła jest większy co do wartości niż sam węzeł, do którego jest on „przywieszony”! Co możemy zrobić, aby przywrócić porządek? W tym miejscu wystarczy zwyczajnie wymienić 26 i 99 miejscami, aby wszystko się *lokalnie* „uspokoilo”. Zauważmy, że taka lokalna zamiana przywraca porządek jedynie na aktualnie analizowanym poziomie – burząc go może na następnym! Zatem aby w całej stercie zapanował porządek, należy proces zamieniania kontynuować⁵ w górę aż do osiągnięcia „korzenia”. (W naszym przykładzie konieczna będzie jeszcze zamiana liczb 99 i 41). Programową realizację opisanej powyżej czynności wykona procedura o nazwie *DoGory*. Opisaną sytuację ilustruje rysunek 5 - 20.

Teraz, gdy już wiemy CO to jest sterła i JAK się ją tworzy, pora wyjaśnić wreszcie dlaczego sterła umożliwia łatwe tworzenie kolejek priorytetowych. W §5.5. wymieniliśmy istotną cechę wyróżniającą kolejki priorytetowe od innych podobnych struktur danych: pierwszym obsługiwany „klientem” jest

Rys. 5 - 20.
Poprawne wstawianie nowego elementu do sterły.



ten, który ma największą wartość (lub też w przypadku rekordów największą wartość pewnego wybranego pola). Jeśli trzymać się ciągle analogii kolejki do kasy sklepowej, to można by powiedzieć, że wszyscy ustawiają się elegancko na koniec „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 – należały w tym celu dokonać przeszukiwania, 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)

⁵ Jeśli zachodzi oczywiście potrzeba.

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 stercie 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 wstępnie warto wreszcie zaprezentować kilka linii kodu w C++, które lepiej przemówią niż przewlekłe wyjaśnienia. Definicja klasy *Sterta*⁶ jest następująca:

```
stertha
```

```
class Sterta
{
private:
    int *t;
    int L; // ilość elementów
public:
    Sterta(int nMax) // prosty konstruktor
    {
        t=new int[nMax+1];
        L=0;
    }
    void wstaw(int x);
    int obsluz();
    void DoGory();
    void NaDol();
    void pisz();
}; // koniec definicji klasy Sterta
```

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 szczegół 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.

⁶ Aby nie rozwlekać kodu nadmiernym generalizowaniem, podany zostanie przykład dla sterty liczb całkowitych.

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” w porównaniu z oryginalnym algorytmem polegającym na systematycznym zamienianiu elementów ze sobą (w miarę potrzeby) podczas przechodzenia przez węzły drzewa, jednak pozwala ona nieco przyspieszyć procedurę⁷.

Nawiązując do kolejek priorytetowych wspomnialiśmy, 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 obsługiwany „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).

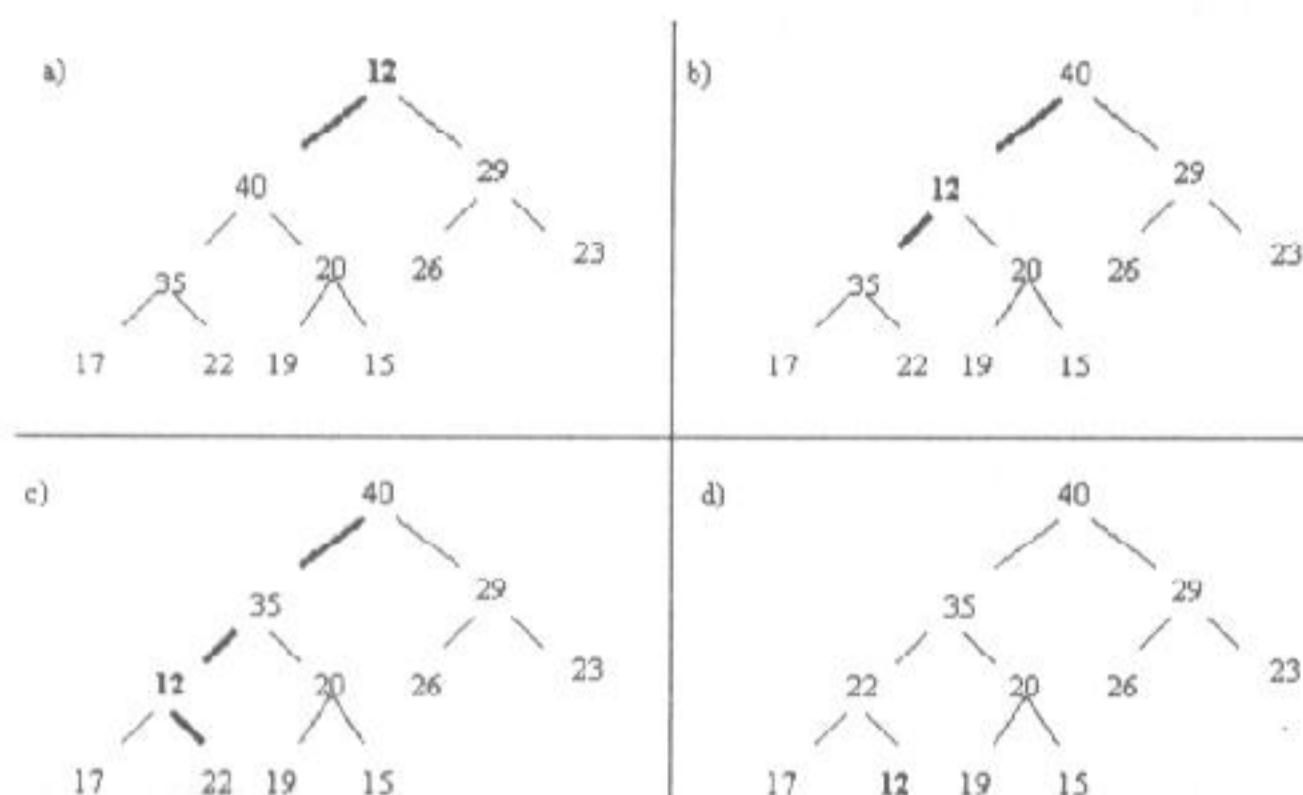
Jak powinna działać procedura *NaDol*? Zmiana wartości w korzeniu mogła zaburzyć spokój zarówno w lewym, jak i prawym jego potomku. Nowy korzeń należy przy pomocy zamiany z większym z jego potomków przesiąć w dół drzewa

⁷ Która oczywiście pozostanie w dalszym ciągu „logarytmiczna” – cudów bowiem w informatyce nie ma!

aż do momentu znalezienia właściwego dlań miejsca. Popatrzmy na efekt działania procedury *NaDol* wykonanej na pewnej stercie (patrz rysunek 5 - 21).

Rys. 5 - 21.

Ilustracja procedury *NaDol*.



Element 12 zostało zaznaczony wyłuszczoną czcionką. Za pomocą pogrubionej kreski zaprezentowano drogę, po której zstępował element 12 w stronę swojego... miejsca ostatecznego spoczynku!

Oto jak można sposób zrealizować procedurę *NaDol*:

```
void Sterta::NaDol()
{
    int i=1;
    while(1)
    {
        int p=2*i;      // lewy potomek węzła "i": p, prawy: 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 efektownego zastosowania sterty do... sortowania danych.

Wystarczy bowiem dowolną tablicę do posortowania wpierw zapamiętać w stercie używając metody *wstaw*, a następnie zapisać ją „od tyłu” w miarę obsługiwania przy pomocy metody *obsluz*:

```
#include "sterta.h"
int tab[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
void main()
{
    Sterta s(14);
    for (int 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 algorytmu *Quicksort* (patrz opis algorytmu *Quicksort*).

Powyższa procedura może być jeszcze bardziej przyspieszona poprzez wyłączenie kodu metod *wstaw* i *obsluz* 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 do zapłacenia 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).

5.6. 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 na temat 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

studia 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.

Nasze rozważania zaczniemy od najpopularniejszych i najczęściej używanych *drzew binarnych*, których użyteczność do rozwiązywania przeróżnych zagadnień algorytmicznych jest niezaprzeczalna.

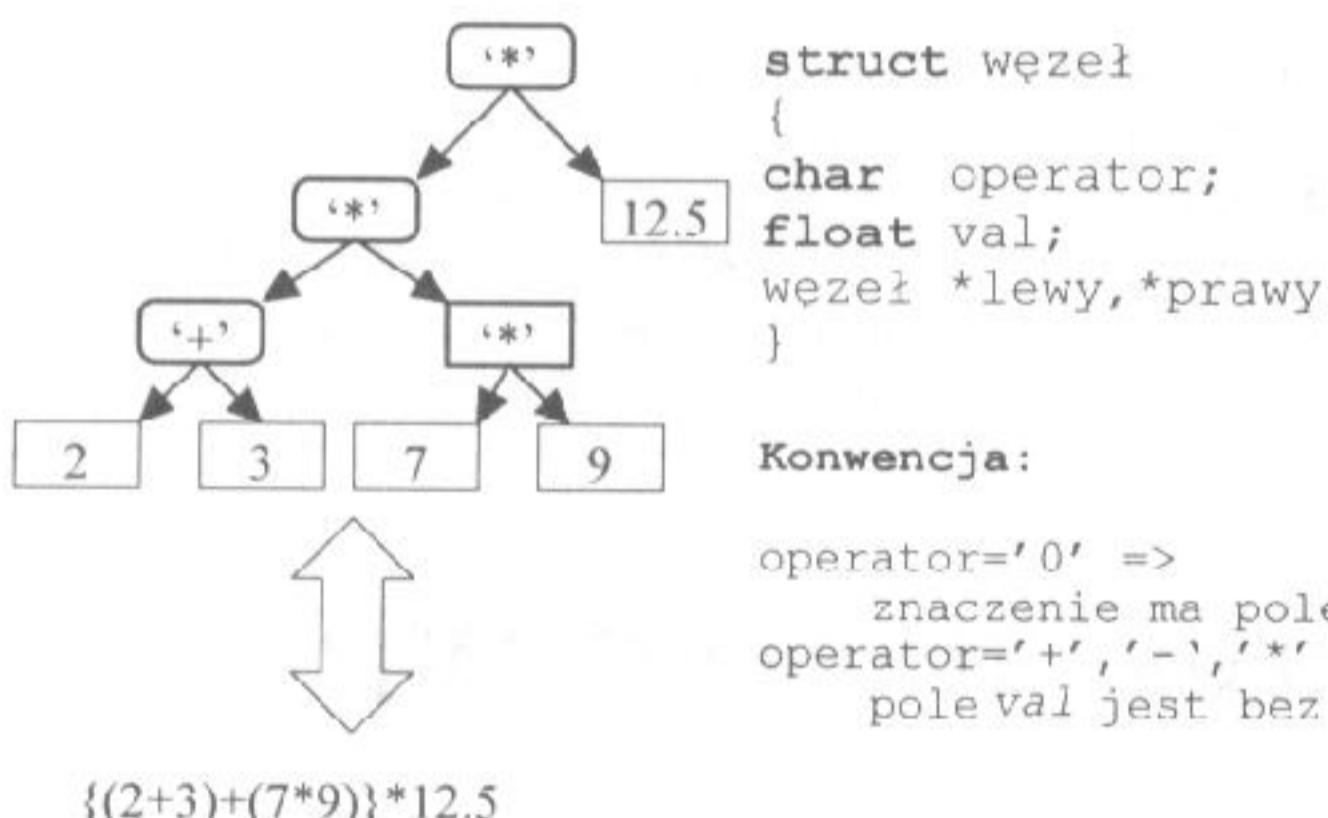
Co to są zatem drzewa binarne? Są to struktury bardzo podobne do list jednokierunkowych, ale wzbogacone o jeszcze jeden *wymiar* (lub *kierunek* jak kto woli...).

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. Aby dobrze zrozumieć sposób działania i użyteczność drzew binarnych, popatrzmy na rysunek 5 - 22.

Rys. 5 - 22.
Drzewa binarne
i wyrażenia
arytmetyczne.



Pokazuje on jeden z możliwych przykładów zastosowania drzew binarnych, a mianowicie reprezentowanie 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 (*listek* drzewa) i operatora arytmetycznego, wiążącego w ogólnym przypadku trzy typy węzłów:

Tabela 5 - 2.

Typy węzłów w drzewie opisującym wyrażenie arytmetyczne.

lewy potomek	prawy potomek
wyrażenie	wyrażenie
wyrażenie	wartość
wartość	wyrażenie

Jeśli napiszemy odpowiednie funkcje obsługujące powyższą strukturę danych wedle przyjętych przez nas reguł postępowania, to możemy przy pomocy takiej prostej reprezentacji 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 roboty zaprzegniemy rekurencję, to algorytmy obsługi drzew binarnych (i nie tylko), stają się bardzo proste i zrozumiałe na pierwszy rzut oka.

Czy reprezentacja przy pomocy rekurencyjnych struktur danych jest optymalna? Na to pytanie można odpowiedzieć sensownie jedynie mając przed oczami ostateczne zastosowanie implementowanego drzewa: jeśli nie dbamy zbytnio 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, w §5.5 została podana dość prosta metoda na zapamiętanie w tablicy innej „drzewiastej” struktury danych – sterty. Podobnie w §5.2.2 poznaliśmy tzw. metodę tablic równoległych do reprezentacji list z wieloma kryteriami sortowania.

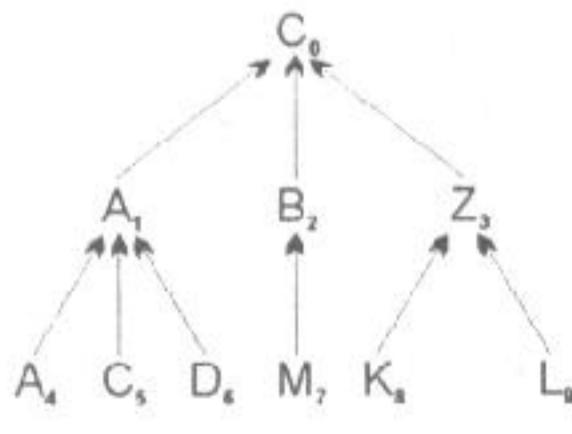
Jak widać, intelligentne użycie tablic może nam podsunąć możliwości z trudem uzyskiwane w przypadku optymalnych, listowych struktur danych.

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 określeń: *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 używała 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*¹ here!” – pokazując jednocześnie na narysowane na tablicy drzewo...

Ale wróćmy do tematu i pokażmy wreszcie obiecana implementację drzew przy pomocy tablic, tak aby uzyskać informację o węzłach „ojcach”. Rysunek 5 - 23 przedstawia drzewo służące do zapamiętywania liter (czyli pole *val* jest typu *char*).

Rys. 5 - 23.
Tablicowa reprezentacja drzewa.



	syn	ojciec
0	C	0
1	A	0
2	B	0
3	Z	0
4	A	1
5	C	1
6	D	1
7	M	2
8	K	3
9	L	3

¹ W ten sposób francuskie dzieci określają nieodłączny atrybut każdego mężczyzny.

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, spróbujmy popatrzeć na możliwe sposoby przechadzania się po gałęziach drzew...

5.6.1. 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 zaanonsowanej już na rysunku 5 - 22. Zobaczyliśmy tam, że drzewo doskonale się nadaje do reprezentacji informacyjnej 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 - 22 – do zapamiętywania zarówno operatorów (ograniczymy się tu do: +, -, * i do dzielenia wyrażonego przy pomocy : lub /), jak i operandów (liczb rzeczywistych).

wyrazen.cpp

```
struct wyrazenie
{
    double val;
    char op;
    wyrazenie *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 - 24, który ukazuje kilka pierwszych etapów tworzenia drzewa binarnego wyrażenia arytmetycznego.

Do tworzenia drzewa użyjemy dobrze nam znanego z poprzednich dyskusji stosu (patrz §5.3). Tym razem będzie on służył do zapamiętywania wskaźników do rekordów typu *struct wyrazenie*, co implikuje jego deklarację przez *STOS<wyrazenie*> s* (Jak widać warto było raz się pomóczyć i stworzyć stos w postaci klasy szablonowej).

Rys. 5 - 24.

„nadchodzące” elementy:

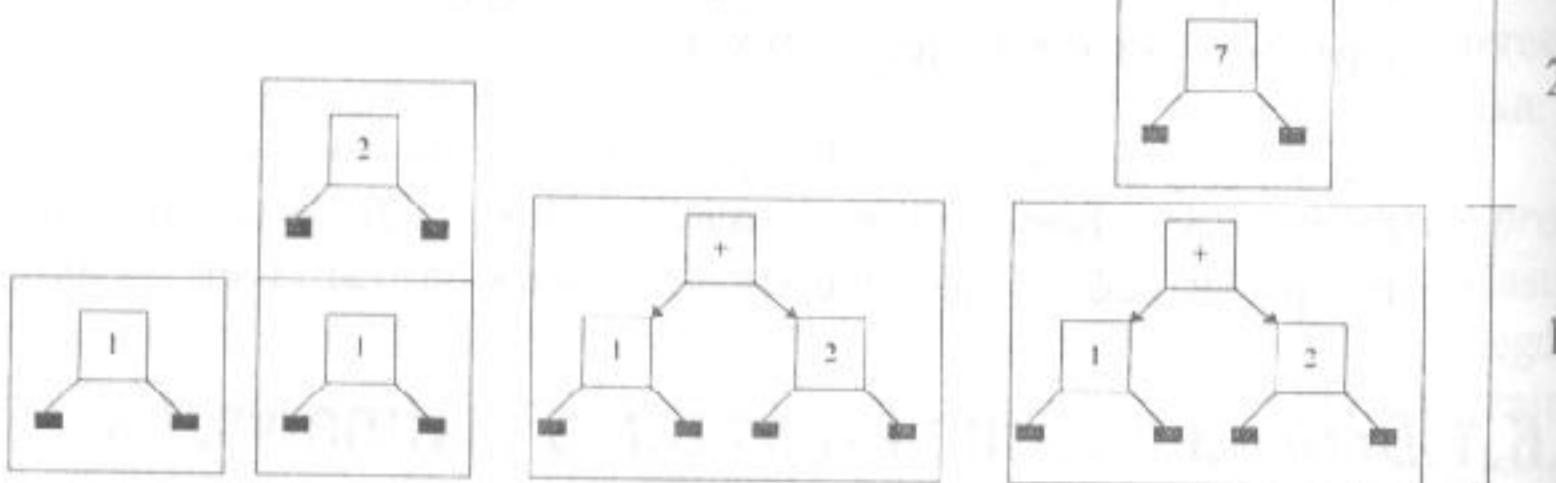
Tworzenie drzewa binarnego wyrażenia arytmetycznego.

1 2

+

7 etc.

Etapy tworzenia drzewa binarnego (zawartość stosu):



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 - 24 możemy zaobserwować opisany wyżej proces w bardziej poglądowej formie niż powyższy suchy opis. Pierwsze dwa argumenty, **1** i **2**, jako nie będące 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 „zewnętrzną”, jest operator **+**. Tworzona jest nowa komórka pamięci, jednocześnie sam fakt nadania 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 dalej...

W opisany wyżej sposób pracują kompilatory w momencie obliczania wyrażeń za pośrednictwem stosu. Jedyną różnicą jest to, że nie są odkładane na stos 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 użyciu².

Przypatrzymy się już konkretnym instrukcjom w C++, które zajmują się inicjacją drzewa binarnego.

```
typedef struct
{
    double val;
    char op;
}VAL;

void main()
{
    STOS<wyrazenie*> s;
    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';}
        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;
            x->prawy=p1;
        }
        s.push(x);
    }
    pisz_infix(x); cout << "=" << oblicz(x) << endl;
    pisz_prefix(x); cout << "=" << oblicz(x) << endl;
}
```

W powyższym listingu tablica *t* zawiera *poprawną* sekwencję danych, tzn. taką, która istotnie stworzy drzewo binarne mające sens. Warto odrobinę eksperymentować z zawartością tablicy, aby zobaczyć, jak algorytm „zareaguje” na błędny ciąg danych. Można się spodziewać, że w przypadku np. braku drugiego operanda lub operatora rezultaty otrzymane będą również błędne – jest to prawda, ale najlepiej jest przekonać się o tym „na własnej skórze”.

² Notabene wbrew pozorom ONP jest dość wszechstronnie stosowana: patrz kalkulatory firmy Hewlett Packard, język Forth, język opisu stron drukarek laserowych Postscript... W pewnych „kręgach” jest to zatem dość znana notacja.

Jak jednak obejrzeć zawartość drzewa, które tak pieczołowicie stworzyliśmy? Wbrew pozorom zadanie jest raczej trywialne i sprowadza się do wykorzystania własności „topograficznych” drzewa binarnego. Sposób interpretacji formy wyrażenia (czy jest ona *infiksowa*, *prefiksowa* czy też *postfiksowa*) 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 w kolejności:
        (wypisz(w→ left) op wypisz(w→ right))
}
```

Realizacja programowa jest oczywiście dosłownym tłumaczeniem powyższego zapisu:

```
void pisz_infix(struct wyrazenie *w)
{
    // funkcja wypisuje w postaci wrostkowej
    if(w->op=='0') // wartość liczbowa...
        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 wyrazenie *w)
{
    // funkcja wypisuje w postaci prefiksowej
    if(w->op=='0') // wartość liczbowa...
        cout << w->val << " ";
    else
    {
        cout << w->op << " ";
        pisz_prefix(w->lewy);
        pisz_prefix(w->prawy);
    }
}
```

Jak łatwo zauważyc, w zależności od sposobu przechadzania się po drzewie możemy w różny sposób przedstawić jego zawartość bez wykonywania jakiejkolwiek zmiany w strukturze samego drzewa!

Reprezentacja wyrażeń arytmetycznych byłaby z pewnością niekompletna, gdybyśmy jej nie uzupełnili funkcjami do obliczania ich wartości. 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 wyrazenie *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 '*':           // to są znane operatory
        case ':':
        case '/':
            return (poprawne(w->lewy)*poprawne(w->prawy));
        default :
            return (0); //błąd!!! -> operator nieznany
    }
}
```

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. Jego schemat jest bardzo zbliżony do tego zastosowanego w funkcji *poprawne*:

```
double oblicz(struct wyrazenie *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 ':':
case '/':
    if(oblicz(w->prawy) != 0)
        return (oblicz(w->lewy)/oblicz(w->prawy));
    else
    {
        cerr << "\nDzielenie przez zero!\n";
        exit(-1);
    }
}
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.

5.7. Uniwersalna struktura słownikowa

Nasze rozważania poświęcone strukturom drzewiastym zakończymy prezentując szczegółową implementację tzw. *Uniwersalnej Struktury Słownikowej* (określonej 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 wchodzą zbliżone kwestie.

Z uwagi na czytelność wyjaśnień wszelkie przykłady dotyczące *USS* będą tymczasowo obywały się bez poruszania zagadnienia polskich znaków dialektycznych: *q*, *ę*, *ć* etc. Temat ten poruszę dopiero pod koniec tego paragrafu, gdzie zaproponuję prosty sposób rozwiązania tego 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 przymierzają. 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 zapamiętać 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 nierozerwianie 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”¹... Sprawy komplikują się nieproporcjonalnie, 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操作owania danymi. Marzeniem byłoby posiadanie struktury danych, która przez swoją naturę automatycznie zapewnia kompresję danych już w pamięci komputera, nie ograniczając dostępu do zapamiętywanych informacji.

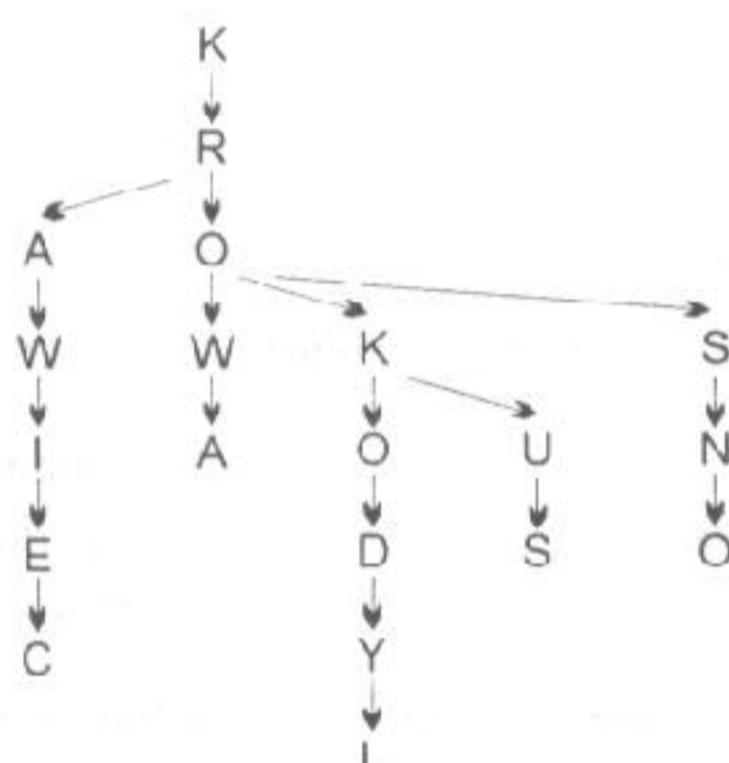
Prawdopodobnie wszyscy Czytelnicy domyślili się natychmiast, że *USS* należy do tego typu struktur danych.

Idea *USS* opiera się na następującej obserwacji: wiele słów posiada te same rdzenie (przedrostki), różniąc się jedynie końcówkami (przyrostkami). Przykładowo

¹ Ang. *Brak pamięci*.

weźmy pod uwagę następującą grupę słów: KROKUS, KROSNO, KRAWIEC, KROKODYL, KRAJ. Gdyby można było zapamiętać je w pamięci w formie drzewa przedstawionego na rysunku 5 - 25, to problem kompresji mielibyśmy z głowy. Z 31 znaków do zapamiętania zrobiło nam się raptem 21, co może nie oszałamia, ale pozwala przypuszczać, że w przypadku rozbudowanych słowników zysk byłby jeszcze większy. Zakładamy oczywiście, że w słowniku będą zapamiętywane w dużej części serię słów zaczynających się od tych samych liter – czyli przykładowo pełne odmiany rzeczowników etc.

Rys. 5 - 25.
Kompresja danych zaletą
Uniwersalnej Struktury
Słownikowej.



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 realizacją *USS*:

uss.cpp

```

const int n=29; // liczba rozpoznawanych liter

typedef struct slownik
{
    struct slownik *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). Literze ‘a’ (lub ‘A’) odpowiada komórka *t[0]*, analogicznie literom ‘z’ (lub ‘Z’) komórka *t[25]*. Dodatkowe komórki pamięci będą służyły do znaków specjalnych, które nie należą do podstawowych liter alfabetu, ale dość często wchodzą w skład słów (np. myślnik, polskie znaki diakrytyczne...).

Dla 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.

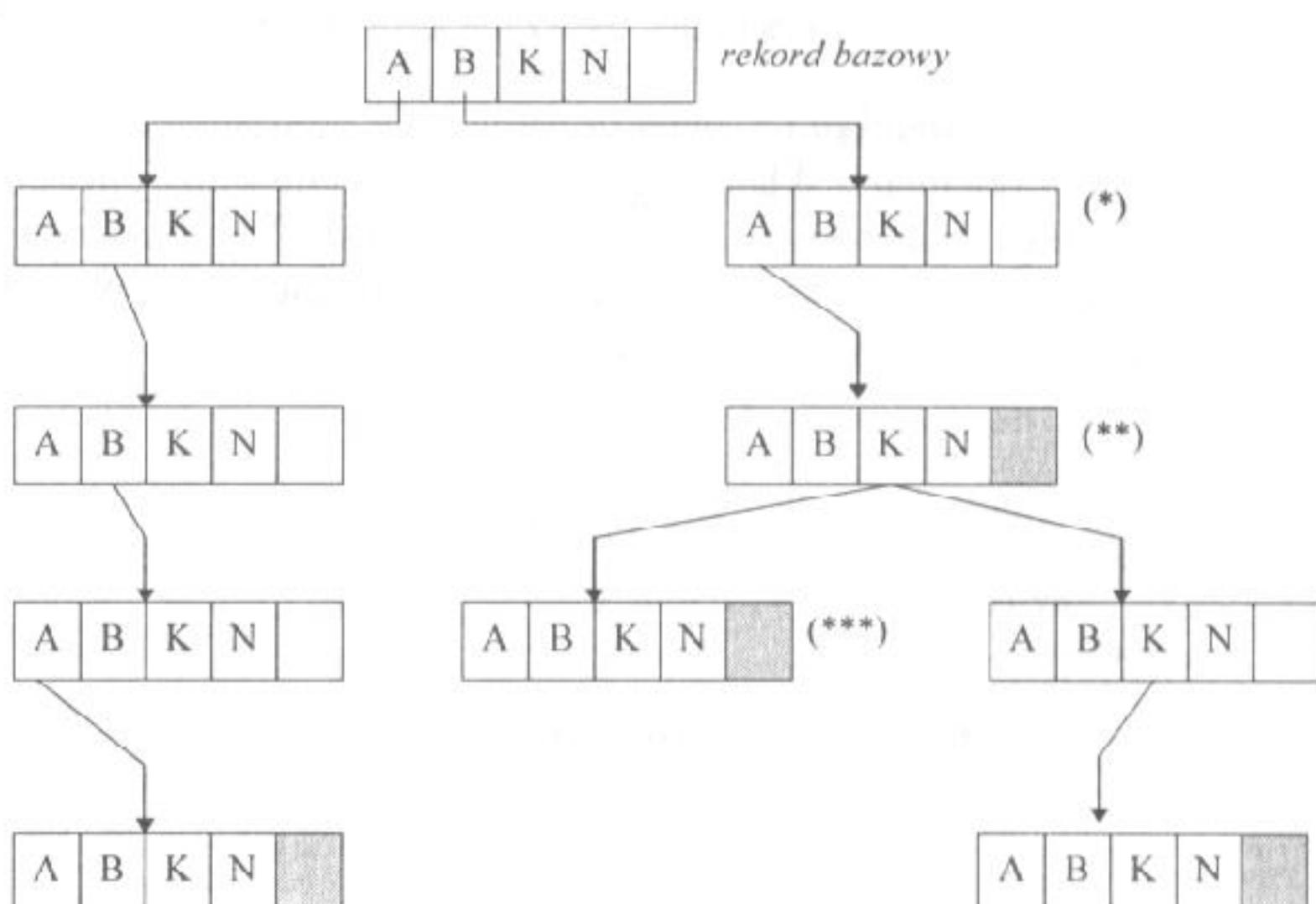


Zapętlenie wskaźnika $t[n-1]$ do swojej własnej tablicy oznacza znacznik końca słowa.

Dokładną zasadę działania *USS* wyjaśnimy na przykładzie zamieszczonym na rysunku 5 - 26.

Rys. 5 - 26.

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 , to oznacza 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: wpierw ‘B’, potem ‘A’, na koniec znacznik końca słowa – co daje razem słowo BA².

² Przyjmijmy dla potrzeb tej książki, że ono coś istotnie oznacza....

Proces przechadzania się po drzewie nie jest bynajmniej zakończony: od komórki (**) odchodzi strzałka do (***) w której także następuje „zapętlanie”. 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 tworzą one jeszcze kompletnego modułu obsługi słownika, ale ta reszta, której brakuje (obsługa dysku, „ładne” procedurki wyświetlania etc.), to już tylko zwykła „wykończeniówka”.

Omówmy po kolej procedury tworzące zasadniczy szkielet modułu obsługi *USS*.

Funkcje *do_indeksu* i *z_indeksu* pełnią rolę 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';
    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 to ostatnie służy jako znacznik końca słowa.

```
void zapisz(char *slowo, USS_PTR p)
{
    USS_PTR q; // zmienna pomocnicza
    int pos;
    for(int i=1; i<=strlen(slowo); i++)
    {
        pos=do_indeksu(slowo[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_słownik* 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_słownik(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])// koniec słowa => CR
            cout << z_indeksu(i) << endl << " ";
        else
            cout << z_indeksu(i);
        cout << "---"; // to dla ładnego wyglądu...
        pisz_słownik(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 *slowo, USS_PTR p)
{
    // szukaj słowa w słowniku
    int test=1;
    int i=0;
    while ((test==1) && i<strlen(slowo))
```

```

{
    if (p->t[do_indeksu(slowo[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*:

```

void main()
{
int i;
char tressc[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 >> tressc;
    zapisz(tressc,p);
}
pisz_slownik(p); // wypisujemy słownik
for(i=1 ;i<=4;i++) // szukamy 4 słów
{
    cout <<"Podaj słowo które mam poszukać w słowniku:";
    cin >> tressc;
    szukaj(tressc,p);
}
}

```

Przypuśćmy, że podczas sesji z programem wpisaliśmy następujące słowa: *alf*, *alfabet*, *alfabetycznie*, *anagram*, *anonim*, *ASTRonomia*, *Ankara* (duże i małe litery zostały celowo pomieszane 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

```

5.8. Zbiory

Implementacja programowa zbiorów matematycznych napotyka na szereg ograniczeń związanych z używanym językiem programowania. Miłośnicy Pascala znajdują zapewne definicje zbliżone do:

```
type Litera = 'A' .. 'Z';
ZbiorLiter = set of Litera;
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 wymog *jednakowego typu* zapamiętywanych elementów.

Niemniej, 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, testowanie przynależności do zbiorów...

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 jawnny, 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. Założymy, 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 najwyklejsza tablica typu *unsigned char*, tak jak w przykładzie poniżej:

```
set.cpp
class Zbior
{
  unsigned char zbior[256]; // cała tablica ASCII
public:
  Zbior() // konstruktor, „zeruje” zbiór
  {
    for(int i=0; i<256; i++)
      zbior[i]=0;
  }
```

```

Zbior& operator +(unsigned char c)
{
    // dodaj 'c' do zbioru
    zbior[c]=1;
    return *this; // zwraca zmodyfikowany obiekt
}

Zbior& operator -(unsigned char c)
{
    // usuwa 'c' ze zbioru
    zbior[c]=0;
    return *this; // zwraca zmodyfikowany obiekt
}

int 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
{
    cout << "{";
    for(int i=0; i<256; i++)
        if(zbior[i]==1) // element obecny
            cout << (char)i << " ";
    if(i==0)
        cout << "Zbiór pusty!";
    cout << "}\n";
}
}; // koniec definicji klasy Zbior

```

Pomimo dużej prostoty, powyższa implementacja umożliwia już manipulacje typowe dla zbiorów:

```

void main()
{
    Zbior 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 << "Zbior 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 *Zbior* o przecinanie (iloczyn) i odejmowanie zbiorów.

Możliwe jest stworzenie dowolnej w zasadzie implementacji zbiorów, tj. akceptujących zmienną liczbę danych (wymaga dynamicznego przydziału pamięci, np. przy pomocy list) jak również akceptujących złożone elementy składowe, np. struktury. Wydaje się jednak, że zaprojektowanie klasy *Zbior* z użyciem klas szablonowych (patrz §5.2.1) i list, byłoby „nadużyciem siły”, w przypadku, jeśli jedynymi niezbędnymi nam elementami zbiorów miałyby zostać jedynie... znaki alfabetu!

5.9. Zadania

Zad. 5-1

Zastanów się, jak można w prosty sposób zmodyfikować model *Uniwersalnej Struktury Słownikowej* (patrz strona 154), 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 zapamiętyanych słów.

Zad. 5-2

Zestaw dość podobnych zadań. Napisz funkcje, które usuwają:

- pierwszy element listy;
- ostatni element listy;

- c) pewien element listy, który odpowiada kryteriom poszukiwań podanym jako parametr funkcji (aby uczynić funkcję uniwersalną wykorzystaj metodę przekazania wskaźnika funkcji jako parametru).

Zad. 5-3

Napisz funkcję, która:

- zwraca liczbę elementów listy;
- wraca k -ty element listy;
- usuwa k -ty element listy.

5.10. Rozwiązania zadań

Zad. 5-1

Modyfikacja struktury USS:

```
typedef struct slownik
{
    struct slownik *t[n];
    char             *tlumaczenie;
} USS, *USS_PTR;
```

Tłumaczenie jest „dopisywane” (alokowane) w funkcji zapisz podczas zaznaczania końca słowa – w ten sposób nie stracimy związku *słowo-tłumaczenie*.

Koszt:

- bez drugiego języka:

Koszt = $(n=29)*4$ bajty („duży” model pamięci)=696000 bajtów = ok. 679 kB.

- z drugim językiem:

Założenie: średnia długość słowa angielskiego wynosi 9 bajtów + ogranicznik, czyli 10 bajtów.

Koszt = przypadek poprzedni plus $25,000 * 10$ plus pewna ilość nie użytych wskaźników na tłumaczenie – przyjmijmy zaokrąglenie na 1000. Ostatecznie mamy: $25,000*10+1000*4=254,000$ bajtów, czyli ok. 248 kB.

W danym przypadku koszt wzrósł o ok. 36 % pierwotnej zajętości pamięci.

Zad. 5-3

Oto propozycja rozwiązania zadania 5-3a:

```
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)`.

Podczas rozwiązywania zadań 5-2 i 5-3 proszę dokładnie przemyśleć efektywny sposób informacji o sytuacjach błędnych (np. próba usunięcia k -tego elementu, podczas gdy on nie istnieje etc.).

Rozdział 6

Derekursywacja

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 użytku języków nie umożliwiających 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. Tym niemniej w praktyce warto coś na temat wiedzieć, gdyż trudno derekursywacji odmówić 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 co 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, w 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, iż 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. 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 pewnym 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.

6.1. 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 przetłumaczają nasze dzieła na postać zrozumiałą przez (mikro)procesor.

Dodajmy jeszcze, że efekt tego tłumaczenia marnie przypomina to, co z takim trudem napisaliśmy i uruchomiliśmy. Wykleta ongiś instrukcja *goto* (a w każdym razie jej odpowiedniki) występuje w kodzie wynikowym częściej. Patrzmy 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:
```

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.

² Przedstawione oczywiście symbolicznie za pomocą pseudokodu asemblerowego.

```
ASM(Instr2)
koniec:
```

ASM(instr) znaczy ciąg instrukcji asemblerowych odpowiadających instrukcji *instr*, a **if**, **if_not** i **goto** są elementarnymi instrukcjami procesora (słowni 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
```

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 obsługiwać wszystkie jej egzemplarze – 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;
}
```

³ Funkcja *F* oznacza grupę przekształceń dokonywanych na parametrach funkcji.

Jak odróżnić powrót z procedury P , który powoduje definitive jej zakończenie, od tego, który przekazuje kontrolę do Instr2? Okazuje się, że jedyny łatwy do zautomatyzowania sposób polega na użyciu tzw. stosu wywołań rekurencyjnych (patrz również §5.3).

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 możliwe jest określenie momentu 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.

⁴ Warto przypomnieć, że instrukcja *goto* istnieje również w C++.

6.2. Odrobina formalizmu... nie zaszkodzi!

Mimo iż podręcznik ten bazuje na przykładach, od czasu do czasu warto przywdziać „garnitur naukowy” i zachowywać się dostoźnie – a nic tak nie przekonuje o wadze tematu jak *Definicje i Twierdzenia*. Oto i one:

Def. 1 Procedura iteracyjna *I* jest równoważna procedurze rekurencyjnej *P*, jeśli wykonuje dokładnie to samo zadanie co *P*, dając identyczne rezultaty.

Przykładowo dwie poniższe procedury *symetrial* i *symetria2* mogą być uważane za równoważne. Obie zajmują się dość bławym zadaniem rysowania „szlaczka” typu <<<<-->>>> – o regulowanej przez parametr *x* szerokości.

```
void symetrial(int x)
{
    if (x==0)
        cout << "-";
    else
    {
        cout << "<";
        symetrial(x-1);
        cout << ">";
    }
}

void symetria2(int x)
{
    for(int i=1; i<=x; i++)
        cout << "<";
    cout << "-";
    for(i=1; i<=x; i++)
        cout << ">";
```

Def. 2 Wywołanie rekurencyjne procedury *P* jest zwane *terminalnym* (ang. *end-recursion*), jeśli nie następuje po nim już 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 jakiejkolwiek 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.

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));
    }
}
```

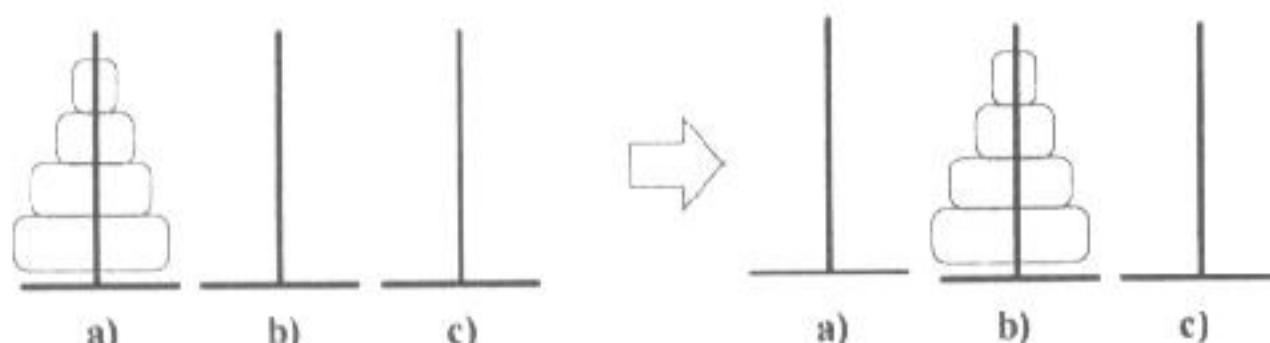
```
void P2(x)
{
    while (!Cond(x))
    {
        Instr2(x);
        x=F(x);
    }
    Instr1(x);
}
```

6.3. Kilka przykładów derekursywacji algorytmów

Wypróbujmy teraz świeżo nabycą 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.

Rys. 6 - 1.
Wieże Hanoi
– prezentacja
problemu.



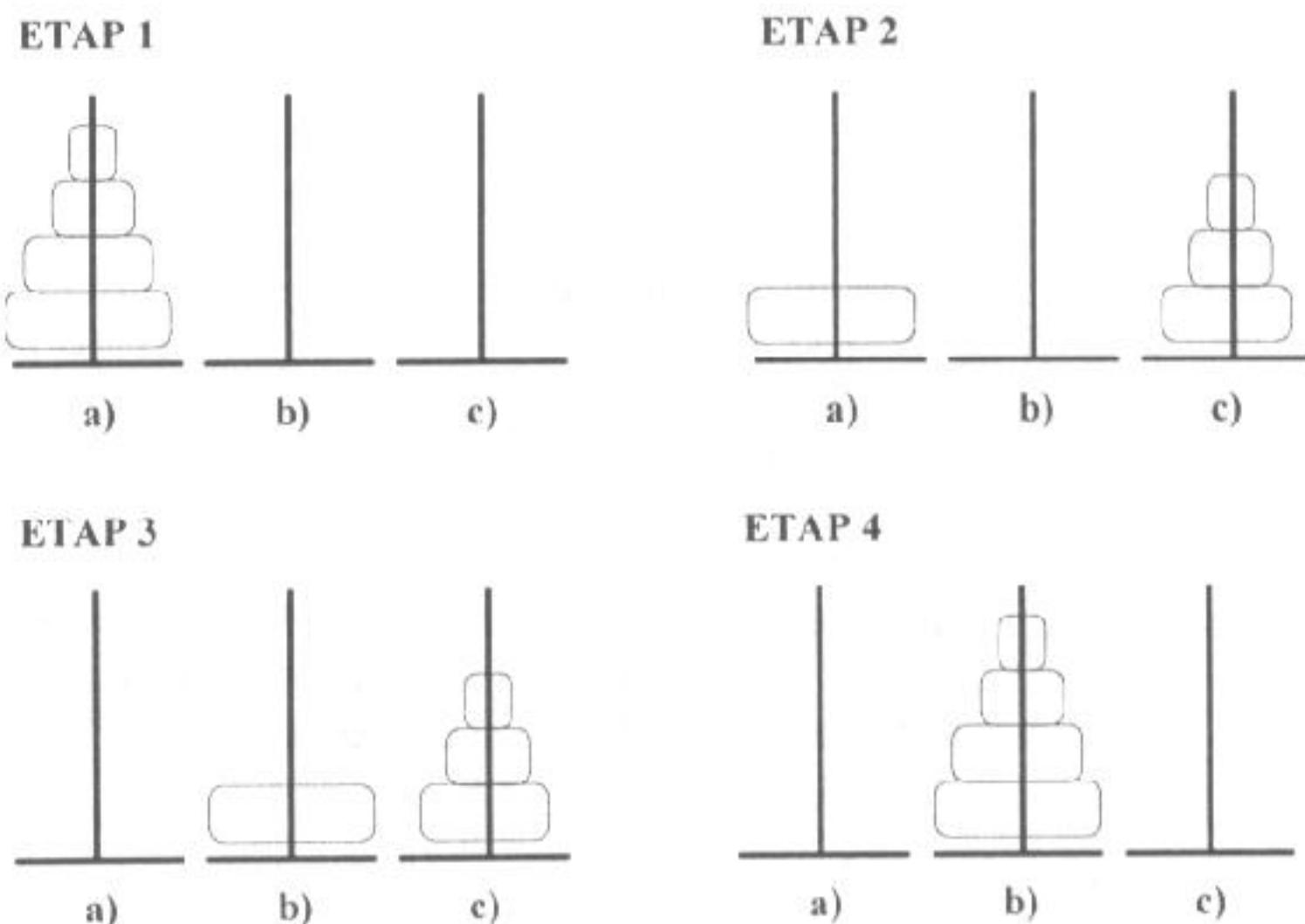
Musimy przełożyć krążki z drążka oznaczonego a na drążek b , posługując się drążkiem pomocniczym c – tak jednak postępując, aby w żadnym przypadku krążek o mniejszej średnicy nie został przykryty przez inny krążek o większej średnicy. Przyjmuje się, że krążek o numerze l ma najmniejszą średnicę, a ten o numerze n – największą. Ponadto, dla potrzeb programu wynikowego oznaczymy 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 przemieścić $n-1$ krążków z jednego drążka na drugi, zadanie sprowadza się do wykonania przemieszczeń symbolicznie przedstawionych na rysunku 6 - 2.

Rys. 6 - 2.

*Wieże Hanoi
– sposób rozwiązywania.*



Etap pierwszy przedstawia sytuację wyjściową. Założymy 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?

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 spowokowanie 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. Wynik ten nie jest trudny do uzyskania, ale dla czytelności wykładu zostanie pominięty.

O ile jednak nie możemy specjalnie w ten czas ingerować (sam problem jest dość czasochłonny „z natury”), to możemy nieco ułatwić generację kodu kompilatorowi, eliminując drugie wywołanie rekurencyjne, które spełnia warunek narzucony przez *Twierdzenie 1* (patrz str. 170). 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;
    }
    cout << "Przesuń dysk nr 1 z " << a << " na "
        << b << endl;
}
```

⁵ Wynik ten nie jest trudny do uzyskania, ale dla czytelności wykładu zostanie pominięty.

Pokaźna 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 zamiocnić 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 res)
{
    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 wspomnialiśmy 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 „dotykowego” – oprócz przepisu na rozwiązywanie łamigłówki – nie dostarcza! Poprzedzając 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;
}
```

6.4. Derekursywacja z wykorzystaniem stosu

W tym paragrafie zapoznamy się z nową metodą derekursywacji, która niestety jest dość kontrowersyjna. Zmuszeni bowiem będziemy do swoistego zaprzeczenia wielkim 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 asemblera. Zasada jest prosta: wiedząc, jak kompilator traktuje wywołania rekurencyjne, będziemy usiłowali robić to samo, lecz próbując po drodze 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 możliwa jest do wykonania przez procesor komputera. Kompilator rozpatruje programy pod kątem ich składni i nie jest raczej 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.

6.4.1. 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 BASICu strachu przed modyfikacją jakiejś ważnej „gdzie indziej” 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 będącym 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 { i } 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_wywol*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_wywol)
{
    ...
    F(param_wywol);
    ...
}
```

Pierwszy etap transformacji polega na usunięciu funkcji *F* z wywołania *P*:

```
void P(param_wywol)
{
    ...
    param_wywol=F(param_wywol);
    P(param_wywol);
    ...
}
```

Jest to najzwyczajniejsze przepisanie kodu w nieco innej postaci. Chcemy uczynić *zm_loc* i *param_wywol* 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_wywol*, 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_wywol)
    push(zm_loc)
        param_wywol=F(param_wywol);
        P(param_wywol);
    pop(zm_loc);
    pop(param_wywol);
    ...
}
```

⁶ Zarówno *zm_loc* jak i *param_wywol* reprezentują listy zmiennych – to dla skrócenia zapisu.

Dokonaliśmy 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 $param_wywol$ właściwymi wartościami⁷, tak aby zachować pełną równoważność funkcjonalną naszego programu – przed i po przeróbce. Analizując jeszcze naszą 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 str. 170). 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;
}
```

6.5. 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 na rezerwację 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.

⁷ Przed pierwszym wywołaniem procedury P .

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. Przejedźmy jednak do szczegółów.

Przypomnijmy raz jeszcze ogólną postać procedury rekurencyjnej:

```
void P1(param_wywol)
{
    ...
    param_wywol=F(param_wywol);
    P1(param_wywol);
    ...
}
```

Wiemy, że wywołanie *P1(param_wywol)* modyfikuje (lub może zmodyfikować) *zm_loc* i *param_wywol*. 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_wywol)* modyfikuje *zm_loc* i *param_wywol*, wykonałyby czynność odwrotną, tak aby przywrócić ich wartości przed wywołaniem! Inaczej mówiąc, chodzi nam o doprowadzenie programu do postaci:

```
void P2()
{
    ...
    param_wywol=F(param_wywol); (1)
    P2;
    FUNKCJA_ODWROTNIA(zm_loc,param_wywol); (2)
    ...
}
```

Działanie owej tajemniczej funkcji odwrotnej musi być takie, aby wartości *zm_loc* i *param_wywol* były dokładnie takie same w punktach programu oznaczonych (1) i (2). Jak to zrobić? Ba! oto jest dopiero pytanie! Odpowiedź na nie będzie inna w przypadku każdego programu i nie pozostaje nam nic innego, jak tylko pokazać jakiś konkretny przykład.

Poniższa procedura *P1* liczy elementy wymyślonego *ad hoc* ciągu matematycznego:

```
void P1(int a, int& b)
{
    if(a==0)
```

odwrotna.cpp

⁸ Co nie znaczy, że bezmyślnie!

```
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 by nie była 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 należałyby 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:

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.

```

void main()
{
    for (int i=0; i<17; i++) {P1(i, b); cout << b << " ";}
    cout << endl;
    for (i=0; i<17; i++)
        {a=i; P2(); cout << b << " ";}
}

```

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...

6.6. Klasyczne schematy derekursywacji

Poznane wcześniej metody eliminacji zmiennych lokalnych z procedur, jak również ich „deparametryzacja” 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. przy pomocy 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 przeplatany 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 zeru. Jeśli odpowiedź brzmi tak, procedura zwraca wartość 1 – w przypadku zaś przeciwnym 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,

¹⁰ W rozmaitych wariacjach zależnych od zestawu instrukcji procesora.

¹¹ Abstrakcji, skomplikowania...

¹² Jest to wymuszone naturalną chęcią zakończenia kiedyś szeregu wywołań rekurencyjnych!

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 $i+1$.

Jest to droga, którą pójdziemy w celu odnalezienia sposobu derekursywacji pewnych schematów, często spotykanych podczas programowania z wykorzystaniem technik rekurencyjnych.

Uwaga: Wszystkie rozpatrywane dalej schematy dotyczą procedur już bezparametrowych i pozbawionych zmiennych lokalnych.

6.6.1. 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 wywoływanego grup instrukcji oznaczonych symbolicznie jako $A(x)$, $B(x)$ i $C(x)$.

Widać od razu, iż każdorazowe spełnienie warunku instrukcji *if... else* spowoduje pewno wykonanie $A(x)$ i $N++$. Niespełnienie zaś warunku spowoduje jednokrotne wykonanie $C(x)$. Tyle możemy zaobserwować odnośnie kodu obsługiwanej 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 wywoływanego i powrotnego 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 dereksywacji ma bardzo mało wspólnego z algorytmiką, a bardzo wiele z „dziwnymi” sztuczkami łatwo prowadzącymi do przyjęcia złego stylu programowania.

6.6.2. 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, ilość 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);
}

```

Założymy, 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 przy pomocy skoku do jej początku. Podczas kolejnego wykonania procedury P możemy w bardzo łatwy sposób

¹³ Patrz §2.3.

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 oczywiście w pełni dopuszczalny, jednakże jedynie wówczas, 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 dugo 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>
--	---

¹⁴ Jeśli N wynosi 0, to wszystkie zaległe wywołania zostały już ukończone.

Sprawdźmy teraz, czy w istocie podane wyżej przekształcenie działa. W tym celu powróćmy do programu przykładowego ze strony 179 (zapisanego teraz w nieco zwięzlejszej postaci).

odwrot2.cpp

```
void P2()
{
    if(a!=0)
    {
        a--;
        P2();
        b=b+(++a);
    }
    else
        b=1;
}
```

Stosując omawiane przekształcenie otrzymujemy natychmiast:

```
void P2_ITERAT()
{
    int k=0;
    while (a!=0)
    {
        a--;
        k++;
    }
    while(k--!0)
        b=b+(++a);
}
```

Wykonanie programu potwierdza równoważność obu procedur.

6.6.3. 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
```

```
int N=1;
void P()
{
    do
    {
        while(warunek(x))
        {
            A(x);
            N*=2;
        }
        D(x);
```

```

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 170.

Mając do dyspozycji metodę funkcji przeciwnych (patrz §6.5) łatwo dojdziemy do następującej wersji zaproponowanej tam procedury:

hanoi_it.cpp

```

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 $n--$ anulują się wzajemnie, mogą być zatem po prostu usunięte.

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 wyniki.

6.7. 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. Tym 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 pewną „odmiennosć” tematu przeszukiwanie tekstów zostało zgrupowane w rozdziale kolejnym.

7.1. 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 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)
{
    for(int i=0; (i<n) && (tab[i]!=x); i++);
    return i;
}
```

¹ Wskaźniki do funkcji zostały omówione szczegółowo w §5.1

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 nie mająca 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.

W tym miejscu wypada jeszcze uściślić, że ten typ przeszukiwania, polegający na zwykłym sprawdzaniu elementu po elemencie, jest metodą bardzo wolno działającą. Winna być ona stosowana jedynie wówczas, gdy nie posiadamy żadnej informacji na temat struktury przeszukiwanych danych, ewentualnie sposobu ich składowania w pamięci. Jest oczywiste, iż dowolny algorytm przeszukiwania liniowego jest klasy $O(n)$!

7.2. 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 zbiorami danych: 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. Łatwo możemy bowiem wyeliminować z poszukiwań te obszary tablicy, w których element x na pewno nie może się znaleźć. Dokładnie omówiony przykład *poszukiwania binarnego* znalazł się już w rozdziale 2 – patrz zad. 2-2 i jego rozwiązanie. W tym miejscu możemy dla odmiany podać iteracyjną wersję algorytmu:

binary-i.cpp

```
int szukaj(int tab[], int x)
{
    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 „przepoławianie” tablicy 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 Rozkład „logarytmiczny”). 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 $\log_2 20000$ (ok. 14 porównań).

Nic tak dobrze nie przemawia do wyobraźni, jak dobrze dobrany przykład liczbowy, a powyższy na pewno do takich należy...

7.3. Transformacja kluczowa

Zanim powiemy choćby słowo na temat transformacji kluczowej, musimy sprecyzować dokładnie dziedzinę zastosowań tej metody. Otóż jest ona używana,

gdy maksymalna ilość elementów należących do pewnej dziedziny² \mathfrak{K} jest z góry znana (E_{\max}), natomiast wszystkich możliwych (różnych) elementów tej dziedziny mogłyby 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{K} byłby fizycznie niewykonalny³.

Ponieważ poprzednie zdanie brzmi makabrycznie, być może warto jest podać ilustrujący je przykład:

- chcemy zapamiętać $R_{\max} = 250$ słów o rozmiarze 10 (tablica o rozmiarze $250 * 11 = 2750$ 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 mogłyby ją wszystkie pomieścić...

Idea transformacji kluczowej polega na próbie odnalezienia takiej funkcji H , 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 pamiętały z rozdziału 5, nie był to przypadek list posortowanych, drzew binarnych, sterty...

² „Dziedziny” w sensie matematycznym.

³ A nawet jeśli tak, to zdrowy rozsądek zatrzymałby nas przed jego realizacją!

⁴ Jeden dodatkowy bajt na znak ‘\0’ kończący串tekstowy w C++.

Naturalną konsekwencją nowego sposobu zapamiętywania danych jest maksymalne uproszczenie procesu poszukiwania. Poszukując bowiem elementu x charakteryzowanego przez pewien klucz¹ v możemy się posłużyć 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!

Mamy tu zatem do czynienia z zupełniem 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 mączyć 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, to sprecyzowanie, czym jest *dobra* funkcja H , wymaga osobnego paragrafu.

7.3.1. W poszukiwaniu funkcji H

Funkcja H na podstawie klucza v dostarcza indeks 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 składować 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 tą funkcję dwóch własności:

¹ 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).

- 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).

Pierwszy punkt nie wymaga komentarza, do drugiego zaś jeszczego powróćmy, gdyż porusza on bardzo ważny problem. 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! Transformacja kluczowa jest bardzo silnie związana z aplikacją końcową i często etap uruchamiania może znacznie się wydłużać.

7.3.2. 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/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 przy pomocy 5 bitów (patrz tablica 7 - 1) – wybór kodu nie jest niczym zdeterminowanym.

Tabela 7 - 1.
Kodowanie liter przy pomocy 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=01110
O=01111	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 „rozsianie” rekordów po 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 wykazały, że istnieje grupa prostych funkcji arytmetycznych (modulo, mnożenie, dzielenie), które dość dobrze się do tego celu nadają. Omówimy je kolejno w kilku paragrafach.

suma modulo 2: $H(v_1v_2\dots v_n) = v_1 \oplus v_2 \oplus \dots \oplus v_n$

Przykład:

Dla $R_{\max} = 37$ $H(„KOT”) = (010110111010100)_2$ daje $(01011)_2 \oplus (01110)_2 \oplus (10100)_2 = (17)_{10}$.

Zalety:

- funkcja H łatwa do obliczenia; suma modulo 2, w przeciwieństwie do iloczynu i sumy logicznej, nie powiększa (jak to czyni suma logiczna) lub pomniejsza (jak iloczyn) swoich argumentów.
- Używanie operatorów $\&$ i $|$ powoduje akumulację danych odpowiednio na *początku* i na *koncu* 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 $H(„KTO”) = (10101)_2 \oplus (00101)_2 \oplus (11101)_2 = (9)_{10}$, natomiast $H(„TOK”) = (17)_{10}$.

dzielenie modulo R_{\max} : $H(v) = v \% R_{\max}$

Przykład:

Dla $R_{\max} = 37$ $H(„KOT”) = (01011 01110 10100)_2 \% (37)_{10} = (11732)_{10} = 3.$

Zalety:

- funkcja H łatwa do obliczenia.

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 nie mieszczących się w reprezentacji wewnętrznej komputera, obliczenie modulo już nie jest możliwe przez zwykłe dzielenie arytmetyczne.

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!=NULL; s++)
        tmp=(64*tmp+(*s))% Rmax;
    return tmp;
}
```

mnożenie: $H(v) = \left[((v * \theta) \% 1) * E_{\max} \right]$ 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:

$$\theta_1 = \frac{\sqrt{5} - 1}{2} = 0,6180339887 \text{ i } \theta_2 = 1 - \theta_1 = 0,3819660113.$$

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 $\theta = 0,6180339887$, $E_{\max} = 30$ i klucza $v = „KOT” = 11732$ otrzymamy² $H(„KOT”) = 23$.

7.3.3. 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, prawda należy do złożonych. Z jednej strony widzimy już, że idealne funkcje H nie istnieją³, z drugiej 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

² Programowo można otrzymać tę wartość przy pomocy instrukcji `(int)(fmod(11732 * 0.61803398887, 1) * 30);` ponadto należy na początku programu dopisać `#include<math.h>`

³ Daje się to nawet uzasadnić teoretycznie (patrz np. dobrze znany w statystyce tzw. *paradoks urodzin*).

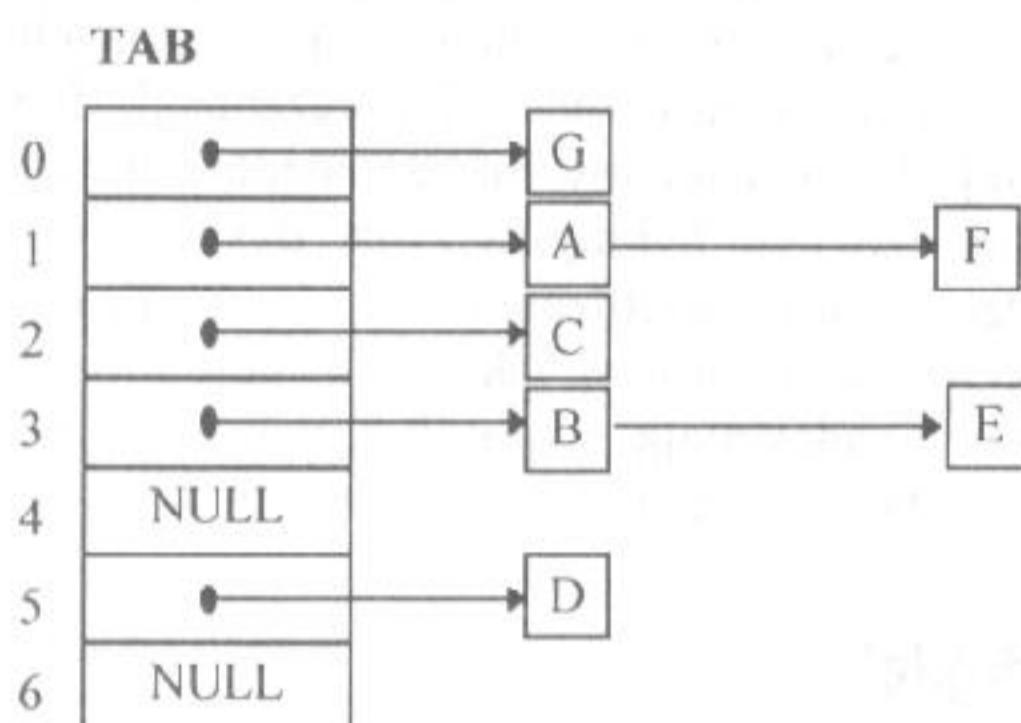
pod indeks m , stwierdzimy, że już wcześniej ktoś się tam „zameldował”, wystarczy doczepić x na koniec listy, której głowa jest zapamiętana w $T[m]$.

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 - 1.

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 (która musimy przewidującą napisać!) wykrywa tę sytuację i wstawia element F na koniec listy $T[1]$.

Rys. 7 - 1.
Użycie list
do obsługi kon-
fliktów dostępu.



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 (która musimy przewidującą napisać!) wykrywa tę sytuację i wstawia element F na koniec listy $T[1]$.

Podobna sytuacja dotyczy rekordów B i E . Proces poszukiwania elementów jest zbliżony do ich wstawiania – Czytelnik 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 sztuczne⁴ – równie dobrze można było trzymać się list i innych dynamicznych 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!

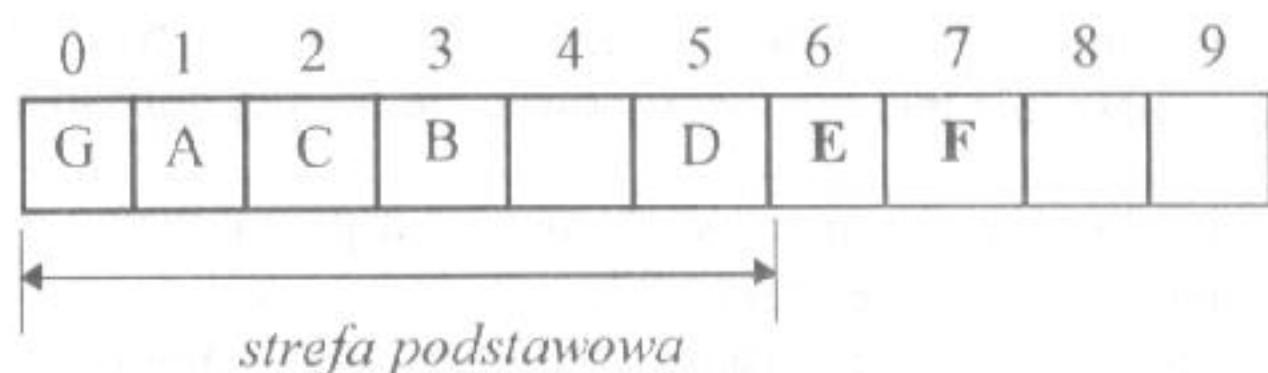
Metoda transformacji kluczowej została z założenia przypisana aplikacjom, które pozwalając przewidzieć maksymalną ilość rekordów do zapamiętania, umożliwiają zarezerwowanie pamięci na statyczną tablicę stwarzającą łatwy, indeksowany dostęp do nich. Jeśli możemy zarezerwować tablicę na wszystkie elementy, 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 *przepelnienia*. Do tej drugiej elementy trafiałyby w momencie stwierdzenia braku miejsca w części podstawowej. Strefa przepelnienia wypełniana byłaby liniowo wraz z napływem nowych elementów „kolizyjnych”. W celu ilustracji nowego pomysłu spróbujmy wykorzystać dane z rysunku 7 - 1, zakładając rozmiar stref: *podstawowej* i *przepelnienia* na odpowiednio: 6 i 4.

Efekt wypełnienia tablicy jest przedstawiony na rysunku 7 - 2.

Rys. 7 - 2.

Podział tablicy do obsługi konfliktów dostępu.



⁴ Choć parametry „czasowe” tej metody są bardzo korzystne.

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 silnie 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 zapełnienia 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 opisanej poprzednio metodzie w sposób nieco sztuczny rozwiązaliśmy problem konfliktów dostępu w tablicy T . Podzieliliśmy 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 należałyby 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 usiłowały czynić poprzednie metody – spróbować się do niej dopasować?

Idea jest następująca: w momencie zapisu nowego rekordu⁵ do tablicy, w przypadku stwierdzenia konfliktu możemy spróbować zapisać element na pierwsze kolejne wolne miejsce. Algorytm funkcji *wstaw* byłby wówczas następujący (zakładamy próbę zapisu do tablicy T rekordu x charakteryzowanego kluczem v):

⁵ Oczywiście może to być również dowolna zmienna prosta!

```

int pos=H(x.v);
while (T[pos] != WOLNE)
    pos = (pos+1) % Rmax;
T[pos]=x;

```

Założymy teraz, że poszukujemy elementu charakteryzującego się kluczem k . W takim przypadku funkcja *szukaj* mogłaby wyglądać następująco:

```

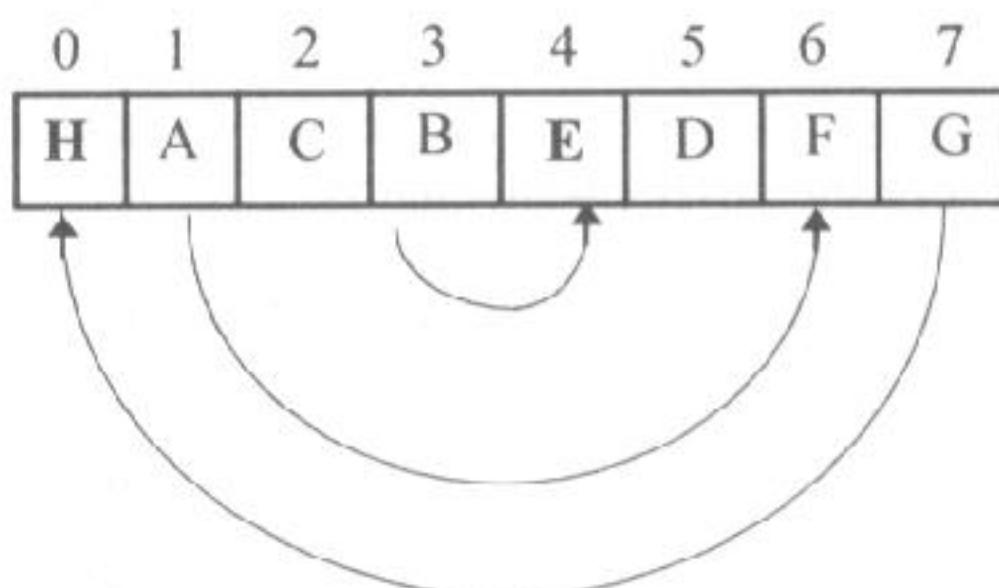
int pos=H(k);
while ((T[pos] != WOLNE) && (T[pos].v != k))
    pos = (pos+1) % Rmax;
return T[pos]; // zwraca znaleziony element

```

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 wstawienia do tablicy T rekordów A, B, C, D, E, F, G i H którym funkcja H przypdzieliła 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 - 3:

Rys. 7 - 3.
Obsługa konfliktów dostępu przez próbowanie liniowe.



Dość ciekawymi jawią się teoretyczne wyliczenia średniej ilości prób potrzebnej do odnalezienia danej x . W przypadku poszukiwania zakończonego sukcesem średnia liczba prób wynosi około:

$$\frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$$

gdzie α jest współczynnikiem zapełnienia tablicy T . Analogiczny wynik dla poszukiwania zakończonego niepowodzeniem wynosi około:

$$\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$$

Przykładowo dla tablicy zapełnionej w dwóch trzecich swojej pojemności ($\alpha = \frac{2}{3}$) liczby te wyniosą odpowiednio: 2 i 5.

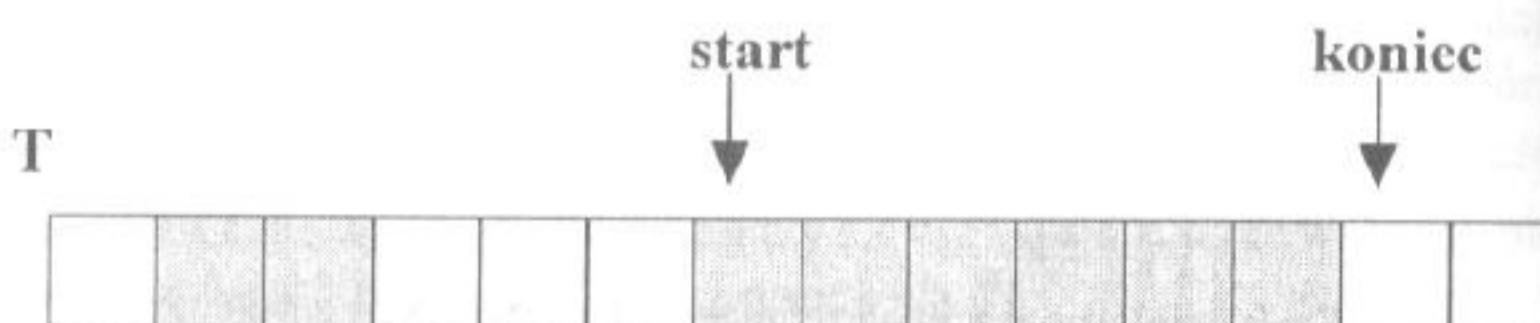
W praktyce należy unikać szczelnego zapełniania tablicy T , gdyż zacytowane powyżej liczby stają się bardzo duże (α nie powinno przybierać wartości bliskich 1). Powyższe wzory zostały wyprowadzone przy założeniu funkcji H , która rozsiewa równomiernie elementy po dużej tablicy T . Te zastrzeżenia są tu bardzo istotne, bowiem podane wyżej rezultaty mają charakter statystyczny.

Podwójne kluczowanie

Stosowanie próbkowania liniowego prowadzi do niekorzystnego liniowego zapełniania tablicy T , co kłoci się z wymogiem narzuconym wcześniej funkcji H (patrz §7.3.1). Intuicyjnie rozwiązanie tego problemu nie wydaje się trudne: trzeba uczynić coś, aby nieco bardziej losowo „porozrzucić” elementy. Próbkowanie liniowe nie było z tego względu dobrym pomysłem, gdyż napotkałszy pewien zapełniony obszar tablicy T , proponowało wstawienie nowego elementu tuż za nim – jeszcze go powiększając! Czytelnik mógłby zadać pytanie: a dla czego jest to aż takie groźne? Oczywiście względy estetyczne nie grają tu żadnej roli: zauważmy, że liniowo zapełniony obszar przeszkadza w szybkim znalezieniu wolnego miejsca na wstawienie nowego elementu! Fenomen ten utrudnia również sprawne poszukiwanie danych.

Rozpatrzmy prosty przykład przedstawiony na rysunku 7 - 4.

Rys. 7 - 4.
Utrudnione
poszukiwanie
danych przy
próbkowaniu li-
niowym.



Na rysunku tym zacieniowane komórki tablicy oznaczają miejsca już zajęte. Funkcja $H(k)$ dostarczyła pewien indeks, od którego zaczyna się przeszukiwana strefa tablicy (poszukujemy oczywiście 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 kosztowne⁶ porównanie $T[pos].v \neq k$ (patrz algorytm procedury *szukaj* ze strony 201).

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 ctykietą 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*. Podczas napotkania kolizji następuje próba „rozrzucenia” elementów przy pomocy drugiej, pomocniczej funkcji H .

Procedura *wstaw* pozostaje niemal niezmieniona:

```
int pos = H1(x.v);
int krok = H2(x.v);
while (T[pos] != WOLNE)
    pos = (pos+krok) % Rmax;
T[pos] = x;
```

Procedura poszukiwania jest bardzo podobna i Czytelnik z pewnością będzie w stanie ją napisać samodzielnie, wzorując się na przykładzie poprzednim.

Przedyskutujmy teraz problem doboru funkcji $H2$. Nie trudno się domyślić, iż ma ona duży wpływ na jakość procesu wstawiania (i oczywiście poszukiwania!). Przede wszystkim funkcja $H2$ powinna być różna od $H1$! W przeciwnym wypadku doprowadilibyśmy tylko do bardziej skomplikowanego tworzenia stref „ciągły” – a właśnie od tego chcemy uciec... Kolejny wymóg jest oczywisty: musi być to funkcja prosta, która nie spowolni nam procesu poszukiwania/wstawiania. Przykładem takiej prostej i jednocześnie skutecznej w praktyce funkcji może być $H2(k)=8-(k\%8)$: 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 ilości prób przy poszukiwaniu zakończonym sukcesem i porażką. W przypadku poszukiwania zakończonego sukcesem średnia liczba prób wynosi około:

⁶ Koszt operacji porównania zależy od stopnia złożoności klucza, tzn. od ilości i typów pól rekordu, które go tworzą.

$$\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$$

(gdzie α jest, tak jak poprzednio, współczynnikiem zapełnienia tablicy T).

Analogiczny wynik dla poszukiwania zakończonego niepowodzeniem wynosi około:

$$\frac{1}{1-\alpha}.$$

7.3.4. Zastosowania transformacji kluczowej

Dotychczas obracaliśmy się wyłącznie w kręgu elementarnych przykładów: tablice o małych rozmiarach, proste klucze znakowe lub liczbowe... 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/linker, możliwe jest wykorzystanie metod transformacji kluczowej do odszukiwania skompilowanych modułów w dużych plikach bibliotecznych.

7.3.5. 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. Tym niemniej ogólne wnioski o charakterze praktycznym są warte zacytowania:

- przy słabym wypełnieniu⁷ tablicy T wszystkie metody są w przybliżeniu tak samo efektywne;
- metoda próbkowania 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.

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

⁷ Tzn. do ok. 30-40 % całkowitej objętości tablicy.

liczba jest z dużym prawdopodobieństwem przewidywalna. Nic 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, za którą sens stosowania transformacji kluczowej staje się coraz mniej widoczny – dlatego po prostu nie warto zbytnio się do niej zbliżać. Niemniej metoda jest ciekawa i warta stosowania – oczywiście uwzględniajszy kontekst praktyczny aplikacji końcowej.

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 pozostańmy przy ogólnikowym stwierdzeniu „tekst” wiedząc, że za określeniem tym może się kryć dość sporo znaczeń.

8.1. 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 rozwiązuający to zadanie bazując na „pomysłach” symbolicznie przedstawionych na rysunku 8 - 1.

Zarezerwujmy indeksy j i i do poruszania się odpowiednio we wzorcu i tekście podczas operacji porównywania znak po znaku zgodności wzorca z tekstem. Założmy, 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 ($i++; j++$).

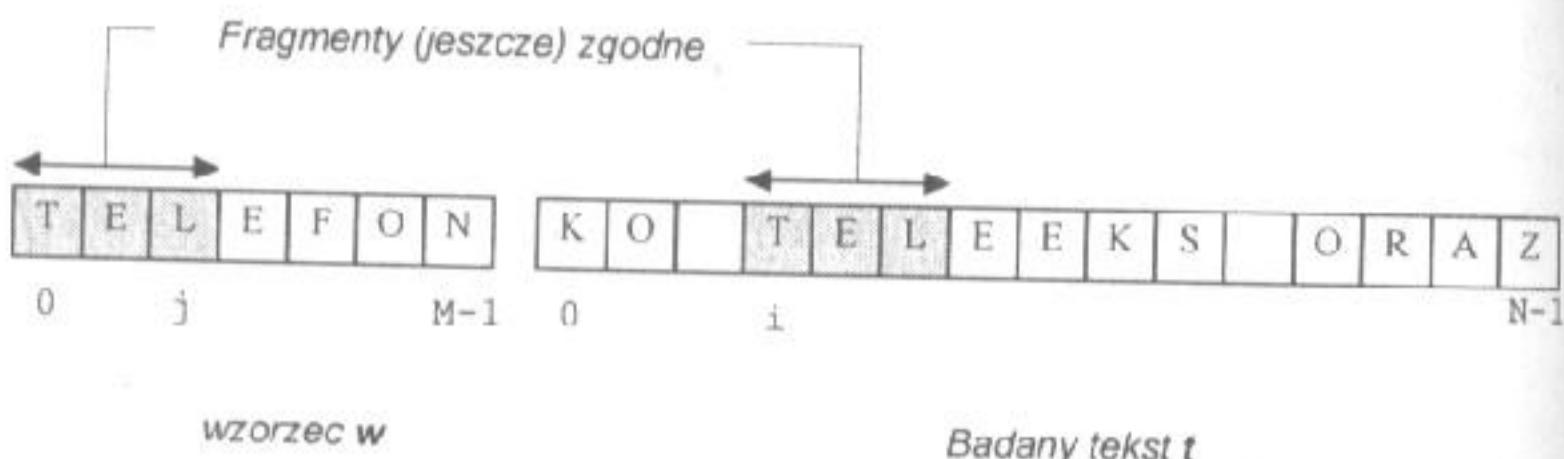
¹ Reprezentujący np. pamięć ekranu.

² Np. ASCII lub dowolny inny.

³ Ang. *pattern matching*.

Rys. 8 - 1.

Algorytm typu
brute-force prze-
szukiwania tekstu.



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 $j-1$ znaków, wyzerowując przy okazji j . Omówmy jeszcze moment stwierdzenia całkowitej zgodności wzorca z tekstem. Kiedy to nastąpi? Otóż nie jest trudno zauważyc, że podczas stwierdzenia zgodności ostatniego znaku j powinno zrównać się z M . Możemy wówczas łatwo odtworzyć pozycję, od której wzorzec startuje w badanym tekście: będzie to oczywiście $i-M$.

Tłumacząc powyższe na C++ możemy łatwo dojść do następującej procedury:

txt-1.cpp

```

int szukaj(char *w, char *t)
{
    int i=0, j=0, M, N;
    M=strlen(w);      // długość wzorca
    N=strlen(t);      // długość tekstu
    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*:

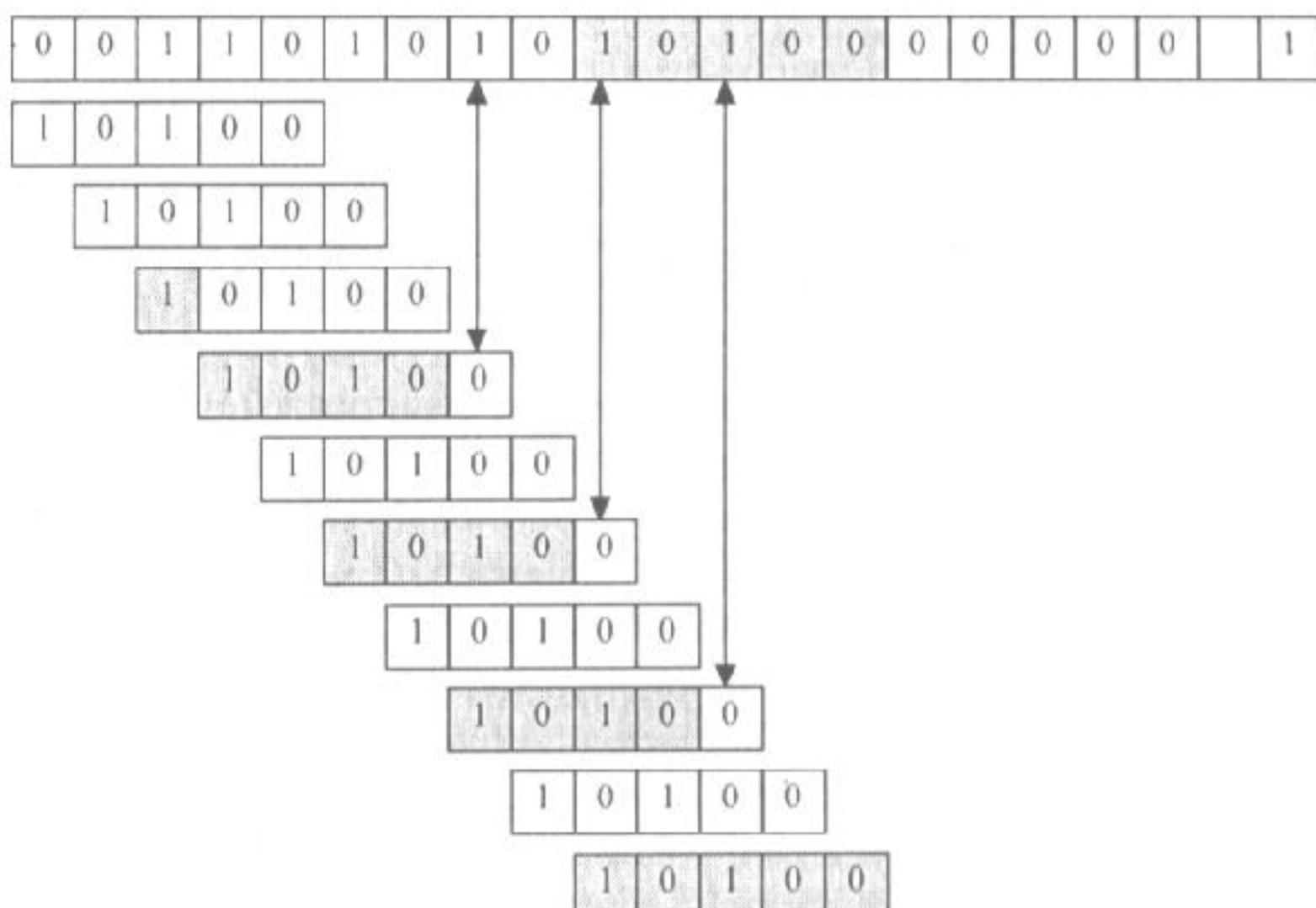
```

void main()
{
    char *b="abrakadabra", *a="rak";
    cout << szukaj(a,b) << endl;      // zwraca 2
}

```

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorzec, lub `-1` w przypadku, gdy poszukiwany tekst nie został odnaleziony – jest to znana nam już doskonale konwencja. Przypatrzymy się dokładniej przykładowi poszukiwania wzorca `10100` w pewnym tekście binarnym (patrz rysunek 8 - 2).

*Rys. 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 wzorzec, dla których proces poszukiwania będzie trwał możliwie najdłużej. Są to oczywiście zarówno tekst, jak i wzorzec złożone z samych „zer” i zakończone „jedynką”⁴.

Spróbujmy obliczyć klasę tego algorytmu dla opisanego przed chwilą ekstremalnego najgorszego przypadku. Obliczenie nie należy do skomplikowanych czynności: zakładając, że „restart” algorytmu będzie konieczny $(N-1)-(M-2)=N-M+1$ razy, i wiedząc, że podczas każdego cyklu jest konieczne wykonanie M porównań, otrzymujemy natychmiast $M(N-M+1)$, czyli około⁵ $M \cdot N$.

⁴ Zera i jedynki symbolizują tu dwa, różnie od siebie, znaki.

⁵ Typowo M będzie znacznie mniejsze niż N .

Zaprezentowany w tym paragrafie algorytm wykorzystuje komputer jako bezmyślnego, ale sprawne liczydło⁶. Jego złożoność obliczeniowa eliminuje go w praktyce z przeszukiwania tekstu binarnych, w których może wystąpić wiele niekorzystnych konfiguracji danych. Jedyną zaletą algorytmu jest jego prosta, co i tak nie czyni go na tyle atrakcyjnym, by dać się zameczyć jego powolnym działaniem.

8.2. 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 był łatwo implementowalny praktycznie – ukazując przy okazji, iż pomiędzy praktycznymi realizacjami a rozwiązaniami teoretycznymi wcale nie istnieje 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 rozpropagowanie.

W roku 1980 R. M. Karp i M. O. Rabin doszli do wniosku, że przeszukiwanie tekstu nie jest aż tak dalekie od standardowych metod przeszukiwania i wy znaleźli algorytm, który działając ciągle w czasie proporcjonalnym do $M+N$, jest ideowo zbliżony do poznanego już przez nas algorytmu typu *brute-force*. Na

⁶ Termin *brute-force* jeden z moich znajomych ślicznie przetłumaczył jako „metodę mastodonta”.

dodatek jest to algorytm łatwo dający się generalizować na poszukiwanie 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.

8.2.1. 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 ogóle¹, to w razie restartu nie musimy cofać wskaźnika i o $j-1$ pozycji, jak to było poprzednio (patrz str. 208). W tym przypadku możemy po prostu zwiększać 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 wstępny zbadaniu wzorca, w celu obliczenia ilości 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 wzorca² 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.

Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorzec bardzo wolno w prawo, patrząc jednocześnie na już zbadany tekst – tak aby obserwować

¹ Przykład: „ABBBBBBB” – znak ‘A’ wystąpił tylko jeden raz.

² Lub i w przypadku badanego tekstu.

Rys. 8 - 3.

Wyszukiwanie optymalnego przesunięcia w algorytmie K-M-P.



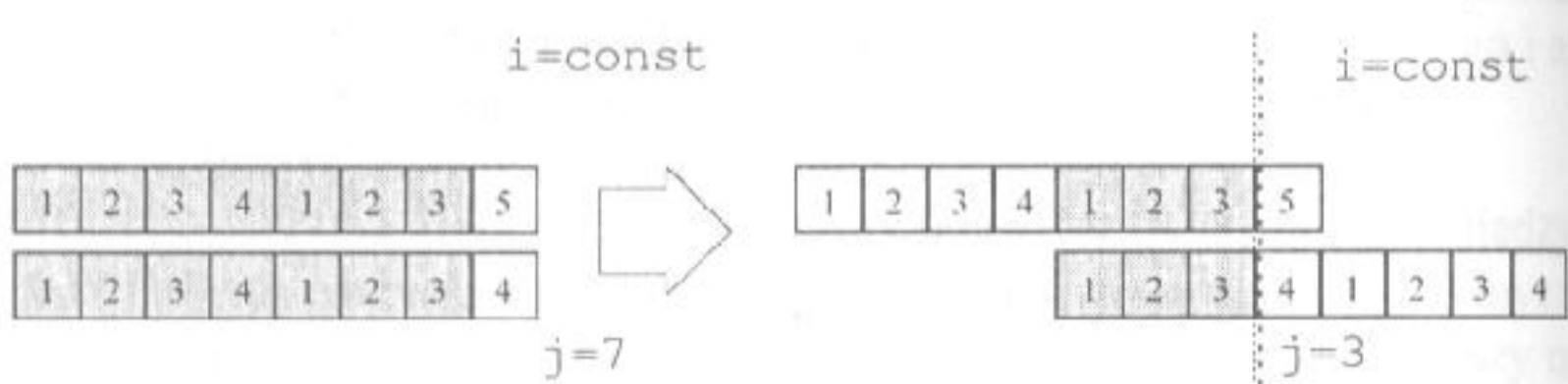
ewentualne pokrycie się tej części wzorca, która znajduje się po lewej stronie przerywanej kreski, z tekstem, który umieszczony jest powyżej wzorca. W pewnym momencie może okazać się, że następuje pokrycie obu tych części. Zatrzymujemy wówczas przesuwanie i kontynuujemy testowanie (znak po znaku) zgodności 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 -I znaków było zgodnych” – w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie sam wzorzec, 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, poatrzymy na efekt ich działania na kilku kolejnych przykładach.

Rys. 8 - 4.

„Przesuwanie się” wzorca w algorytmie K-M-P (1).

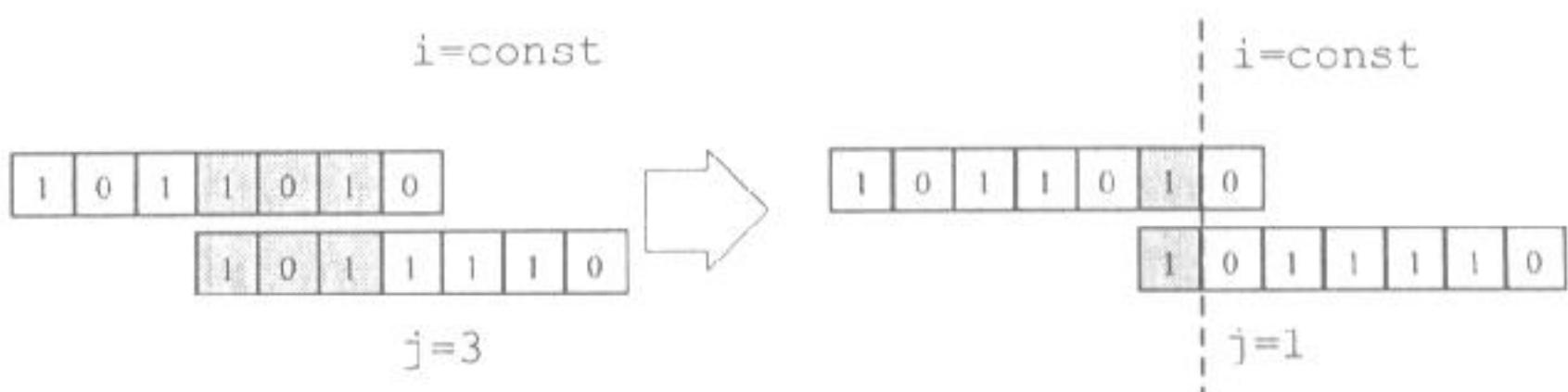


Na rysunku 8 - 4 możemy dostrzec, iż na siódmej pozycji wzorca³ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność. Jeśli zostawimy indeks i „w spokoju”, to modyfikując wyłącznie j możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? „Ślizgając” go wolno w prawo (patrz rysunek 8 - 3) 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!

³ Licząc indeksy tablicy tradycyjnie od zera.

Analogiczny przykład znajduje się na rysunku 8 - 5.

Rys. 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 przesunięć *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 niemożliwe jest tu przesuwanie wzorca w celu uzyskania nałożenia się znaków. Z tego powodu chcemy, aby indeks j pozostał niezmieniony 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 i 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

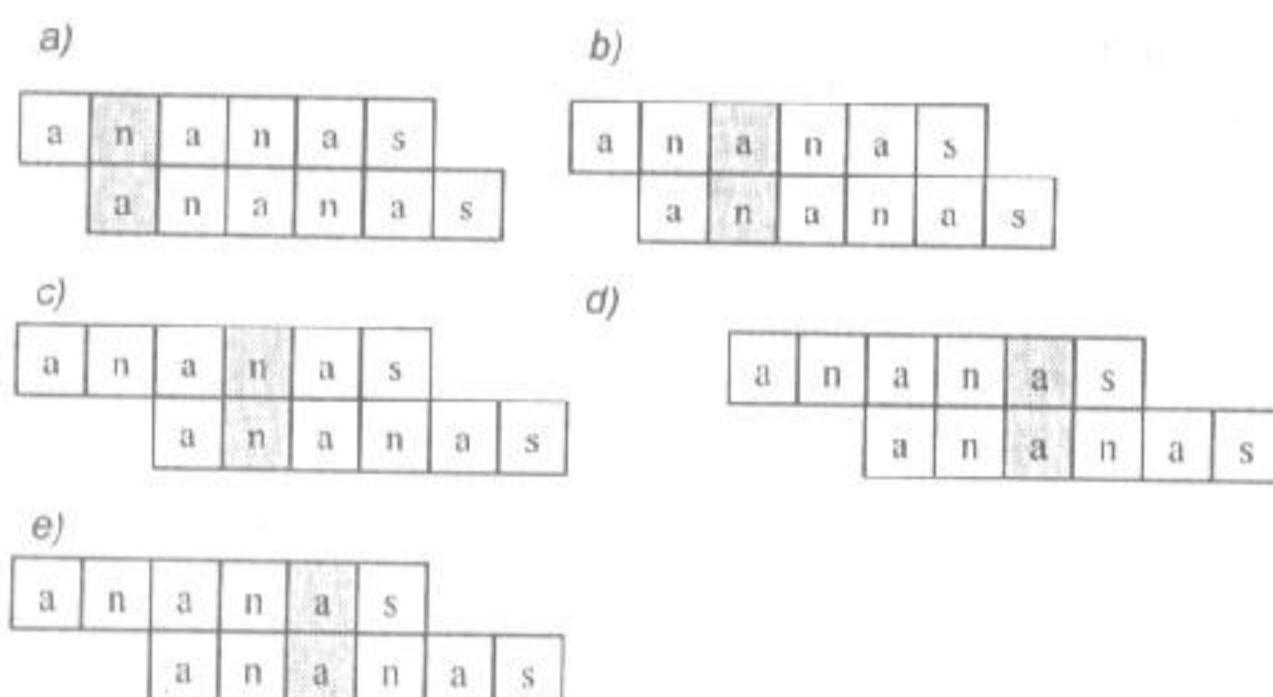
inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna – jest ona praktycznie identyczna z *kmp* z tą tylko różnicą, iż algorytm sprawdza zgodność wzorca.. z nim samym!

```
int shift[M];
int 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 .

Rys. 8 - 6.

Optymalne
przesunięcia wzor-
ca „ananas”.



Popatrzmy, jaki będzie efekt zadziałania funkcji *init_shifts* na słowie „ananas” (patrz rysunek 8 - 6). Zaciemnione 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 zaciemnianą (porównaj rysunek 8 - 5). 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 tekstu słowa „ananas”, to w funkcji *kmp* można „wbudować” tablicę przesunięć:

```
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 zwięzłym kodem wynikowym asemblerowym, jest zatem bardzo szybka. Posiadacze kompilatorów, które umożliwiają generację kodu wynikowego jako tzw. „assembly output”⁴ 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 dyskietce).

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 zauważyc, 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).

⁴ W przypadku kompilatorów popularnej serii Turbo C++/Borland C++ należy skompilować program „ręcznie” poprzez polecenie *tcc -S -Ixxx plik.cpp*, gdzie *xxx* oznacza katalog z plikami typu *H*; identyczna opcja istnieje w kompilatorze GNU *c++*, należy „wystukać”: *c++ -S plik.cpp*.

8.2.2. Algorytm Boyera i Moore'a

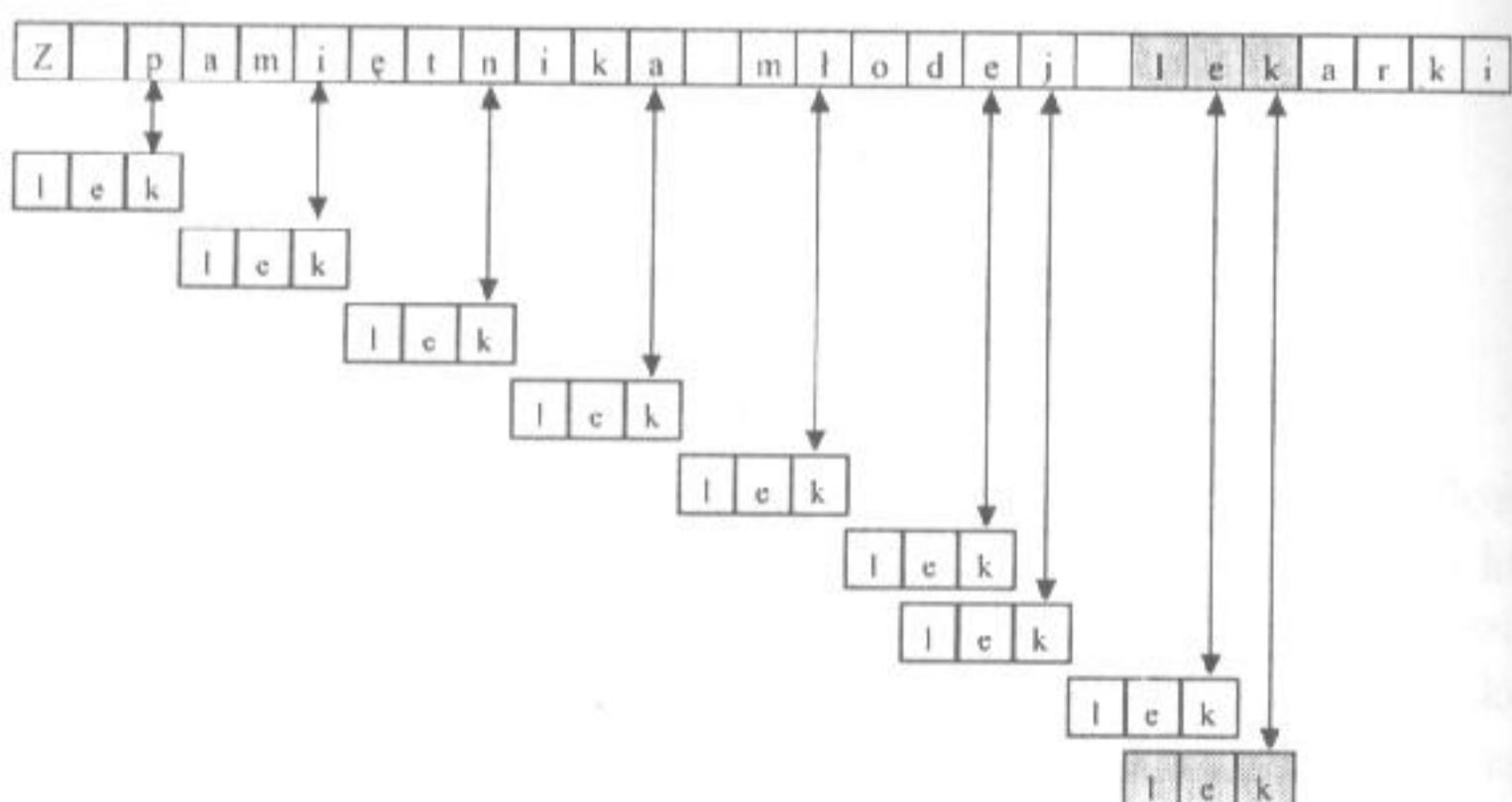
Kolejny algorytm, który będziemy omawiali, jest ideowo znacznie prostszy do zrozumienia niż algorytm *K-M-P*. W przeciwieństwie do metody *K-M-P* porównywaniu ulega *ostatni* znak wzorca. To niekonwencjonalne podejście niesie ze sobą kilka istotnych zalet:

- 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 l – 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”⁵.

Pierwsze pięć porównań trafia na litery: p , i , n , a i l , 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 przesuwa wzorzec o tyle pozycji do przodu, aby litery e nałożyły się na siebie, i porównywanie jest kontynuowane.

Rys. 8 - 7.
Przeszukiwanie
tekstu metodą
Boycera i Moore'a.



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 .

⁵ Tytuł znakomitego cyklu autorstwa Ewy Szumańskiej.

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 alfabetie – wszystkie znaki, które mogą wystąpić w tekście plus spacja. Będziemy również potrzebowali prostej funkcji *indeks*, która zwraca w przypadku spacji liczbę zero – w pozostałych przypadkach numer litery w alfabetie. 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. Jedna z możliwych wersji funkcji *indeks* jest przedstawiona poniżej:

```
const K=26*2+2*2+1; //znaki ASCII+polskie litery+spacja
int shift[K];
int indeks(char c)
{
    switch(c)
    {
        case ' ':return 0; // spacja=0
        case 'e':return 53;
        case 'E':return 54; // polskie litery
        case 'ł':return 55;
        case 'Ł':return 56; // itd. dla pozostałych
        default: // polskich liter
            if(islower(c)) // 'c' jest mała litera?
                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 zbytnio zdziwiony sposobem inicjalizacji:

```
int init_shifts(char *w)
{
    int i, M=strlen(w);
    for(i=0;i<K;i++)
        shift[i]=M;
    for(i=0;i<M;i++)
        shift[indeks(w[i])]=M-i-1;
}
```

Przejďmy wreszcie do prezentacji samego listingu algorytmu:

```
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;
}

```

Algorytm Boyera 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 przesunięć⁶ autorzy algorytmu proponują skombinowanie powyższego algorytmu z tym zaproponowanym przez Knutha, Moitisa 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.

8.2.3. 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:

- wzorzec 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 znaków⁷. Z tych M znaków wyliczamy na bieżąco $H_t = H(t)$.

Zakładając 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 każdego

⁶ Rozważ np. wielokrotne występowanie takich samych liter we wzorcu.

⁷ Na samym początku będzie to oczywiście M pierwszych znaków tekstu.

słowa wejściowego o długości M wydaje się tak samo kosztowny – jak nie bardziej! – jak zwykłe sprawdzanie tekstu znak po znaku (patrz algorytm *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 znaczaco potrafi uprościć obliczenia wykonywane w kroku bieżącym.

Załóżmy, że ciąg M znaków będziemy interpretować jako pewną liczbę całkowitą. Przyjmując za b – jako podstawę systemu – ilość 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 zobaczymy 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 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 dla tego „nowego” słowa wartości funkcji $H(x')$. Czy istotnie zachodzi potrzeba powtarzania całego wyliczenia? Być może istnieje pewne ułatwienie bazujące na zależności jaką 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 z uwagi na przykład wielkość liczby, z którą mamy do czynienia – a poza tym, gdzie tu byłby zysk szybkości?! Identyczny wynik otrzymuje się jednak aplikując funkcję *modulo* po każdej operacji częst-

kowej i przenosząc otrzymaną wartość do następnego wyrażenia cząstkowego. Dla przykładu weźmy obliczenie:

$$(5 * 100 + 6 * 10 + 8) \% 7 = 568 \% 7 = 1.$$

Wynik ten jest oczywiście prawdziwy, co można łatwo sprawdzić z kalkulatorem. Identyczny rezultat da nam jednak następująca sekwencja obliczeń:

$$\begin{array}{ccc} 5 * 100 \% 7 = 3 & \downarrow & (3 + 6 * 10 \% 7 = 0) \\ & & \downarrow \\ & & (0 + 8 \% 7 = 1) \end{array}$$

... co też jest łatwe do weryfikacji.

Implementacja algorytmu jest prosta, lecz zawiera kilka instrukcji wartych omówienia. Popatrzmy na listing:

rk.cpp

```

int rk(char w[], char t[])
{
    unsigned long i, bM_1=1, Hw=0, Ht=0, M, N;
    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;
}

```

W pierwszym etapie następuje wyliczenie początkowych wartości H_t i H_w . Ponieważ ciągi znaków trzeba interpretować jako liczby, konieczne będzie zastosowanie znanej już nam doskonale funkcji *indeks* (patrz str. 217). 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 $b*p$ 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 pierwszą⁸, 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 $t[i] \dots t[i+M-1]$ 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 znacznie szybciej przez komputer niż zwykłe mnożenie. Przykładowo dla $b=64$ możemy zapisać mnożenie $b*p$ jako $p<<6$.

Gwoli formalności jeszcze można dodać, że gdy nie występuje kolizja (typowy przypadek!), algorytm Robina i Karpa wykonuje się w czasie proporcjonalnym do $M+N$.

⁸ W naszym przypadku jest to liczba 33554393.

Rozdział 9

Zaawansowane techniki programowania

Rozdziały poprzednie (szczególnie 2 i 5) 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 nicco przewrotna i łatwo może nieuważnego 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 ilustrowane bardzo różnorodnymi zadaniami i generalnie rzecz biorąc będą dostarczać urzekająco efektownych rezultatów. Co więcej, będzie się wręcz wydawać, że dostajemy do ręki uniwersalne recepty, które *automatycznie* spowodują zniknięcie wszelkich nierozerwiązywalnych 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 spałą 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...*).

9.1. Programowanie typu „dziel-i-rządź”

Programowanie typu „dziel-i-rządź”¹ polega na wykorzystaniu podstawowej cechy rekurencji: dekompozycji problemu na pewną skończoną ilość 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 zbieg 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 pod-problemów.

Uwaga: funkcja $KOMB(N)$ nie jest rekurencyjna.

Zasadniczo znak $+$ nie jest użyty powyżej w charakterze arytmetycznym, ale jeśli będziemy rozumować przy pomocy 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 dla estetycznych celów (choć nie jest to oczywiście zabronione), ale ma za zadanie zwiększenie efektywności programu. Inaczej rzecz ujmując: chodzi nam o *przyspieszenie* algorytmu.

Technika „dziel-i-rządź” pozwala w wielu przypadkach na zmianę klasy algorytmu (np. z $O(n)$ do $O(\log 2N)$ etc.). Z drugiej jednak strony istnieje grupa zadań, dla których zastosowanie metody „dziel-i-rządź” nie spowoduje pożądanego przyspieszenia – z rozdziału 3 wiemy, jak porównywać ze sobą algorytmy, i przed-

¹ Termin ten, rozpropagowany w literaturze anglojęzycznej, 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...

zastosowaniem omawianej metody warto wziąć do ręki kartkę i ołówek, aby przekonać się, czy w ogóle warto zasiadać do klawiatury!

Oto formalny zapis metody zaprezentowany przy pomocy pseudo-języka programowania:

```
dziel_i_rządź (N)
{
    jeśli N wystarczająco mały
        zwróć Przypadek_Elementarny (N) ;
    w przeciwnym wypadku
    {
        „Podziel” Pb (N) na mniejsze egzemplarze:
        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 paragrafach.

9.1.1. Odszukiwanie minimum i maksimum w tablicy liczb

Z metodą „dziel-i-rządź” mieliśmy już w tej książce do czynienia w sposób niejawny 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 w 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!

Ćwicz. 9-1

Proszę wyprowadzić wzory tłumaczące wartości (x_i, y_i) na współrzędne ekranowe (x_{ekr}, y_{ekr}) , znając rozdzielcość ekranu graficznego X_{max} i Y_{max} oraz maksymalne odchylenia wartości funkcji $f(x)$, które oznaczymy jako f_{min} i f_{max} .

Powróćmy teraz do właściwego zadania. Pierwszy pomysł na zrealizowanie algorytmu poszukiwania minimum i maksimum pewnej tablicy² 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];
    }
}
```

Założmy, 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. Dorożenie *else* tuż po (*) spowoduje, że w najgorszym razie wykonamy $2(n-1)$ porównań, a w najlepszym – tylko $n-1$. Nie zmieni 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-rządź”. Idea jest następująca:

Przypadek ogólny:

- jeśli tablica ma rozmiar równy 1, to zarówno *min*, jak i *max* są równe jedynemu elementowi, który się w niej znajduje;
- jeśli tablica ma dwa elementy, poprzez ich porównanie możemy z łatwością określić wartości *min* i *max*.

² W przykładzie będzie to tablica liczb całkowitych, co nie umniejsza ogólności algorytmu.

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 powyższe 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 najwykłeszym 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 uczyni to bez trudu 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 §3.7.3).

Obliczenie złożoności praktycznej $T(n)$ tego programu nie należy do trudnych. Jeśli odpowiednio rozpiszemy równanie:

$$T(n) = 2 + 2T\left(\frac{n}{2}\right) = 2 + 2\left[2 + T\left(\frac{n}{4}\right)\right] = etc.$$

i założymy, że istnieje pewne k , takie że $n=2^k$, to otrzymalibyśmy takie samo równanie, jak w przypadku procedury *min_max1*. Nie powinno to budzić zdziwienia, biorąc 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!

Trafiłyśmy już na zapowiedziany we wstępie przypadek problemu, dla którego zastosowanie metody „dziel-i-rządź” nic 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 być dla nas jasne, że z uwagi na wprowadzenie dodatkowego obciążenia pamięci (stos wywołań rekurencyjnych) i niepomijalnej ilości 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-rządź” czyni wręcz cuda (już zaraz kilka z nich zresztą zaprezentuję...), ale o ich zaistnieniu można się przekonać jedynie wyliczając złożo-

żoność 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).

9.1.2. Mnożenie macierzy o rozmiarze $N \times N$

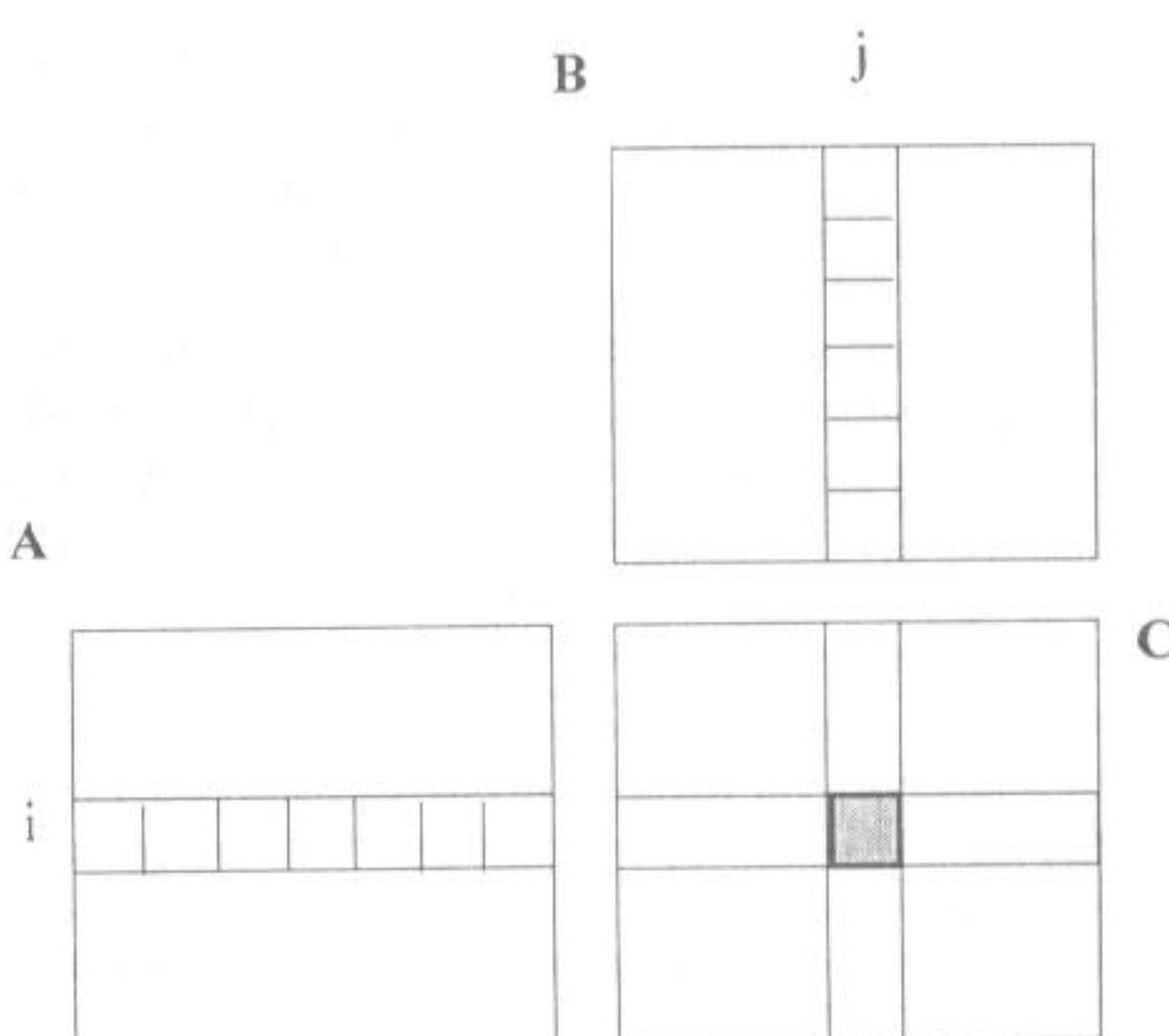
W wielu zagadnieniach natury numerycznej często zachodzi potrzeba mnożenia ze sobą macierzy, co z definicji jest dość czasochłonną operacją. Sposób wyliczania iloczynu dwóch macierzy może być symbolicznie przedstawiony w sposób zaprezentowany na rysunku 9 - 1.

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).

Rys. 9 - 1.
Mnożenie
macierzy.



Koszt wyliczenia jednego elementu macierzy C , mając na uwadze ilość wykonywanych operacji mnożenia (przyjmijmy rozsądnie, że to one są tu „najkosztowniejsze”), jest równy oczywiście N . Ponieważ wszystkich elementów jest N^2 , to koszt całkowity wyniesie N^3 , czyli program 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-rządź”, który był lepszy niż wydawałoby się „nienaruszalne” $O(N^3)$.

Oznaczmy elementy macierzy A , B i C w sposób następujący:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \bullet \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

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 macierzy A i B na 4 równe części (zakładając oczywiście, że N jest potęgą liczby 2...) i wykonanie mnożenia macierzy mniejszego rzędu wydaje się bezpośrednim zastosowaniem techniki „dziel-i-rządź”. 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 macrycami rzędu $\frac{N}{2}$: P, Q, R, S, T, U, V, służących do zapamiętania wyników następujących obliczeń³:

³ Obok równań są zaznaczone operacje arytmetyczne wymagane do wyliczenia danej równości.

$$\begin{aligned}
 P &= (A_{11} + A_{22})(B_{11} + B_{22}) & *, +, + \\
 Q &= (A_{21} + A_{22})B_{11} & +, * \\
 R &= A_{11}(B_{12} - B_{22}) & *, - \\
 S &= A_{22}(B_{21} - B_{11}) & *, - \\
 T &= B_{22}(A_{11} + A_{12}) & *, + \\
 U &= (A_{21} - A_{11})(B_{11} + B_{12}) & *, +, - \\
 V &= (A_{12} - A_{22})(B_{21} + B_{22}) & *, +, -
 \end{aligned}$$

Gdy mamy te cząstkowe wyniki, otrzymanie matrycy 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 $*$ i dodatkowych 18 operacji $+$ lub $-$.

Zauważmy, że algorytm Strassena przenosi w inteligentny sposób ciężar obliczeń z zawsze kosztownej operacji mnożenia⁴ na znacznie szybsze dodawanie lub odejmowanie.

Rozkład rekurencyjny w algorytmie V. Strassena jest następujący:

$$T(n) = 7T\left(\frac{N}{2}\right) + aN^2 \in O(N^{2.81})$$

(a jest pewną stałą).

Bliższe badania praktyczne tego algorytmu wykazały, że realny zysk powyższej metody daje się zauważać w przypadku mnożenia macierzy dla N rzędu kilkudziesiąt, 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”

⁴ Zwłaszcza jeśli elementami macierzy są liczby „rzeczywiste” (typ *double* w C++).

pozornie całą żądaną pamięć, faktycznie operuje jej „stronami”, które dosyła mu w miarę potrzeb 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 „zauważalności” roli operacji dodawania i odejmowania. Z wymienionych wyżej względów algorytm V. Strassena należy raczej traktować jako ciekawy wynik teoretyczny, o niepodważalnych walorach edukacyjnych!

9.1.3. 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 $O(n)$).

Metoda „dziel-i-rządź” w przypadku mnożenia liczb całkowitych może być zastosowana po dokonaniu następującej obserwacji:

$$X = [A \ B] = A * 2^{\frac{N}{2}} + B,$$

$$Y = [C \ D] = C * 2^{\frac{N}{2}} + D.$$

A i B oraz C i D oznaczają odpowiednio „połówki” reprezentacji binarnych liczb X i Y . Iloczyn $X*Y$ 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 dodawaniem i przesu-

nięciami bitowymi⁵. 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? Wynika ona z obserwacji wynikowej podczas analizy procedury *min_max*: dokonaliśmy podziału problemu, ale w niczym nie zmniejszyliśmy ilości wykonywanej pracy. Cudów zatem nie będzie⁶! Zanim jednak rozczarujemy się na dobre do metody „dziel-i-rządź”, popatrzmy na następujące „przepisanie” operacji X^*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 Strassena!) zmniejszyliśmy ilość 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 co do kosztów operacji elementarnych (+, -, przesunięcie bitowe), kiedy istotnie ten algorytm może dać „zauważalne” rezultaty w porównaniu z metodą klasyczną.

9.1.4. Inne znane algorytmy „dziel-i-rządź”

Nie eksponując specjalnie tego, już w rozdziałach poprzednich mogliśmy zapoznać się z kilkoma ciekawymi algorytmami, które można zaklasyfikować do metody „dziel-i-rządź”.

Programem, który zdecydowanie „króluje” wśród nich, jest niewątpliwie słynny *QuickSort* (patrz opis). 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 derekusywacji *problem wież Hanoi* (patrz rozdział 6) jest również dobrym przykładem intelligentnej dekompozycji

⁵ 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...).

⁶ Dla niedowiarków dowód matematyczny: $T(n) \in O(N^{\log_2 4}) = O(N^2)$.

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-rządź”, choć „filozoficznie” różni się nieco od schematu ze strony 225. Jest ona dobrym przykładem na to, jak dobry algorytm może przyspieszyć rozwiązanie postawionego problemu, dla którego znana jest prosta, ale nieefektywna metoda (przeszukiwanie liniowe).

9.2. Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas...

Nazwa nowej metody jest bardzo intrugująca, ale w literaturze przedmiotu przyjęło się nazywać pewną klasę metod jako „żarłoczne” (ang. *greedy*, franc. *glouton*). 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 w 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 jego szybkim i w miarę po-prawnym rozwiązaniem, choć niekoniecznie optymalnym.

Schemat generalny algorytmu jest następujący:

```

żarłok(W)
{
    ROZW=∅ ; // zbiór pusty
    dopóki(nie Znaleziono(ROZW) i W≠∅ )
        wykonuj:
        {
            X=Wybór(W);
            jeśli Odpowiada(X) to ROZW=ROZW ∪ {X};
        }
        jeśli Znaleziono(ROZW) zwróć ROZW;
    }
}

```

w przeciwnym wypadku 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 kąsek, nie troszcząc się specjalnie o przyszłość... Popatrzmy na kilka przykładów zastosowania nowej metody.

9.2.1. 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 wiktualów, tak aby umówioną imprezę uczynić iście 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. Założymy, 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 najwartościowski ładunek, tzn. zmaksymalizować funkcję $\sum_{i=1}^3 c_i x_i$.

Aby rozważania uczynić mniej teoretycznymi, popatrzmy na konkretny przykład kilku konfiguracji danych:

$$\begin{array}{lll} w_1=16, & w_2=12, & w_3=10, \\ c_1=80, & c_2=70, & c_3=60. \end{array}$$

Kilka przykładów zamieszczonych w tabelce 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, nic nam jednak nie gwarantuje, że nie istnieją lepsze konfiguracje parametrów x_i .

Trzeba w tym miejscu być może podkreślić, że podstawowa idea, na której została zbudowana procedura żartok, nie gwarantuje optymalności.

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

Wręcz przeciwnie, w typowym przypadku otrzymane rozwiązanie¹ będzie tylko prawie optymalne!

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 meta-algorytmów, każda próba bezmyślnego ich stosowania spali (prawdopodobnie) na panewce.

¹ Jeśli oczywiście istnieje!

Aby to dokładniej zilustrować, przeanalizujmy kilka możliwych strategii rozwiązania problemu plecakowego przy użyciu algorytmu „żarłoczniego”. Pierwsze, pozornie optymalne rozwiązanie polega na próbach wypełniania plecaka przy pomocy najdroższego sera (*s1*): 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 przy pomocy tej metody, aby się przekonać, że nie daje ona najlepszych rezultatów. Najlepszym przykładem może tu być analiza tabelki 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 ręku” 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 nasycić 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 n=3;
void greedy(double M, double W[n], double C[n], double X[n])
{
    double Z=M; // pozostaje do wypełnienia
    for(int 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];
}
void 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 << "Total:" << 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.

9.3. 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 formula rekurencyjna zostanie wykonana przez komputer, ile będzie „kosztowało” zrealizowanie wywołań rekurencyjnych, powrotów z nich, kombinowanie rezultatów cząstkowych etc.

Mожет 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 wykorzystanie korzyści, płynących z rekurencyjnego formułowania rozwiązań, bez używania rekurencji?*

Wbrew pozorom nic jest to paradoks – technika *programowania dynamicznego* bazuje właśnie na tym – zdawałoby się niemożliwym do zrealizowania – postulacie. Nadaje się ona szczególnie dobrze do rozwiązywania problemów o charakterze numerycznym:

- obliczanie najkrótszej drogi w grafach (które poznamy szczegółowo w rozdziale 10);
- wyliczenie pewnej skomplikowanej wartości podanej przy pomocy równania rekurencyjnego...

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 pod-problemów, 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 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-rządź”.

„dziel-i-rządź”

- problem rzędu N rozłoż na pod-problemy mniejszego „kalibru” i rozwiąż je;
- połącz rozwiązania pod-problemów w celu otrzymania rozwiązania globalnego.

„programowanie dynamiczne”

- mając dane rozwiązanie problemu elementarnego, wylicz na jego podstawie
- problem wyższego rzędu i kontynuuj obliczenia, aż do otrzymania rozwiązania rzędu N .

Nowa technika ma pewien posmak optymalności: raz znalezione rozwiązanie pewnego pod-problemu zostaje zarejestrowane w tablicy i w miarę potrzeb jest później wykorzystywane. Nie był to bynajmniej przypadek metody „dziel-i-rządź”, która pozwalała na wielokrotne wyliczanie tych samych wartości.

Nowo poznaną metodę zilustrujemy dwoma przykładami o różnym stopniu skomplikowania, zaczynając od... doskonale nam znanego problemu obliczania elementów ciągu Fibonacciego (patrz §2.4.1). Przypomnimy (po raz kolejny) definicję tego ciągu:

$$fib(0) = 1,$$

$$fib(1) = 1,$$

$$fib(n) = fib(n-1) + fib(n) \text{ gdzie } n \geq 2$$

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;

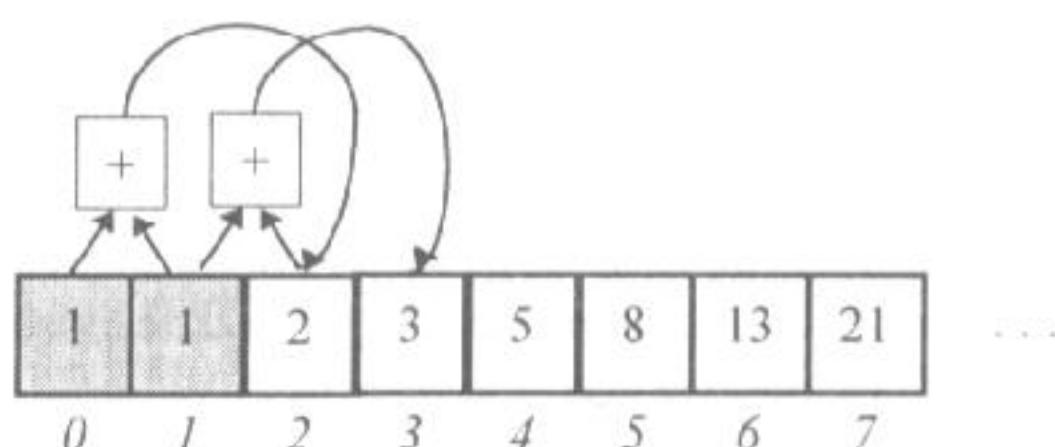
inicjacja – początkowymi wartościami w tablicy fib będą oczywiście warunki początkowe: $fib[0]=1$ i $fib[1]=1$;

progresja algorytmu – ten punkt zależy ścisłe od wzoru rekurencyjnego, który implementujemy przy pomocy 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.

Rys. 9 - 2.
Obliczanie wartości ciągu liczb Fibonacciego.



Zupełnie już dla formalności podam procedurę, która realizuje omówione wcześniej obliczenia:

```
void fib_dyn(int x, int f[])
{
    f[0]=1;
    f[1]=1;
    for(int i=2; i<x; i++)
        f[i]=f[i-1]+f[i-2];
}
```

Nieco bardziej skomplikowana sytuacja występuje w przypadku równań rekurencyjnych posiadających więcej niż jedną zmienną. Patrzmy na następujący wzór:

$$P(i, j) = \begin{cases} 1 & \text{jeśli } i = 0 \text{ oraz } j > 0, \\ 0 & \text{jeśli } i > 0 \text{ oraz } j = 0, \\ \frac{P(i=1, j) + P(i, j-1)}{2} & \text{jeśli } i > 0 \text{ oraz } j > 0. \end{cases}$$

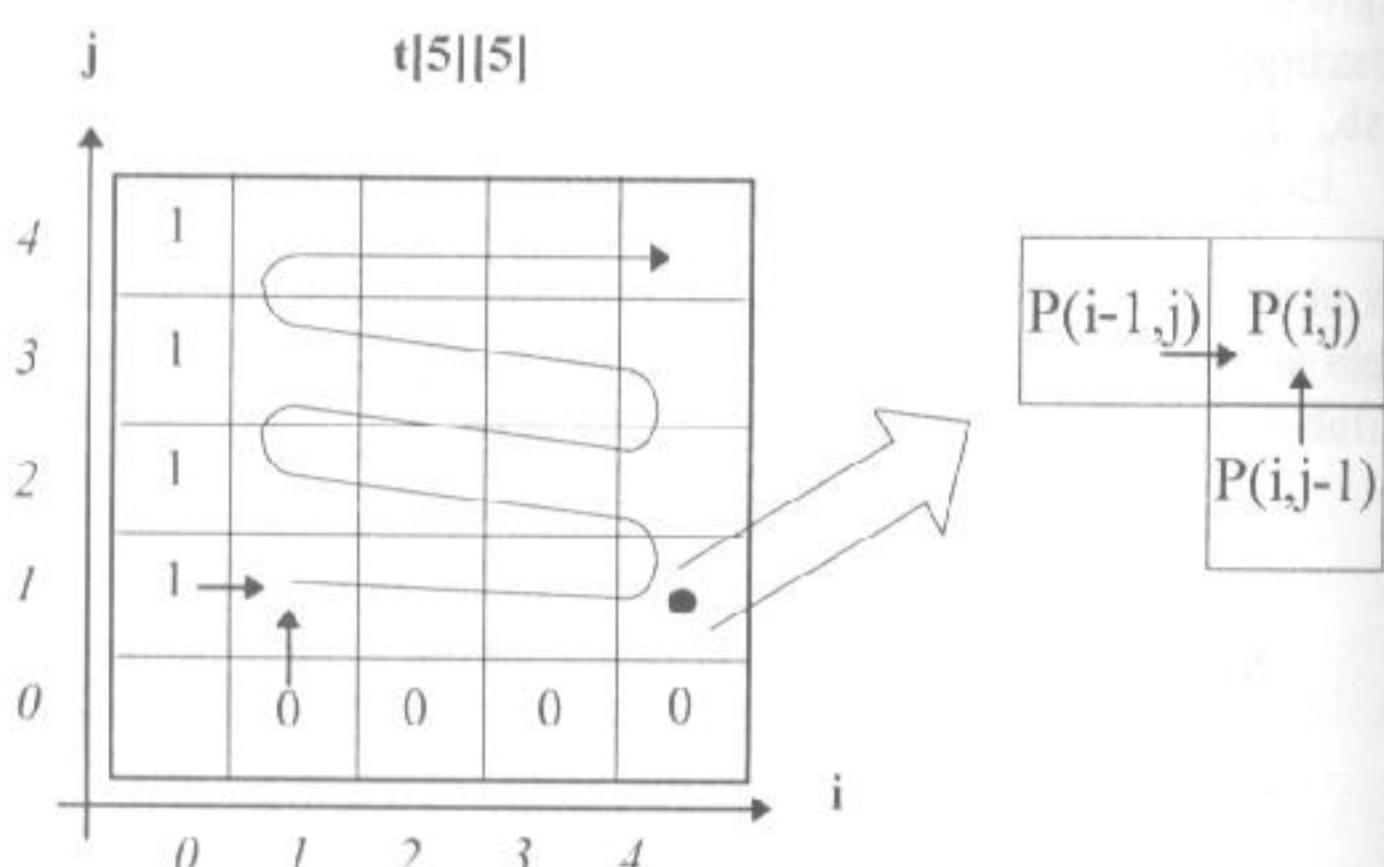
Mamy tu do czynienia z dwiema zmiennymi, i oraz j , interesuje nas obliczenie wartości parametru P . Powyższy wzór jest dość nieprzyjemny już na pierwszy rzut oka – można również udowodnić, że jest bardzo kosztowny, jeśli chodzi o czas obliczeń. Mamy zatem doskonaty przykład dowodzący, że jeśli nie musimy stosować rekurencji, to najlepiej byłoby tego w ogóle nie czynić... pod warunkiem posiadania alternatywnych dróg rozwiązania.

Technika programowania dynamicznego taką drogą podpowiada. Sposób obliczenia wzoru rekurencyjnego jest trywialny, jeśli wpadniemy na pomysł użycia tablicy dwuwymiarowej, której „współrzędne” pozioma i pionowa będą odpowiadać zmiennym i oraz j . Patrzmy 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 wstępnie 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.

Rys. 9 - 3.
„Dwuwymiarowy”
wzór rekurencyjny.



Gdy mamy te wszystkie informacje, realizacja programowa jest natychmiastowa:

```
const n=5;
void dynam(double P[n][n])
{
    int i, j;
    for(i=1; i<n; i++) // inicjacja
    {
        P[i][0]=0;
        P[0][i]=1;
    }
    for(j=1; j<n; j++) // progresja
        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 rekurencji dwu- i więcej wymiarowych (jeśli możemy sobie na takie określenie pozwolić...) możemy bardzo łatwo popełnić błąd polegający na próbie wykorzystania wartości z tablicy, które w danym etapie nie są jeszcze obliczone. Tego typu potknięcia są czasami bardzo trudne do wykrycia, więc warto przy tym szczególnie uważać.

9.4. Uwagi bibliograficzne

W tym rozdziale mieliśmy okazję poznać kilka prostych technik programowania, których efektywne użycie może znacznie zwiększyć sprawność programisty w rozwiązywaniu problemów przy pomocy komputera. Oczywiście, nie są to wszystkie meta-algorytmy, które można napotkać w literaturze problemu – wybór padł na te techniki, które nie są zbyt trudne do pojęcia i 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 szczegółowo 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. Nie jest przesadą stwierdzenie, iż bez tej struktury danych niemożliwe byłoby rozwiązanie wielu problemów algorytmicznych.

Grafy posiadają dość złożoną podbudowę teoretyczną (w zasadzie można nawet wyodrębnić osobny dział matematyki tylko im poświęcony), ale w naszej prezentacji postaramy się uniknąć zbytniego formalizowania.

Odrobiona 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 potwierdzić, czy tytuł ten jest już dostępny na rynku wydawniczym w formie książkowej, czy też pozostał na zawsze uproszczonym skryptem uczelnianym.

Celem tego rozdziału jest zaprezentowanie minimalnej 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 niemieckiemu matematykowi L. Eulerowi. Przy ich pomocy rozwiązał on problem, który stawiali sobie mieszkańcy Koenigsberg, a mianowicie jak przemierzyć wszystkie siedem mostów znajdujących się w tym mieście, tak aby nie przechodzić dwukrotnie przez ten sam.

Ta historyczna anegdota stanowi jednocześnie doskonały przykład na to, do czego grafy mogą się w praktyce przydać: wszelkie zadania algorytmiczne, w których w grę wchodzą problemy odnajdywania (optymalnych) dróg, mogą być przez grafy doskonale modelowane. Oczywiście nie tylko one!

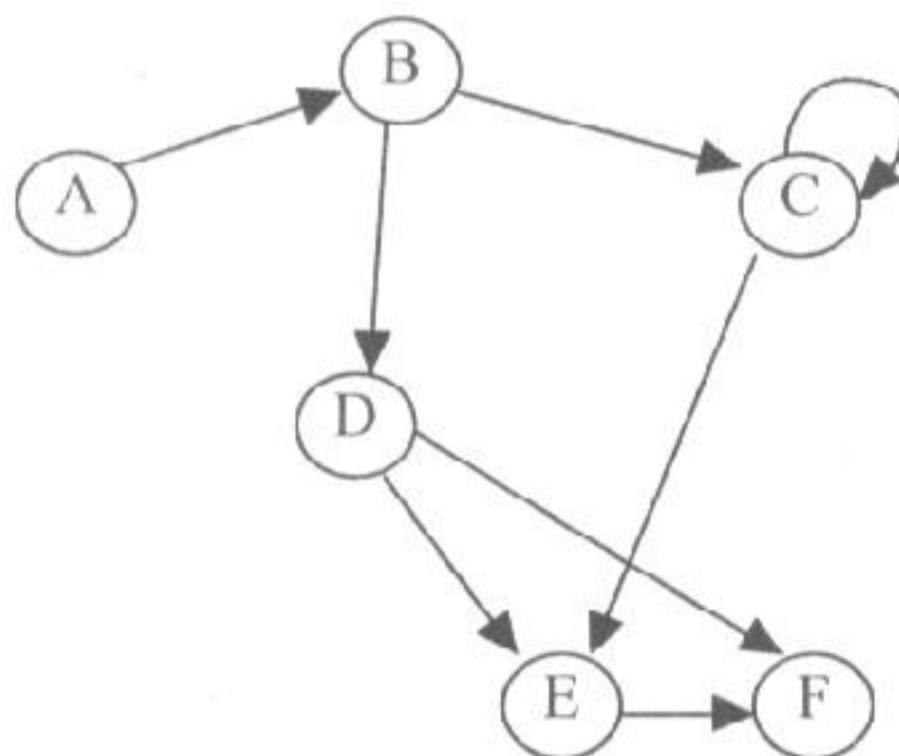
Programista, który dobrze pozna i zrozumie możliwości związane z użyciem grafów, praktycznie podwaja swoje kompetencje związane z umiejętnością sprawnego *modelowania* problemów do rozwiązywania. Dość paradoksalną stroną wielu zagadnień programistycznych jest to, że potrafią one rozwiązywać się niemalże „same” pod warunkiem dobrego zmodelowania całości.

Następny paragraf będzie poświęcony podstawowemu słownictwu związanemu z grafami, po czym przejdziemy do sposobów reprezentowania ich w programach komputerowych.

10.1. Definicje i pojęcia podstawowe

Niezbyt duże grafy doskonale dają się przedstawiać w postaci wiele mówiących rysunków, takich jak np. 10 - 1.

Rys. 10 - 1.
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 **skierowany**, jeśli krawędziom został przypisany jakiś kierunek (na rysunkach symbolizowany przez strzałkę).

Jeśli weźmiemy pod uwagę dwa węzły grafu, x i y , połączone krawędzią, to węzeł x jest **węzłem początkowym**, a węzeł y węzłem **końcowym**.

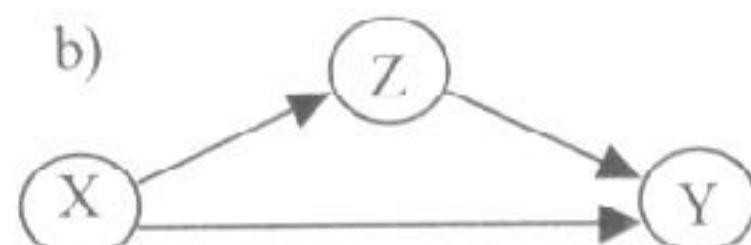
Graf z rysunku 10 - 1 posiada 6 węzłów: A, B, C, D, E i F , niektóre z nich są połączone pomiędzy sobą krawędziami: $(A,B), (B,C), (B,D), (D,F), (D,E)$ i (E,F) .

Węzeł C ma charakter specjalny, 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 przy ich pomocy modelować więcej sytuacji niż tylko z użyciem samych węzłów i krawędzi.

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).

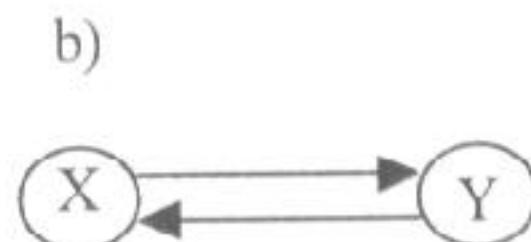
Z definicji pomiędzy dwoma węzłami może istnieć tylko jedna krawędź, ale możliwe jest bardzo łatwe przejście z grafu, który nie jest zgodny z naszą definicją (patrz 10 - 2 (a)), do grafu „standardowego” poprzez zwykłe dołożenie sztucznych wierzchołków (rysunek 10 - 2 (b)).

Rys. 10 - 2.
„Normalizowanie”
grału(1).



Pojęcie grafu skierowanego ma charakter najogólniejszy, gdyż graf **nieskierowany** (patrz rysunek 10 - 3 (a)) może być bardzo łatwo przetransformowany na skierowany (rysunek 10 - 3 (b)).

Rys. 10 - 3.
„Normalizowanie”
grału(2).



Dla pewnych zastosowań celowe jest przypisanie krawędziom grafu wartości (najczęściej liczbowych, ale mogą to również być etykiety innego typu). Zmienia nam się wówczas definicja grafu, gdyż zamiast dwójki (X, Γ) mamy (X, Γ, V) . Trzeci parametr V oznacza właśnie zbiór wartości odpowiadających danym krawędziom.

W teorii grafów można napotkać jeszcze sporo innych definicji i pojęć, ale my chwilowo poprzestaniemy na tych zaprezentowanych powyżej. Nowe pojęcia, jeśli okażą się niezbędne, będą systematycznie wprowadzane w trakcie wykładu.

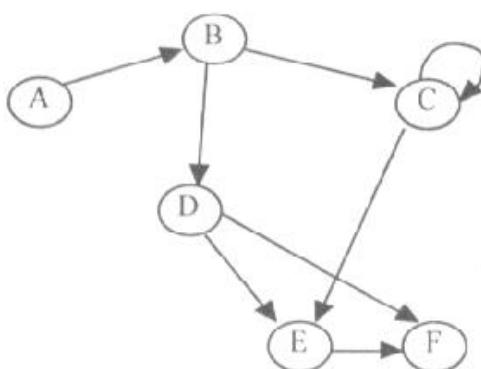
Obecnie przejdziemy do opisu kilku typowych metod reprezentowania grafów w pamięci komputera.

10.2. 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: przy pomocy *tablicy dwuwymiarowej* i tzw. *słownika węzłów*.

Graf może być reprezentowany przy użyciu tablicy dwuwymiarowej, jeśli umówimy się, że wiersze będą oznaczały węzły początkowe krawędzi grafu, a kolumny ich węzły końcowe. Przytakując takie ustawienie, graf z rysunku 10 - 1 może być przedstawiony w postaci tablicy z rysunku 10 - 4.

Rys. 10 - 4.
Tablicowa reprezentacja grafu.



	A	B	C	D	E	F
A		1				
B			1	1		
C				1	1	
D						1
E						
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.

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.

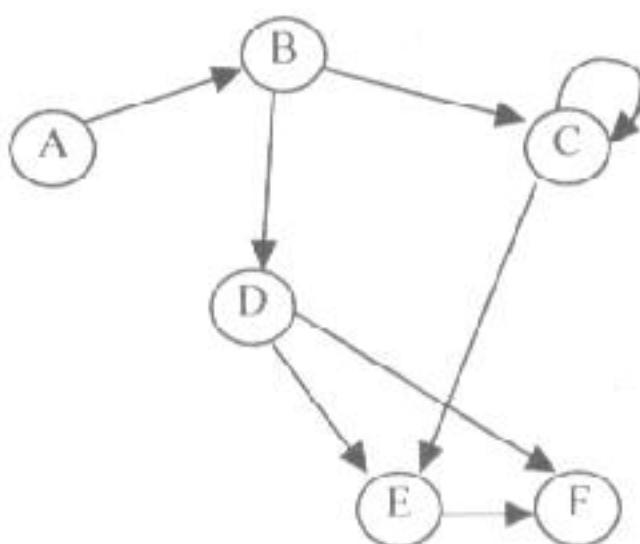
Aby przedstawić graf o liczbie węzłów, która może ulegać zmianie w trakcie wykonywania się programu, należy użyć np. reprezentacji przy pomocy *słownika węzłów*.

Słownik węzłów może dotyczyć dwóch typów węzłów: *następników* (węzłów *odchodzących*) lub *poprzedników* (węzłów *dochodzących*) od danego węzła. Idea jest przedstawiona na rysunku 10 - 5.

Słownik jest zwykłą tablicą wskaźników do list węzłów, odpowiednio *odchodzących* (a) lub dochodzących (b) do danego węzła przy pomocy krawędzi. Niektóre algorytmy dotyczące grafów potrzebują właśnie tego typu informacji, stąd celowość dysponowania taką reprezentacją. Biorąc pod uwagę, że słownik węzłów jest łatwo implementowalny w postaci listy list, znika nam automatycznie problem napotkany przy reprezentacji tablicowej – ilość węzłów grafu może być w zasadzie nieograniczona.

Rys. 10 - 5.

Reprezentacja grafu przy pomocy słownika węzłów.



a)

A	B
B	C,D
C	C,E
D	E,F
E	F
F	NULL

b)

A	NULL
B	A
C	B,C
D	B
E	C,D
F	D,E

10.3. Podstawowe operacje na grafach

Wiele algorytmów dotyczących grafów daje się łatwo wyrazić przy użyciu specjalnej notacji matematycznej, którą poznamy właśnie w tym paragrafie¹.

Mając dane dwa grafy $G1=(X, \Gamma_1)$ i $G2=(X, \Gamma_2)$, możemy na nich zdefiniować następujące operacje:

- **suma grafów**

$$G3=G1+G2=(X, \Gamma_1 \cup \Gamma_2).$$

Graf wynikowy $G3$ zawiera wszystkie krawędzie grafów $G1$ i $G2$.

- **kompozycja grafów**

$$G_3=G_1 \circ G_2 = (X, \Gamma_3 = \{(x, y) | \exists z \in X : (x, z) \in \Gamma_1 \text{ oraz } (z, y) \in \Gamma_2\}).$$

Krawędzie (x, y) grafu wynikowego $G3$ spełniają warunek, że istnieje pewien węzeł z , taki, że (x, z) należy do $G1$, a (z, y) należy do $G2$.

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++)
    {
        g3[x][y] = 0;
        for(int z=0;z<n;z++)
            if(g1[x][z]&&g2[z][y])
                g3[x][y] = 1;
    }
}
  
```

¹ Warto w tym miejscu „odświeżyć” oznaczenia matematyczne poznane w rozdziale 3.

```

{
z=0;
while(1) // pętla nieskończona
{
    if(z==n)
        break;
    if((g1[x][z]==1)&&(g2[z][y]==1))
    {
        g3[x][y]=1;
        break;
    }
    z++;
}
}
}

```

• 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 sama w sobie dość paradoksalnie nic nie oznacza. Jest ona po prostu dość 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. Zauważmy, że domknięcie przechodnie nie daje przepisu na przejście od danego wierzchołka do wierzchołka y : dowiadujemy się tylko, że 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 ilość wierzchołków grafu, czyli nieco formalniej $n=|X|$).

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.

10.4. 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:

$$\Theta_k = \Gamma \cup (y, z) | (x, k) \in \Gamma \text{ oraz } (k, y) \in \Gamma.$$

Zapis powyższy oznacza, że dla danego wierzchołka k do zbioru krawędzi Γ dorzucamy krawędzie łączące poprzedniki i następcy 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++:

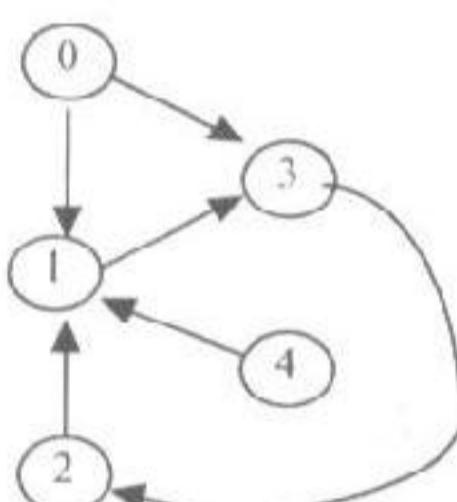
warshall.cpp

```
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 - 6.

Rys. 10 - 6.

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	
1		x	x	x	
2	x		x	x	
3	x		x	x	
4	x	x	x	x	

Efekt wykonania
algorytmu Roy-Warshalla

(Zamiast tradycyjnych „jedynek” na rysunku zostały użyte znaki X).

Z tablicy na rysunku 10 - 6 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 określamb 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 *które*dy należy pójść...

W celu zidentyfikowania drogi (oczywiście jeśli w ogóle ona istnieje!) przyporządkujemy macierzy reprezentującej graf tzw. *macierz kierowania ruchem* (ang. *routing*) 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 matrycy 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 << "Drogi nie ma\n";
    else
    {
        cout << x << endl;
        k=x;
        while(k!=y)
        {
            k=R[k][y];
            cout << k << endl;
        }
    }
    cout << endl;
}
```

Wiemy już, jak wypisać drogę na podstawie macierzy R , najwyższa zatem pora na przedstawienie algorytmu, który ją prawidłowo dla danego grafu wylicza.

Przedstawiona poniżej procedura *route* „zakłada”, że matryca 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 przypadku.

Zapis procedury jest ekstremalnie 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];
}

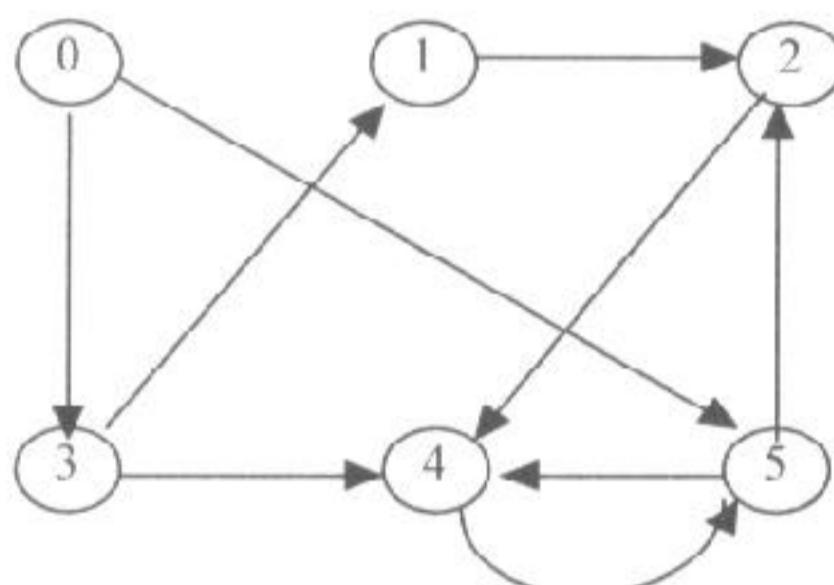
```

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 - 7.

Droga od 0 do 2: 0 3 1 2
 Drogi od 1 do 0: Drogi nie ma
 Droga od 1 do 5: 1 2 4 5
 Drogi od 2 do 0: Drogi nie ma
 Droga od 4 do 2: 4 5 2
 Droga od 5 do 3: Drogi nie ma

Rys. 10 - 7.
 Poszukiwanie drogi w grafie.



Kolejnym problemem, który omówimy, jest znajdowanie w grafie drogi optymalnej pod względem kosztów.

10.5. Algorytm Floyda

Nietrudno jest domyślić się, że nasze nowe zadanie będzie wymagało użycia grafów, które charakteryzują się przypisaniem wartości liczbowych swoim krawędziom.

Algorytm Floyda zaprezentujemy przy użyciu następujących założeń:

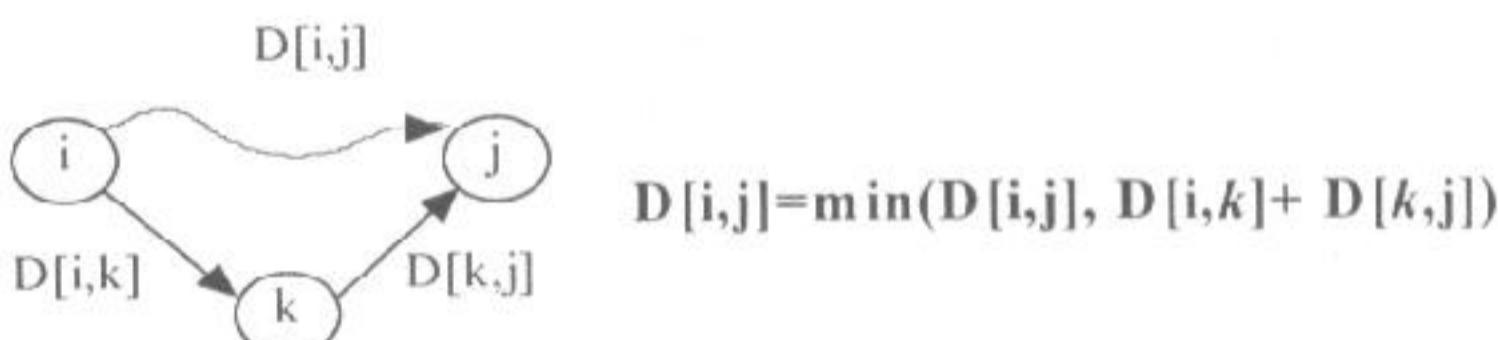
- Dysponujemy macierzą W , w której są zapamiętane wartości przypisane krawędziom grafu;
- $W[i,i]=0$;

- $W[i,j]$ – wartość przypisana krawędzi lub ∞ (inaczej: bardzo dużą liczbą);
- wartość optymalnej drogi będzie zapamiętywana w matrycy D ;

Ideę algorytmu w zrozumiały sposób prezentuje następujący przykład:

Założymy, 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? Patrzmy na rysunek 10 - 8, który przedstawia odpowiedź na to pytanie w nieco bardziej poglądowej formie niż goły wzór matematyczny (przedstawiony obok).

Rys. 10 - 8.
Algorytm Floyda
(l.)



Jest oczywiste, że w przypadku większej ilości takich „optymalnych” wierzchołków pośrednich należy wybrać najlepszy z nich!

Przedstawiony poniżej program jest najprostszą formą algorytmu Floyda, która wyłącznie *oblicza* wartość optymalnej drogi, ale jej nie *zapamiętuje*.

floyd.cpp

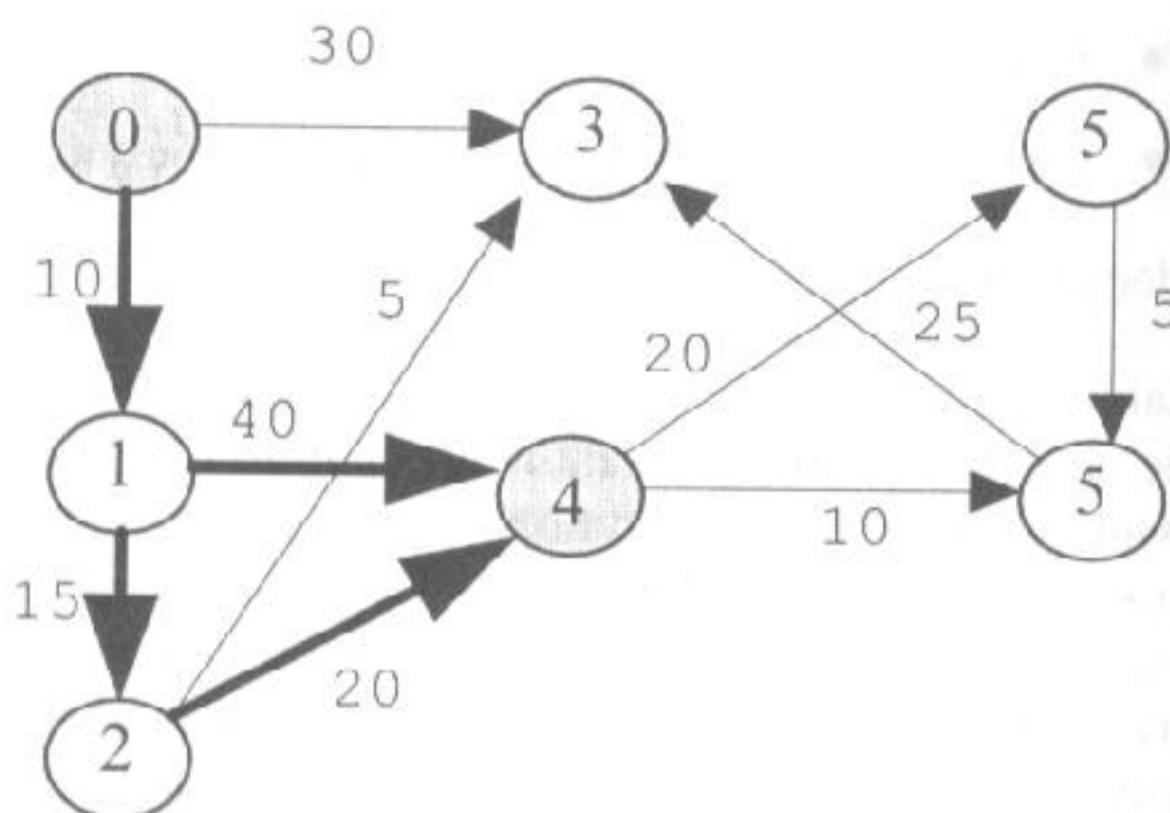
```
void floyd(int g[n][n])
{
    for(int k=0; k<n; k++)
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                g[i][j]=min( g[i][j], g[i][k]+g[k][j]);
}
```

Patrzmy na rysunek 10 - 9, który przedstawia przykład wyboru optymalnej drogi przez algorytm Floyda.

Założymy, że interesuje nas optymalna droga od wierzchołka nr 0 do wierzchołka numer 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.

Elementarne obliczenia wykazują, że druga trasa jest efektywniejsza (koszt: 45) od pierwszej (koszt: 50).

Rys. 10 - 9.
Algorytm Floyda
(2).



Brak możliwości odtworzenia optymalnej drogi jest dość istotną wadą, gdyż o ile w przypadku małych grafów (takich jak ten z rysunku 10 - 9) możemy ją ewentualnie odczytać sami, to przy naprawdę dużych grafach jest to praktycznie niewykonalne.

Potrzebna nam jest zatem jakaś prosta modyfikacja algorytmu Floyda, która nie zmieniając jego zasadniczej idei, umożliwi zapamiętanie drogi.

Jak się okazuje, rozwiążanie nie jest trudne. Do oryginalnego algorytmu (patrz listing wyżej) należy wprowadzić następującą poprawkę:

```
...
    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 matrycy kierowania ruchem R . Czytelnikowi nie powinno sprawić zbytniego kłopotu napisanie procedury, która odtwarza znajdującą się w niej drogę. Założymy, że początkowo matryca R jest wyzerowana. Aby odtworzyć optymalną drogę od wierzchołka i do wierzchołka j , patrzymy na wartość $P[i][j]$. 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 $P[i][j]$ i następnie od $P[i][j]$ do j . Z uwagi na to, że powyższe dwie „pod-drogi” mogą nie być elementarne, łatwo zauważyć rekurencyjny charakter procedury.

Liczę na to, że Czytelnik nie będzie miał kłopotów z jej stworzeniem... w ramach pozytecznego ćwiczenia!

10.6. 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 temat ten jest przydatny we wszelkich zagadnieniach związanymi 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*) i strategii „wszerz” (ang. *breadth first*). 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: odnaleźć optymalną strategię gry, rozwiązać łamigłówkę lub konkretny problem techniczny przedstawiony przy pomocy grafów...

Uwaga: nasze 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 dla bardzo dużych grafów użycie tablic jest niemożliwe. Jedynym wyjściem w takiej sytuacji jest użycie reprezentacji popartej na *słowniku węzłów*. Wiąże się z tym modyfikacja wspomnianych algorytmów, zatem dla ułatwienia zostaną również podane w pseudo-kodzie, tak aby możliwe było ich przepisanie na użytk konkretnej struktury danych.

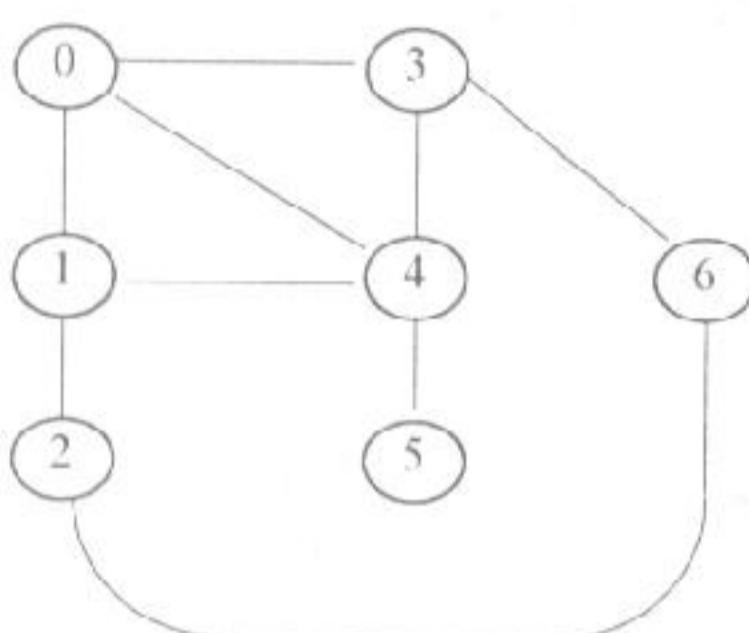
10.6.1. Strategia „w głąb”

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 (w przeciwieństwie do algorytmu „wszerz”, który najpierw bada wszystkie poziomy grafu o jednakowej głębokości)¹. Jego cechą przewodnią jest zatem *maksymalna eksploatacja raz obranej drogi, przed ewentualnym wybrianiem następnej*.

Rysunek 10 - 10. przedstawia niewielki graf, który posłuży nam za ilustrację problemu.

¹ W naszej dotychczasowej dyskusji na temat grafów nie używaliśmy, co prawda, zcytowanej powyżej terminologii, ale jej znaczenie powinno się szybko wyjaśnić podczas analizy konkretnych przykładów.

Rys. 10 - 10.
Przeszukiwanie
grafo „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 grafu².

Algorytm przeszukiwania „w głąb” zapisuje się dość prosto w C++:

depthf.cpp

```

int i, j, G[n][n], V[n];
// G - graf nxn
// V - przechowuje informację, czy dany wierzchołek
//       był już badany (1) lub nie (0)
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);
}

void zwiedzaj(int G[n][n], int V[n], int i)
{
    V[i]=1; // zaznaczamy wierzchołek jako "zbadany"
    for(int k=0; k<n; k++)
        if(G[i][k]!=0) // istnieje przejście
            if(V[k]==0)
                zwiedzaj(G, V, k);
}
  
```

Jak łatwo zauważyc, składa on się 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:

² Kolejność elementów w tej liście jest związana z użyciem reprezentacji tablicowej, w której indeks tablicy (czyli numer węzła) z góry narzuca pewien porządek wśród węzłów.

```

zwiedzaj(i)
{
    zaznacz 'i' jako "zbadany";
    dla każdego wierzchołka 'k' przyległego do 'i'
        jeśli 'k' nie był już zbadany
            zwiedzaj(k)
}

```

Uruchomienie programu 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 2 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 nastepnym 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.

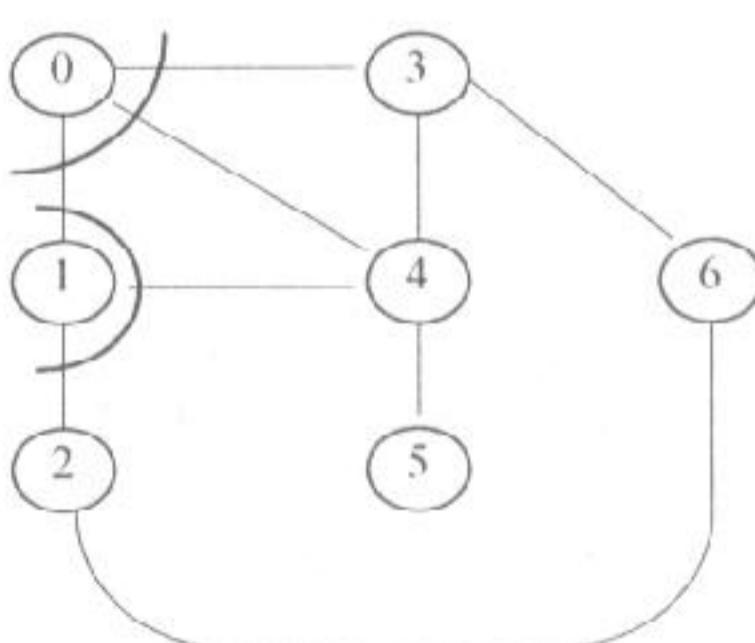
Postępując dalej tą drogą, można odtworzyć sposób pracy algorytmu przeszukiwania „w głąb” dla całego grafu.

10.6.2. Strategia „wszerz”

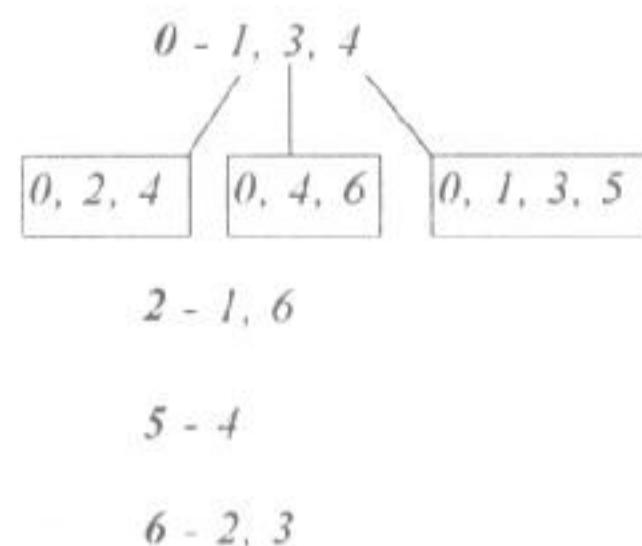
Do analizy przeszukiwania „wszerz” użyjemy takiego samego grafu jak w przykładzie poprzednim. Rysunek został jednak uzupełniony o elementy ułatwiające zrozumienie nowej idei przeszukiwania.

Rys. 10 - 11.

Przeszukiwanie grafu „wszerz”.



Lista wierzchołków przyległych:



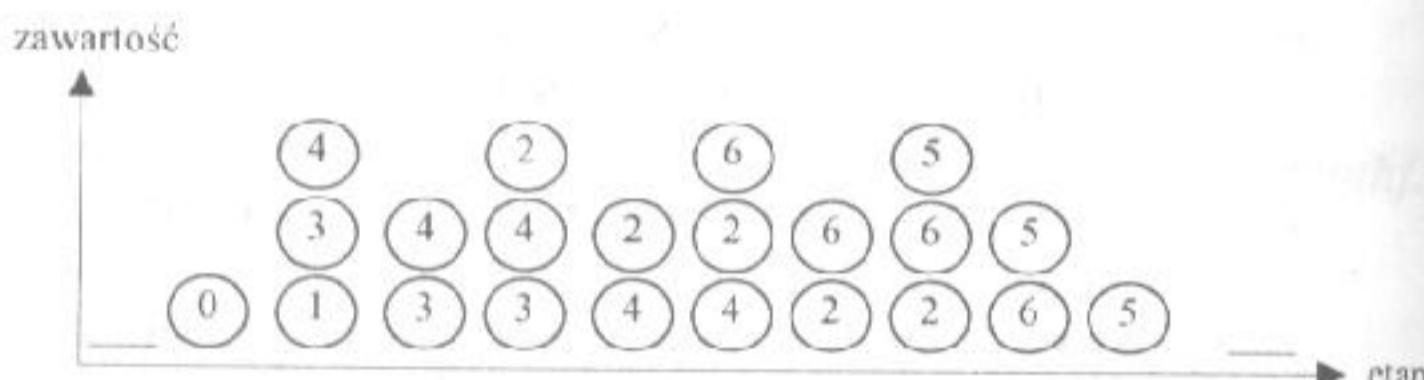
Załóżmy, że rozpoczęliśmy od wierzchołka 0. Na liście wierzchołków przyległych znajdują się kolejno: 1, 3 i 4 i te właśnie wierzchołki zostaną jako pierwsze

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 - 11, będzie taka: $0, 1, 3, 4, 2, 6$ i 5 .

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*³, która „sprawiedliwie” obsługuje wszystkie wierzchołki, zgodnie z kolejnością ich wchodzenia do kolejki (poczekalni).

Zawartość kolejki dla naszego przykładu przedstawić się będzie zatem w ten sposób:

Rys. 10 - 12.
Zawartość kolejki
podczas przeszukiwania grafu
„wszerz”.



Algorytm przeszukiwania „wszerz”, zapisany w C++, przedstawia się następująco:
breadthf.cpp

```
void szukaj(int G[n][n], int V[n], int i)
// rozpoczynamy od wierzchołka 'i'
// G - graf nxn
// V - przechowuje informację, czy dany wierzchołek
// był już badany (1) lub nie (0)
{
    FIFO<int> kolejka(n);

    kolejka.wstaw(i);

    int s;

    while(!kolejka.pusta())
    {
        kolejka.obsluz(s); // bierzemy z kolejki pewien
                            // wierzchołek 's'
        V[s]=1;           // zaznaczamy 's' jako "zbadany"

        for(int k=0; k<n; k++)
            if(G[s][k]!=0) // istnieje przejście
                if(V[k]==0) // 'k' nie był jeszcze badany
```

³ Patrz §5.4

```
    {
        V[k]=1;           // zaznaczamy 'k' jako "zbadany"
        kolejka.wstaw(k);
    }
}
```

Sens tego algorytmu może być wyjaśniony znacznie czytelniej w pseudo-kodzie:

```
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;
            }
    }
}
```

10.7. Problem właściwego doboru

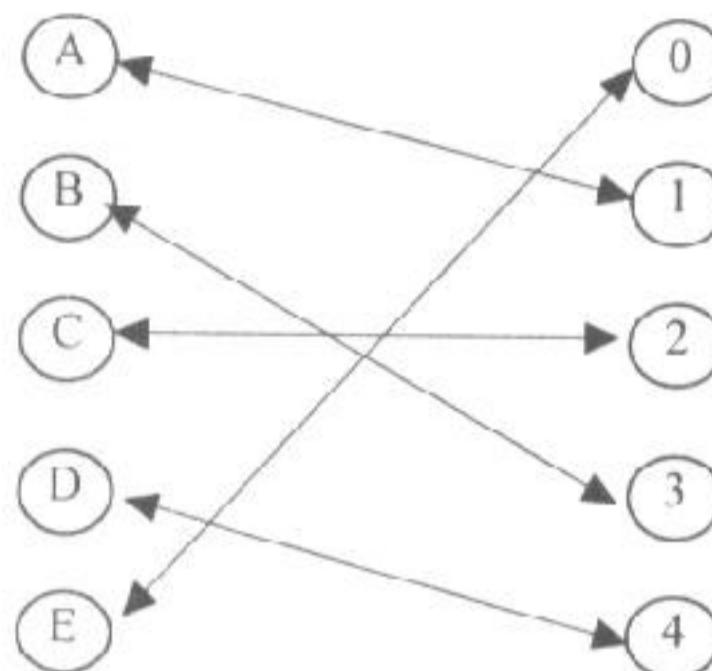
Kończąc rozważania dotyczące grafów, pragnę zaprezentować bardzo ciekawe i złożone zagadnienie: *problem doboru* (lub inaczej *minimalizowania konfliktów*). Będzie to kolejny dowód na to, że dobry model ułatwia odnalezienie właściwego rozwiązania.

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 mu kojarzącym 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 przy pomocy grafów? Cóż, chyba najlepiej zilustruje to rysunek 10 - 13.

*Rys. 10 - 13.
Problem doboru.*

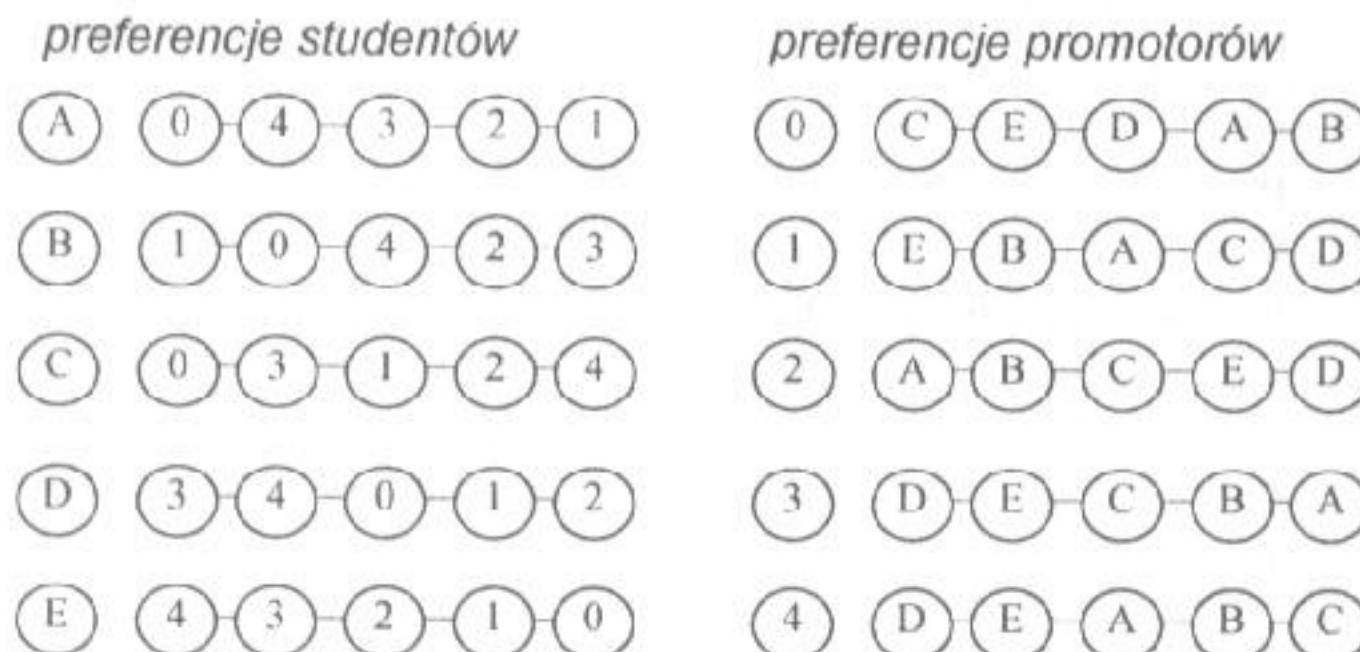


Rysunek przedstawia jedno z możliwych rozwiązań problemu doboru dla $N=5$ studentów i prac. Zadanie jest przedstawione w postaci specjalnego grafu, w którym węzły są pogrupowane według kategorii i ustawione obok siebie. Warto sobie jednak zdawać sprawę, że taka forma wizualizacji jest przydatna wyłącznie dla człowieka, gdyż komputer nie widzi różnic 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, to 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 - 14.

*Rys. 10 - 14.
Listy rankingowe
w problemie
doboru.*



Nietrudno zauważyć, że o ile samo dobranie N dwójk *{student, praca}* jest trywialne, to jednoczesne 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 wolałby 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

D woli bardziej *3* od *0*

B-3

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 dwójk wydaje się niekłopotliwe z programistycznego punktu widzenia, to sprawdzenie stabilności wydaje się dość złożone. Kłopot sprawia tu mnogość potencjalnych rozwiązań, z których każde należało 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 *częstgowego doboru*:

- student i proponuje temat j , który znajduje się najwyższej 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 zaproponować temat ze swojej listy rankingowej, *następny po uprzednio odrzuconym*.

Używając danych z rysunku 10 - 14, algorytm mógłby potoczyć się według etapów z tabeli 10 - 1.

Tabela 10 - 1.

Problem doboru na przykładzie.

propozycja	aktualne dobory	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.

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 uproszczeniu¹:

breadthf.cpp

```

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}; // rozwiązań 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}};

```

¹ Wszelkie dane liczbowe są zgodne z rysunkiem 10 - 14..

Algorytm doboru można zamknąć w rozbudowanej funkcji *main*:

```
void 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=odrzuceny;
                }
            }
        }
    }

    for(int i=0;i<5;i++)
        cout << "(Promotor " << i << ", student "
              << (char)(dobor[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.

10.8. 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 „ocenzurowany” wobec bogactwa istniejących tematów – przyda się znacznej ilości Czytelników, zachęcając ich być może do sięgnięcia po zacytowaną na początku rozdziału literaturę.

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. obliczanie układów równań, interpolacja i aproksymacja, całkowanie i różniczkowanie, przekształcenia symboliczne...)¹ i osobie szukającej wyrafinowanych możliwości można polecić zakup takiego narzędzia.

Celem tego rozdziału będzie ukazanie kilku przydatnych metod z dziedziny algorytmów numerycznych, takich, które potencjalnie mogą znaleźć zastosowanie jako część większych projektów programistycznych. Nie będziemy się zbytnio koncentrować na matematycznych uzasadnieniach prezentowanych programów, ale postaramy 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 są kompletne pod każdym względem i w zasadzie użytkownik potrzebujący konkretnej procedury może z nich korzystać jak ze swego rodzaju „książki kucharskiej”.

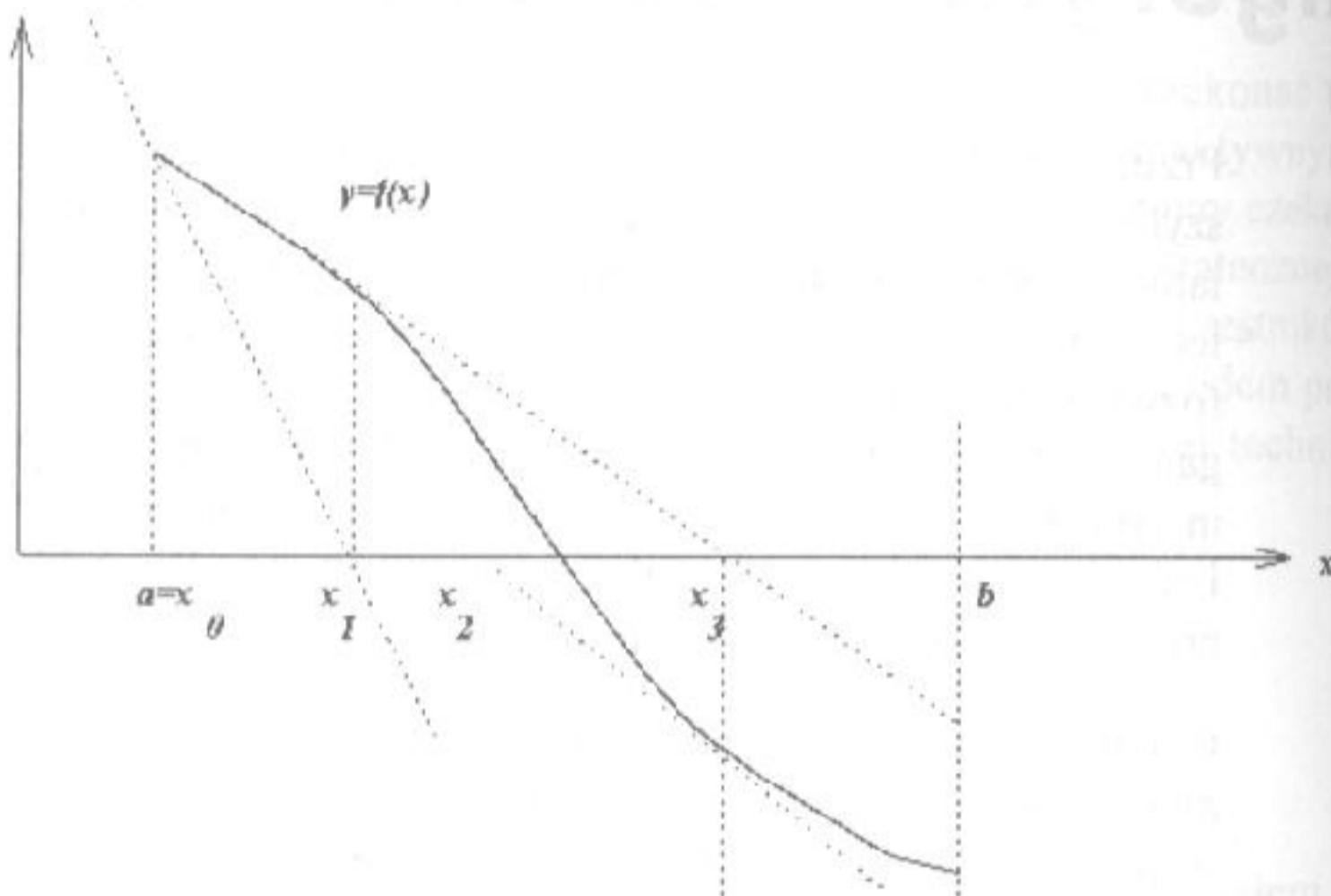
¹ Np. *Eureka*, *Mathcad*.

11.1. 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 przy pomocy komputera jest dość sporo, my ograniczymy się do jednej z prostszych – do tzw. *metody Newtona*. W skrócie polega ona na systematicznym przybliżaniu się do miejsca zerowego przy pomocy stycznych do krzywej, tak jak to pokazuje rysunek 11 - 1.

Rys. 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 (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)².

² Musimy je wpisać do kodu programu w C++ „na sztywno”.

Zaproponujemy rekurencyjną wersję algorytmu, który przyjmuje jako parametry m.in. wskaźniki do funkcji reprezentujących f i f' .

Popatrzmy dla przykładu, na obliczanie przy pomocy metody Newtona zera funkcji $3x^2 - 2$. Procedura *zero* jest dokładnym tłumaczeniem podanego na samym początku wzoru:

newton.cpp

```
const double epsilon=0.0001;

double f(double x) // funkcja f=3x2-2
{
    return 3*x*x-2;
}

double fp(double x) // pochodna f'=(3x2-2)'=6x
{
    return 6*x;
}

double zero(double x0,
            double(*f)(double),
            double(*fp)(double))
{
    if(f(x0)<epsilon)
        return x0;
    else
        return zero(x0-f(x0)/fp(x0), f, fp);
}

void main()
{
    cout << zero(1,f,fp)<<endl; // wynik 0,816497
}
```

Użycie wskaźników do funkcji pozwala uczynić procedurę *zero* bardziej uniwersalną, ale oczywiście nic stoi na przeszkodzie, aby używać tych funkcji w sposób bezpośredni.

11.2. Iteracyjne obliczanie wartości funkcji

Jak efektywnie obliczać wartość wielomianów, dowieemy 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 dysponujemy jawnym wzorem pewnej funkcji występującym w tzw. postaci uwikłanej:

$$F(x, y) = 0.$$

(funkcję w klasycznej postaci $y=f(x)$ można łatwo sprowadzić do postaci uwikłanej). Oznaczmy pochodną cząstkową, liczoną względem zmiennej y przez $F_y(x, y) \neq 0$. Przyjmując pewne uproszczenia, można za pomocą metody Newtona (patrz §11.1) obliczyć jej wartość dla pewnego x w sposób iteracyjny:

- $y_{n+1} = y_n - \frac{F(x, y)}{F_y(x, y)};$
- stop, jeśli $|y_{n+1} - y_n| < \varepsilon$

Wartość początkowa y_0 powinna być jak najbliższa wartości poszukiwanej y i spełniać warunek: $F(x, y_0) \cdot F_y(x, y_0) > 0$.

Zalety metody Newtona szczególnie uwidaczniają się w przypadku niektórych funkcji, gdzie iloraz może (ale nie musi) znacznie się uprościć. Przykładowo, dla $y = \frac{1}{x}$ mamy: $F(x, y) = 0 = x - \frac{1}{y}$ oraz $F_y(x, y) = \frac{1}{y^2}$. Po uproszczeniu wzoru iteracyjnego, powinniśmy otrzymać: $y_{n+1} = 2y_n - x(y_n)^2$.

Powyższe wzory przekładają się na program C++ w następujący sposób:

wartf.cpp

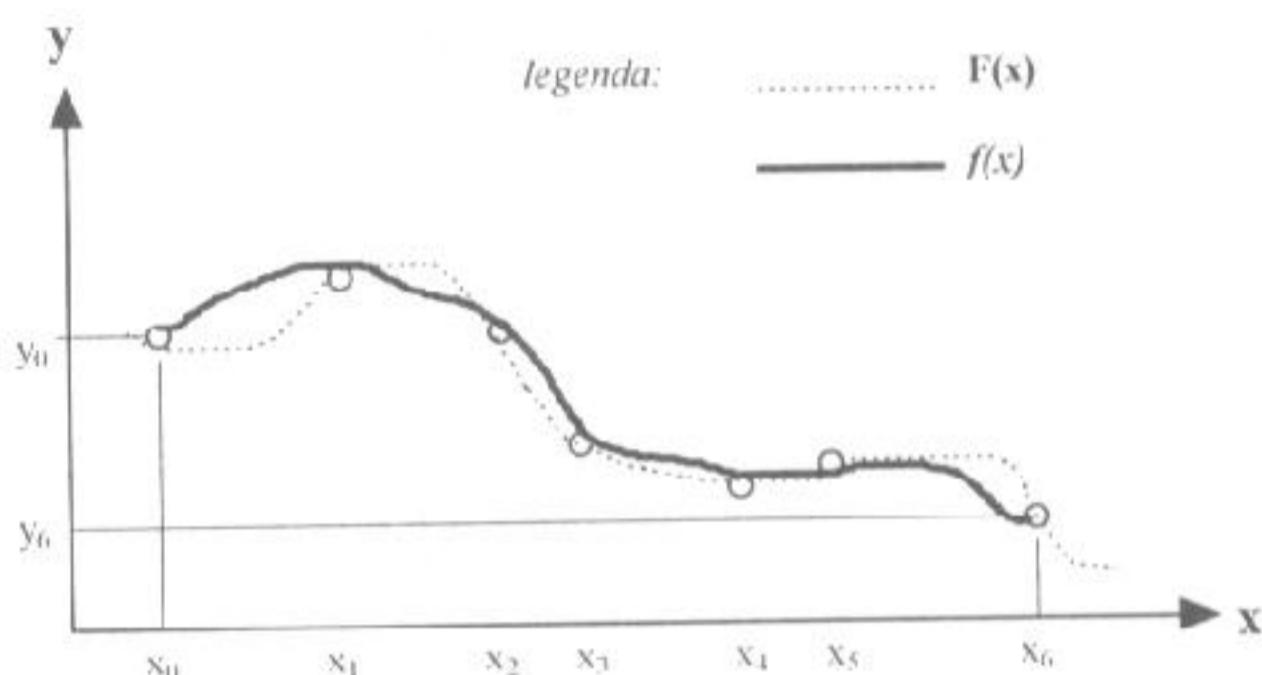
```
double wart(double x, double yn)
{
    double yn1=2*yn-x*yn*yn;
    //fabs(x)-|x|, wartość bezwzględna dla danych double
    if( fabs(yn-yn1)<epsilon)
        return yn1;
    else
        return wart(x,yn1);
}
```

11.3. 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 przy pomocy prostej funkcji (np. wielomianu określonego stopnia), tak aby funkcja interpolacyjna przechodziła *dokładnie* przez znane nam punkty wykresu funkcji jak na rysunku 11 - 2.

Rys. 11 - 2.
Interpolacja funkcji $f(x)$ przy pomocy 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 przy pomocy kłopotliwego obliczeniowo wyznacznika Vandermonde'a, który pozwala na wyliczenie współczynników poszukiwanego wielomianu. Jeśli jednak zależy nam tylko na wartości funkcji w pewnym punkcie z , to istnieje prostsza i efektywniejsza metoda Lagrange'a:

$$F(z) = (z - x_0)(z - x_1) \cdots (z - x_n) \sum_{j=0}^n \frac{y_j}{(z - x_j) \prod_{i=0, i \neq j}^n (x_j - x_i)}.$$

Pomimo dość makabrycznej postaci, wzór powyższy tłumaczy się bezpośrednio na kod C++, przy pomocy dwóch zagębianych pętli *for*:

interpol.cpp

```
const int n=3; //stopień wielomianu interpolującego

// 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 y=sqrt(x).

double interpol(double z, double x[n], double y[n])
```

```

//zwieraca wartosc funkcji w punkcie 'z' (czyli F(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;
}

void main()
{
double z=4.5;
cout << "Wartosc funkcji  $\sqrt{x}$  w punkcie " << z <<
      " wynosi " << interpol(z,x,y) << endl;
}

```

11.4. Różniczkowanie funkcji

W poprzednich paragrafach tego rozdziału bardzo często korzystaliśmy z 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 tabela 11 - 1.

Tabela 11 - I.
Tablica różnic centralnych.

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	$f(x_0+2h)$				

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:

$$f'(x) = \frac{1}{h} \left(\frac{\delta f\left(x - \frac{1}{2}h\right) + \delta f\left(x + \frac{1}{2}h\right)}{2} - \frac{1}{6} \frac{\delta^3 f\left(x - \frac{1}{2}h\right) + \delta^3 f\left(x + \frac{1}{2}h\right)}{2} + \frac{1}{30} \frac{\delta^5 f\left(x - \frac{1}{2}h\right) + \delta^5 f\left(x + \frac{1}{2}h\right)}{2} + \dots \right)$$

$$f''(x) = \frac{1}{h^2} \left(\delta^2 f(x) + \frac{1}{12} \delta^4 f(x) - \frac{1}{90} \delta^6 f(x) + \dots \right)$$

Punktów „kontrolnych” funkcji może być oczywiście znacznie więcej niż 5; w naszym przykładzie 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:

interpol.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=5x2+2x
    {0.9, 5.85}, // zwrócić uwagę na sposób inicjalizacji
}
```

```

{1, 7.00}, // tablicy: wpisane są dwie pierwsze
{1.1, 8.25}, // kolumny, a nie wiersze!
{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,
// Poprawność jej konstrukcji nie jest sprawdzana!
{
    POCHODNE res;
    double h=(t[4][0]-t[0][0])/(double) (n-1); // krok '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;
}
void main()
{
    POCHODNE res=stirling(t);
    cout << "f'=" << res.f1 << ", f''=" << res.f2 << endl;
}

```

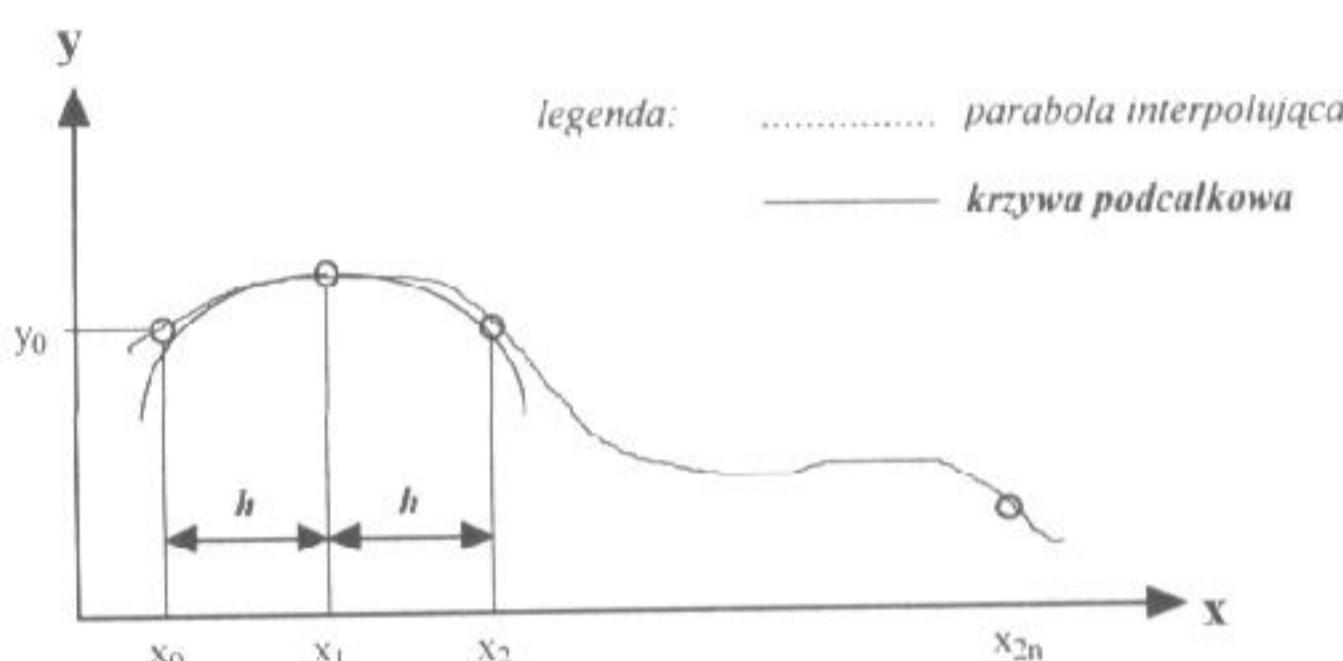
Jeśli już omawiamy różniczkowanie numeryczne, to warto podkreślić związaną z nim dość niską dokładność. Im mniejsza wartość parametru h , tym większy wpływ na wynik mają błędy zaokrągleń, z kolei zwiększenie h jest niezgodne z ideą metody Stirlinga (która ma przecież przybliżać prawdziwe różniczkowanie!). 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.

11.5. 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...), aby otrzymać pożądany rezultat.

Na pomoc przychodzą tu jednak metody interpolacji (czyli przedstawiania skomplikowanej funkcji w prostszej obliczeniowo, przybliżonej postaci. Ideę całkowania numerycznego przedstawia rysunek 11 - 3.

Rys. 11 - I.
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 granice 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; // ilość punktów= 2n+1
// funkcja przykładowa 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 sa 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 całkowania funkcji znanej w postaci *analitycznej*, a nie tylko tabelarycznej:

```
double fun(double x)
{
    // funkcja f(x) jak w przykładzie powyżej
    return x*x-3*x+1;
}
```

```

double simpson_f(double(*f) (double), //wskaźnik do f(x)
                  double a, double b, int N)
// funkcja zwraca całkę znanej w postaci wzoru
// funkcji f(x) w przedziale [a,b],
// N - ilość 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;
}

void main()
{
cout << "Wartość całki =" << simpson(f, -5, 3) << endl;
cout << "Wartość całki =" << simpson_f(fun, -5, 3, 8) << endl;
}

```

11.6. Rozwiązywanie układów równań liniowych metodą Gausса

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 skonstruowanie macierzy 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...).

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. Patrzmy na przykładzie:

- eliminacja x z wierszy 2 i 3 (efekt dodawania wierszy jest pokazany w etapie następnym):

$$\begin{array}{ccc} *(-0,2) & \xrightarrow{\quad} & \begin{array}{l} 5x + 0y + 1z = 9 \\ 1x + 1y - 1z = 6 \\ 2x - 1y + 1z = 0 \end{array} & \xleftarrow{\quad} & *(-0,4) \end{array}$$

- eliminacja y z wierszy 1 i 3 (w pierwszym nie ma już nic do zrobienia):

$$\begin{array}{ccc} 5x + 0y + 1z & = 9 & \\ 0x + 1y - 1,2z & = 4,2 & \xrightarrow{\quad} \\ 0x - 1y + 0,6z & = -3,6 & \xleftarrow{\quad} *1 \end{array}$$

- otrzymujemy ostatecznie macierz *trójkątną*:

$$\begin{array}{ccc} 5x + 0y + 1z & = 9 & \\ 0x + 1y - 1,2z & = 4,2 & \\ -0x + 0y - 0,6z & = 0,6 & \end{array}$$

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. Jedyną „niebezpieczną” operacją metody eliminacji Gaussa 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 rozwiązywanie 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 Gaussa, wraz z danymi przykładowymi:

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]))
                max=j;
        // fabs(x)=|x|, wartość bezwzględna dla danych double
        for(int k=i;k<N+1;k++)
            // zamiana wierszy
        {
            tmp=a[i][k];
            a[i][k]=a[max][k];
            a[max][k]=tmp;
```

```
    }
    if(a[i][i]==0)
        return 0; // Układ sprzeczny!
    for(j=i+1;j<N;j++)
        for(k=N;k>-i;k--)
            //mnożenie wiersza j przez współczynnik "zerujacy":
            a[j][k]=a[j][k]-a[i][k]*a[j][i]/a[i][i];
    }

    // redukcja wsteczna
    for(int j=N-1;j>=0;j--)
    {
        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!
}

void main()
{
    if(!gauss(a,x))
        cout << "Układ jest sprzeczny !\n";
    else
    {
        cout << "Rozwiązańie: \n";
        for(int i=0;i<N;i++)
        cout << "x["<<i<<"]="<<x[i] << endl;
    }
}
```

11.7.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 któryś z nowoczesnych implementacji Fortranu. Język ten, co prawda nie nadaje się do „zwykłego” programowania (tak jak C++ i Pascal), ale wraz z nim są dostarczane zazwyczaj, bardzo bogate biblioteki procedur obliczeniowych (odwracanie macierzy, całkowanie, interpolacja...) – te wszystkie procedury, które programista C++ musi typowo pisać od zera...

Rozdział 12

Czy komputery mogą myśleć?

Zamieszczenie w podręczniku algorytmiki o charakterze ogólnym, rozdziału poświęconego dziedzinie zwanej dość myląco „sztuczną inteligencją”¹, 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. 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 miał on na to zadanie kilkaset stron!

Mój dylemat polegał więc na wyborze kilku interesujących przykładów i na opisaniu ich na tyle prostym językiem, aby nie informatyk 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 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 intelligentnego zachowania* komputera, zbliżającego jego sposób postępowania do ludzkiego. W chwili obecnej potrafimy jedynie kazać komputerowi *naśladować zachowania intelligentne*, gdyż stopień skomplikowania ludzkiego mózgu²

¹ Do tego worka wrzuca się dość sporo bardzo różnych dziedzin: teorię gier, planowanie, systemy ekspertowe, etc.

² Na dodatek, zasada działania ludzkiego mózgu wcale nie jest tak do końca rozumiana przez współczesną naukę.

przewyjsza 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 traktować ten rozdział jedynie, jako zachęcający 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.

12.1. 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:

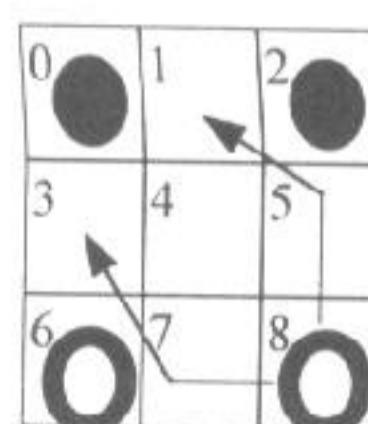
- Jaki język najlepiej się nadaje do naszego zadania?
- Jakich struktur danych należy użyć do reprezentowania szachownicy i pionków?
- Jakich struktur danych należy użyć do reprezentowania toku myślenia gracza?

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 - 1). Możliwe ruchy konika znajdującego się na pozycji o numerze 8 są przedstawione przy pomocy strzałek.

Rys. 12 - 1.
Problem konika
szachowego (I).

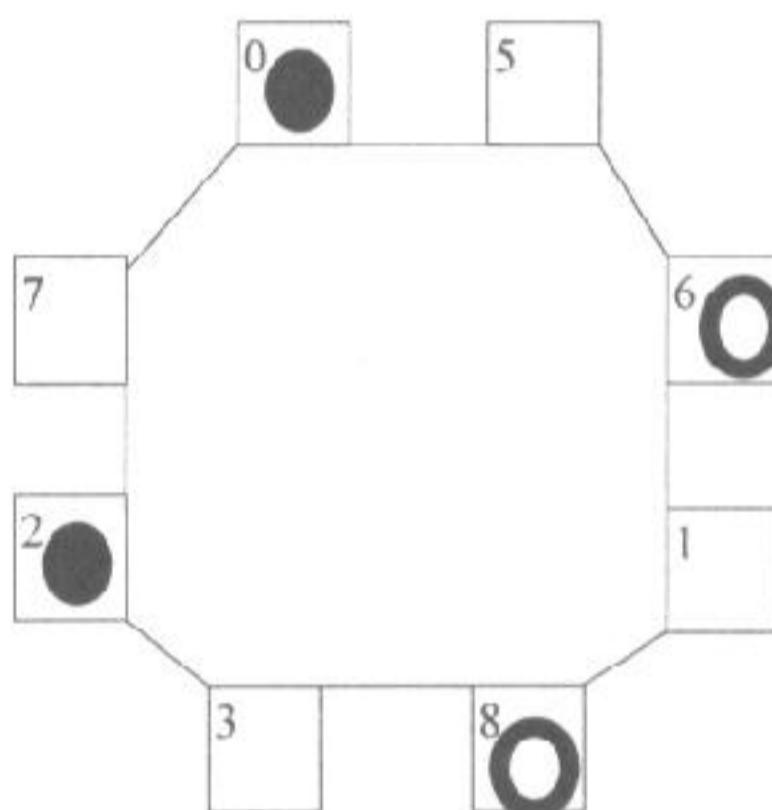


Reprezentacja zadania w tej postaci, w jakiej jest przedstawiona na rysunku, wcale nie ułatwia nam rozwiązania: nie widać klarownie jakie ruchy są dozwolone,

jaki cel należy osiągnąć. Popatrzmy jednak na inne przedstawienie tej samej sytuacji (patrz rysunek 12 - 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.

Rys. 12 - 2.
Problem konika
szachowego (2).



Ćwicz. 12-1

Proszę się zastanowić, jak rozwiązać postawione zadanie, dozwalając być może *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, zważwszy na liczbę możliwych sytuacji i co za tym idzie – rozmiar grafu!

12.2. Gry dwuosobowe i drzewa gier

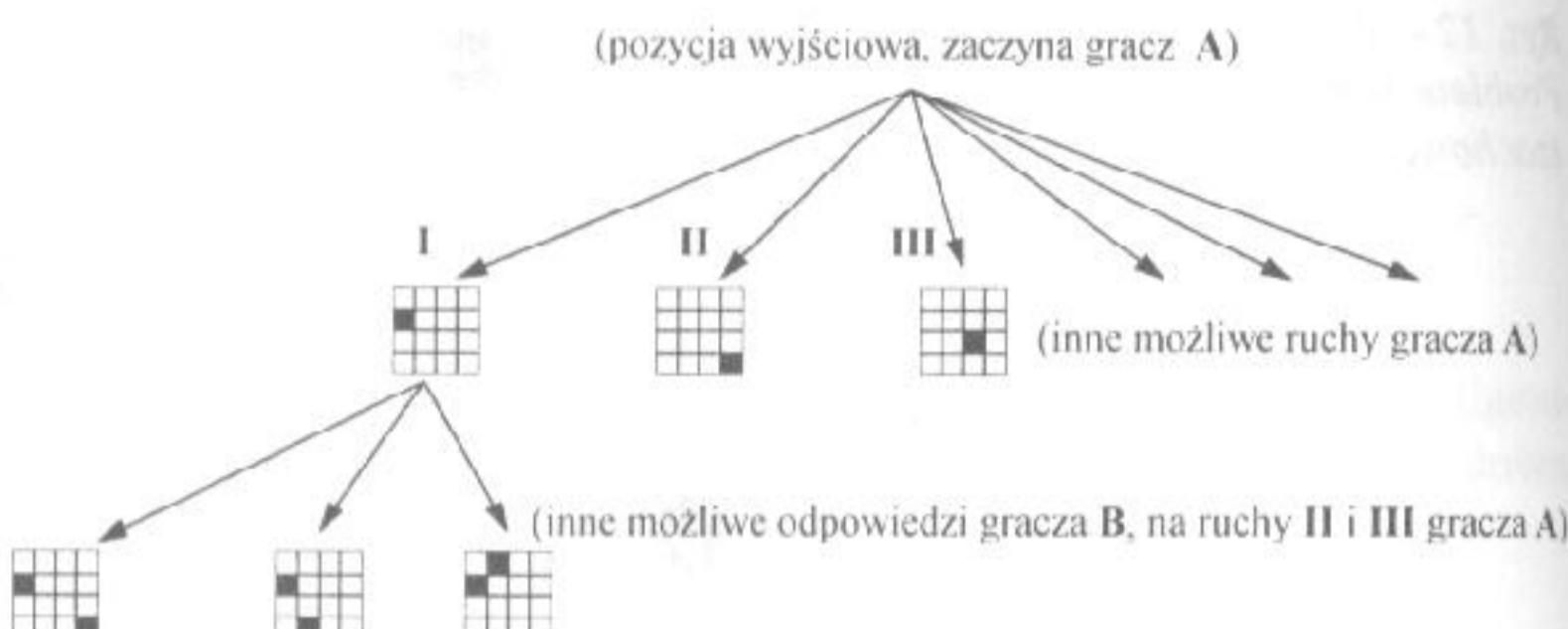
Zaletą typowych gier dwuosobowych jest względna łatwość ich programowej implementacji. Decydują o tym następujące cechy:

- w 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 ułatwiającą reprezentację stanu i przebiegu gry jest *drzewo*. Ruchy kolejnych graczy są przedstawiane przy pomocy węzłów drzewa, w którym poszczególne „piętra” (poziomy zagłębiania) odpowiadają wszystkim możliwym do wykonania ruchom danego gracza. Przykład drzewa gry jest podany na rysunku 12 - 3.

Rys. 12 - 3.
Przykład drzewa pewnej wyimagiowanej gry (2).



Poszczególne węzły są dość skomplikowanymi strukturami danych, pozwalającymi zapisać kompletny stan pola gry (w naszym przykładzie jest to kratownica 4×4 , 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 wcześniej czy później do wygranej, przegranej jednej ze stron lub 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 intelligentnego podejmowania decyzji. W praktyce oznacza to wbudowanie 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 kolejnych (znanych na podstawie wygenerowanego w całości lub częściowo drzewa gry) podejmowana jest decyzja, który ruch wybrać na danym etapie gry.

Pierwsza funkcja jest ideoowo 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 jego „ścieżki”, czyli drogi, po których może się potoczyć dana partia. Niestety, dla większości gier powszechnie uznawanych za godne uwagi rozrywki intelektualne (np. szachy, REVERSI, GO...) 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 odniósła duże sukcesy w poszukiwaniu takich algorytmów, zwanych często zwyczajnie *strategiami przeszukiwania*. Najbardziej znanymi z nich są: *A**, *mini-max*, algorytm cięć α - β , *SSS**. Szczegółowe przedstawienie każdego z tych algorytmów byłoby dość trudne w tej książce, gdzie została przyjęta zasada prezentacji gotowych programów w C++ 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. (Na dołączonej dyskietce 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.

12.3. 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). Analizujemy w ten sposób pewną ilość poziomów³ i analizujemy tylko te ostatnie. Wartości z tych ostatnich poziomów są „wnoszone” do góry, wedle reguł *mini-maxa*.

Prosty przykład prezentowany jest na rysunku 12 - 4, i ilustruje sposób wybrania pierwszego najlepszego ruchu. Wartości liczbowe reprezentują siłę rażenia danej pozycji.

Idea *mini-max* 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 wartością jego wierzchołków potomnych.

Rys. 12 - 4.
Reguła mini-max.



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 mówiąc: zminalizować szanse A na zwycięstwo. Jeśli analiza całego drzewa nie jest praktycznie możliwa, algorytmy poprzestają na pewnej arbitralnej głębokości – na naszym przykładzie jest to $h=2$.

³ Ich ilość jest wybieralna i determinuje głębokość zagłębiania procedury *mini-max*. W przypadku niektórych gier, zbyt głębokie przeszukiwanie nie ma, oczywiście zbytniego sensu: są one zbyt „pływkie”!

Założymy również, że wartości liczbowe węzłów z ostatniego poziomu, zostały nam dostarczone przez pewną znaną funkcję *ewaluacją*. W analizowanym przykładzie został wybrany węzeł z wartością $I=\max(-1, 2, 1)$. 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 nie mających szansy na wyniesienie podczas propagacji wartości wg reguły *mini-max*. Jest ona powszechnie znana jako *algorytm cięć α-β*. Przykładowo, wartość -1 wyniesiona do góry na rysunku 12 - 4 (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 pseudo-kodzie⁴.

Algorytm przeszukiwania drzewa gry, z wykorzystaniem reguły *mini-max*, ma następującą postać⁵:

```

Minimax(węzeł w)
{
    jeśli w jest typu MAX to v=-∞;
    jeśli w jest typu MIN to v=+∞;
    jeśli w jest węzłem terminalnym to
        zwróć ewaluacja(w);
    p1, p2, ... pk = generuj(w); // potomkowie węzła w.
    dla j=1...k wykonuj
    {
        jeśli w jest typu MAX to
            v=max(v, mimimax(pk));
        w przeciwnym wypadku
            v=min(v, (v, mimimax(pk)));
    }
    zwróć v;
}

```

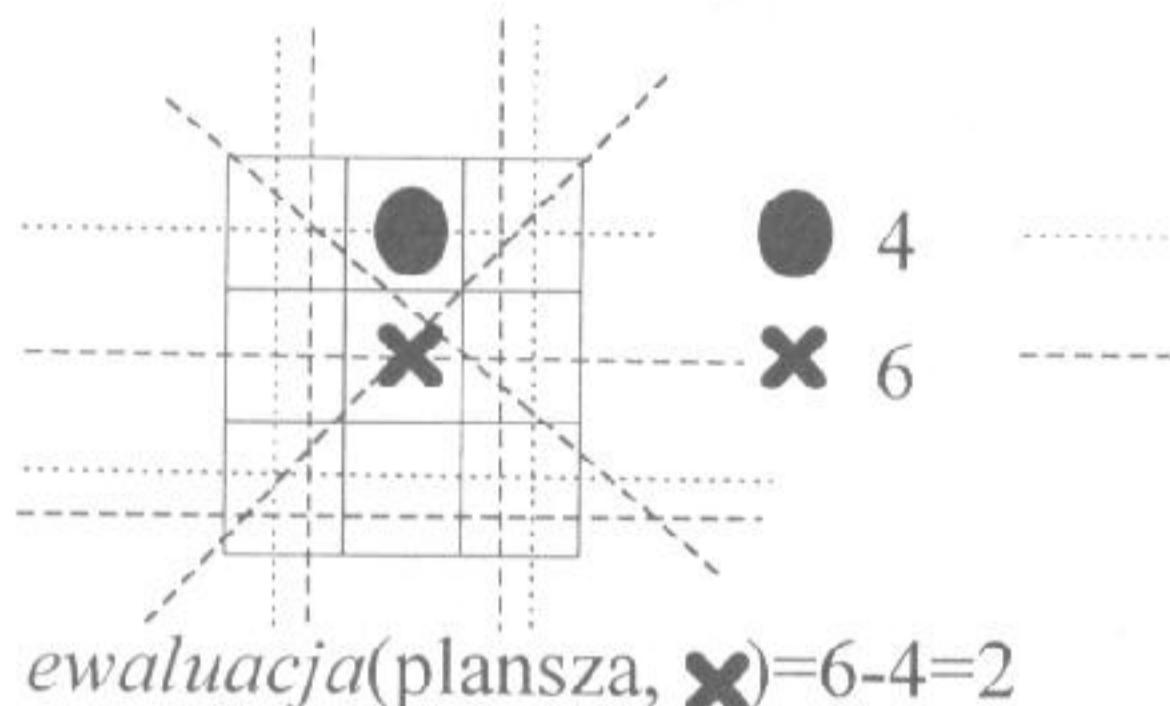
Dotychczas unikaliśmy dyskusji na temat funkcji *ewaluacją*. Powód jest dość prozaiczny: funkcja ta jest silnie związana z rozpatrywaną i nie ma sensu jej omawiać poza jej kontekstem.

⁴ Warto sobie zdać sprawę, że konkretna implementacja procedury *mini-max* może być zmieniona nie do poznania przez grę i sposób jej reprezentacji.

⁵ 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 ilości *linii otwartych* dla danego gracza, tzn. takich, które nie są blokowane przez przeciwnika i w związku z tym rokują nadzieję na skonstruowanie pełnej linii dającej nam zwycięstwo. Omawianą zasadę ilustruje rysunek 12 - 5. Wartość tej liczby jest pomniejszana o ilość linii otwartych dla przeciwnika.

Rys. 12 - 5.
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 - 5. Oprócz wartości typu int możliwe jest pewne wzbogacenie stosowanej semantyki poprzez zastosowanie typu wyliczeniowego⁶:

```
enum KTO{nic, komputer, człowiek};
```

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 można rozwiązać wybierając liczby, które są znacznie większe od tych, zwracanych przez funkcję *ewaluacja*:

```
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)
```

⁶ Stałym *komputer* i *człowiek* odpowiadają na planszy znaki „kółko” i „krzyżyk”.

```
{  
    if (gracz==komputer)  
        return czlowiek;  
    else  
        return komputer;  
}
```

Przydadzą się również funkcje pomocnicze:

```
void WyswietlPlansze(plansza);  
void ZerujPlansze(plansza);  
int KoniecGry(plansza);
```

Ostatnia funkcja 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.

Sama gra jest zwykłą pętlą, która prowokuje wykonanie ruchów. Założymy, że pętla ta została zamknięta w funkcji *Graj*:

```
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 bardziej skomplikowane są gry, takie jak szachy, czy też *Reversi*, mamy bowiem do uwzględnienia efekty uboczne, takie jak bicie pionów, roszady...

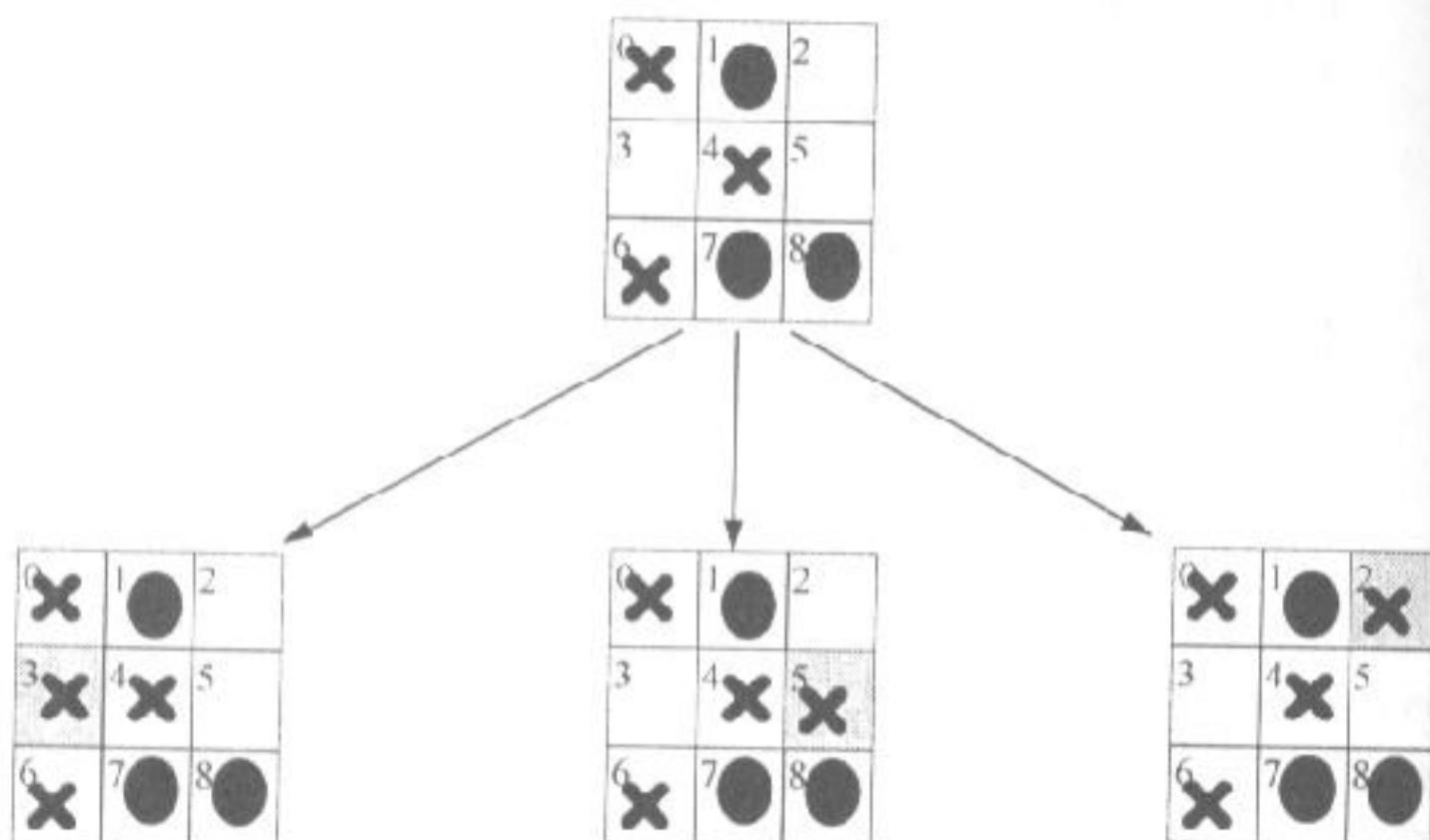
Skąd mamy jednak wiedzieć, jaki ruch powinien zostać wykonany? Odpowiedzi dostarczyć nam powinna funkcja *WybierzRuch*, która używa poznanej wcześniej funkcji *mini-max*:

```

int WybierzRuch(gracz, plansza)
// wybór ruchu zależy od tego, kto gra
if (gracz==czlowiek)
    do{
        cout << "Twój wybór(0..8):";
        cin >> ruch;
    }while(!Zajete(plansza, ruch));
else
{
    cout << "Ruch komputera:\n";
    ruch=Minimax(plansza, gracz);
return ruch;
}

```

Rys. 12 - 6.
Generowanie listy
możliwych ruchów
gracza na podsta-
wie danego węzła.



3 możliwe ruchy dla **X**

Treść procedury *Minimax* jest dokładnym tłumaczeniem algorytmu ze strony 287, oczywiście z uwzględnieniem struktur danych właściwych dla danej gry. Pozorną trudność może sprawić generowanie węzłów-potomków danego węzła. Rysunek 12 - 6 ukazuje wynik funkcji *generuj* dla pewnego węzła *w* (zakładamy, że ruch należał do gracza stawiającego „krzyżyki”). Nasuwają się tutaj na myśl jakieś listy, drzewa, zbiory... Popatrzmy jednak, jak sprytnie można zakodować listę potomków danego węzła, z użyciem tylko jednej pomocniczej planszy (patrz rysunek 12 - 7). Wystarczy się umówić, że wpisanie wartości innej, niż -1 oznacza jeden wygenerowany węzeł: pozwala to nam upakować w jednej planszy całą listę możliwych ruchów!

Rys. 12 - 7.
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

Na tym zakończymy omawianie zagadnień technicznych związanych z programowaniem gier dwuosobowych. Czytelnikowi głębiej zainteresowanemu tą tematyką, polecam jednak pogłębienie swojej wiedzy literaturą specjalistyczną przed przystąpieniem do kodowania np. gry w szachy... 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 postąpili programiści, którzy konstruowali tegoroczną (1996) maszynę mającą pokonać w grze w szachy samego mistrza Kasparowa⁷. We wspomnianym pojedynku góra znowu okazał się człowiek, ale kto wie, co nam przyniesie przyszłość?

⁷ Komputer generował w zadanym czasie, jak największą ilość możliwych strategii, obliczał ich siłę (funkcja *ewaluacja!*) i wybierał tę lokalnie najlepszą.

Rozdział 13

Kodowanie i kompresja danych

W chwili obecnej coraz więcej komputerów jest podłączanych do globalnych sieci komputerowych. Ze względu na relatywnie niskie koszty, w Polsce prym wiedzie Internet, z którego dobrodziejstw korzysta coraz więcej osób, również nie związanych z informatyką. Możliwość „przechadzania się” po sieci, przy pomocy łatwych w użyciu graficznych przeglądarek (np. Netscape, Mosaic) 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 głównym problem stanowiła niska przepustowość łącz, kluczowym zagadnieniem była *kompresja* przesyłanych danych, czyli takie ich zakodowanie¹, które – nie umniejszając ilości przesyłanej informacji – zmniejszy ilość bitów krażących „po kablach”. Obecnie punkt ciężkości przesunął się na bezpieczeństwo danych, tzn. ich ochronę przed niepowołanym dostępem „z zewnątrz”.

Czy kompresja danych jest w ogóle możliwa? 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 $50 \times 8 = 400$ bitów². Czy

¹ 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.

² Dla 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*.

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! Założymy, że dla typowych tekstów ograniczymy się do następującego alfabetu:

‘A’ ... ‘Z’	=	26 znaków
‘ ’(spacja) , ; . -	=	5 znaków
A, Ć, E, Ł, Ñ, Ó, Ž, Ż	=	8 znaków

RAZEM: 39 znaków.

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 bitów³!

Łatwo zauważyć, że znajomość przesyłanego alfabetu pozwala, przy umiejętności dobiorze 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 mini-podręcznik 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*). Nie chciałbym jednak zbyt szeroko omawiać tych zagadnień, gdyż są one związane bardziej z transmisją sygnałów (fizyczna transmisja danych; sens przesyłanej informacji nie jest istotny), niż z informatyką w czystej postaci (aplikacje użytkownika; sens przesyłanej informacji ma kluczowe znaczenie).

W dalszej części rozdziału omówimy szczegółowo popularny system kodowania z tzw. *kluczem publicznym* oraz *kodem Huffmana*, który jest znakomitym i nieskomplikowanym przykładem uniwersalnego algorytmu kompresji danych.

13.1. 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 niemiłych nam celach. Może ono dotyczyć prywatnej korespon-

³ Zyskujemy 25 % pierwotnej długości tekstu!

dencji, jednak w praktyce najczęstsze zastosowanie znajduje we wszelkiego rodzaju transakcjach gospodarczych ze względu na dobro kontrahentów.

Kodowanie pasjonowało ludzi od wieków i czyniono wielkie starania, aby wymyślić takie algorytmy kodujące, które byłyby trudne do złamania w rozsądny czasie. Proces kodowania i dekodowania można przedstawić w postaci prostego schematu, przedstawionego na rysunku 13 - 1.

Rys. 13 - 1.
Algorytmiczny
system
kodujący.



Pewna wiadomość W jest szyfrowana przez nadawcę A przy pomocy 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. Przykładem najprostszego kodu i klucza jest przypisanie literze alfabetu numeru (założymy, 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 rozpoczęnie dekodowanie powinien odjąć od otrzymanych liczb liczbę K , tak aby otrzymać kanoniczny kod tabelkowy⁴. 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 bezpieczeństwo zarówno poufność danych, jak i... samego kuriera!

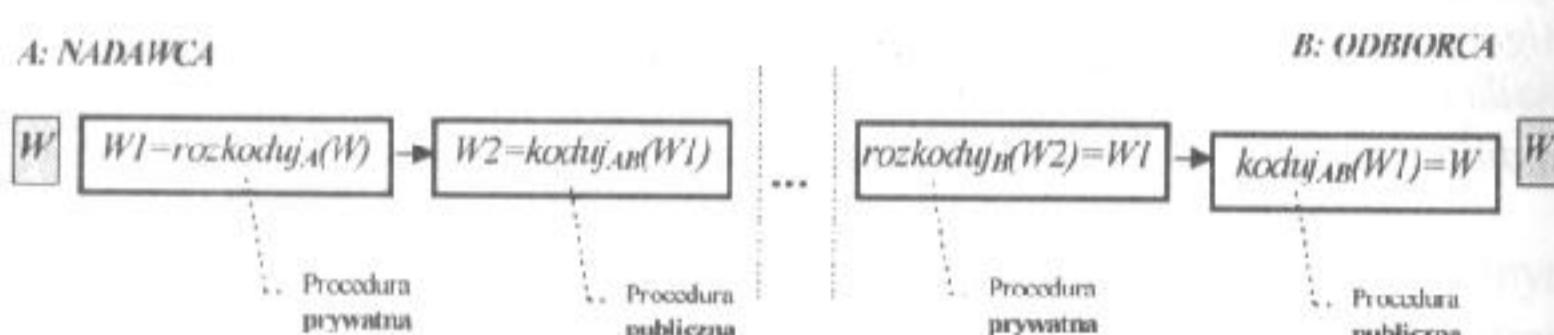
Jak rozwiązać *problem transmisji klucza* w świecie, gdzie ważne jest, aby wiadomość dotarła w ułamku sekundy do odbiorcy, bez obarczania go dodatkową troską o wiarygodność otrzymanego klucza K ? Ponieważ nie znaleziono sensownego rozwiązania tego problemu, z wielką ulgą powitano wynalezienie w 1976 r., metody *kodowania z kluczem publicznym*, która eliminowała całkowicie dystrybucję klucza. Wynalazcami metody byli W. Diffie i M. Hellman, jed-

⁴ Zarówno przykład kodu, jak i klucza są najprostszymi z możliwych i żadna armia na świecie nie zakodowałaby przy ich pomocy nawet jadłospisu dziennego, aby nie osmieszyć się przed przeciwnikiem!

nak jej praktyczna realizacja została opracowana przez R. Rivetsa, 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 złamanie...

Zanim przeanalizujemy system *RSA* na konkretnym przykładzie liczbowym, spróbujmy zrozumieć samą ideę kryptografii z kluczem publicznym.

Rys. 13 - 2.
System kodujący
z kluczem
publicznym.



System kryptograficzny z kluczem publicznym jest przedstawiony na rysunku 13 - 2. Składa się on z trzech procedur: **prywatnych**: *rozkodujA* i *rozkodujB* i **publicznej**: *kodujAB*. 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 *rozkodujA* 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 (*WI*) jest szyfrowana przez powszechnie znaną procedurę szyfrującą *kodujAB* i dopiero w tym momencie wysyłana do *B*.

Odbiorca *B* otrzymuje zakodowaną sekwencję kodową i używa swojej prywatnej funkcji *rozkodujB*, która jest tak skonstruowana, że na wyjściu odtworzy podpisaną wiadomość *WI*. Podobnie specjalna musi być funkcja *kodujAB*, która z cyfrowo podpisanej wiadomości *WI* powinna odtworzyć oryginalny komunikat *W*.



Wymogi bezpieczeństwa zakładają praktyczną *niemożność odtworzenia* tajnych procedur rozkodowujących, na podstawie jawnych procedur kodujących.

Idea jest zatem urzekająca, pod warunkiem wszakże, 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 iloczyn⁵ $N=N1 \times N2$ oraz pewną liczbę P , spełniającą warunek:

$$P \times S \bmod (N1-1) \times (N2-1) = 1.$$

Zostało udowodnione, że dla każdego ciągu kodowego M (tekst zostaje zamieniony na odpowiadający mu ciąg liczbowy o pewnej skończonej długości) spełniona jest równość: $M^P \bmod N = M$.

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 rozsądny czasie.

Wszelkie algorytmy kryptograficzne napotykają na problem wykonywania obliczeń na bardzo dużych liczbach całkowitych. Okazuje się, że obliczenia te mogą zostać znacznie uproszczone, pod warunkiem traktowania 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 podając, np. $x=10000$:

$$(12x^4) + (9876x^3) + (2x^2) + (6x) + 54.$$

⁵ Ponieważ nie są aktualnie znane szybkie metody rozkładu na czynniki pierwsze, dokonanie takiego rozkładu przez osobę postronną jest wysoce nieprawdopodobne.

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_wielomian1(int b, int w[], int rozm)
{
    int res=0, pot=1;
    for(int j=rozmiar-1; j>=0; j--)
    {
        res+=pot*w[j];           // sumy cząstkowe
        pot*=b;                  // następna potęga b
    }
    return res;
}
```

(W przypadku wielomianów o współczynnikach niecałkowitych, należy wszędzie zamienić typ *int* na *double*).

Istnieje jednak tzw. *schemat Hornera*⁶ pozwalający na znacznie prostsze obliczenie $W(b)$:

$$W(b) = \left(\dots \left(\left(a_n b + a_{n-1} \right) b + a_{n-2} \right) b + \dots + a_1 \right) b + a_0.$$

Realizacja schematu Hornera może być następująca:

horner.cpp

```
int oblicz_wielomian2(int a, int w[], int rozm)
{
    int res=w[0];
    for(int j=1; j<rozmiar; res=res*a+w[j++]);
    return res;
}
const n=5;           // stopień wielomianu
void main()
```

⁶ Wynalazcą tej procedury był tak naprawdę Isaac Newton, ale historia przypisała ją Hornerowi.

```

{
int w[n]={1,4,-2,0,7}; // współczynniki wielomianu
cout << oblicz_wielomian1(2,w,n) << endl;
cout << oblicz_wielomian2(2,w,n) << endl;
}

```

Przy użyciu reprezentacji tablicowej, nieskomplikowane staje się również dodawanie i mnożenie wielomianów:

wielom.cpp

```

void dodaj_wiel(int x[],int y[],int z[], int rozm)
{
for(int i=0;i<rozr;i++)
    z[i]=x[i]+y[i];      // wielomian z=x+y
}

void mnoz_wiel(int x[],int y[],int z[], int rozr)
{
int i,j;
for(i=0; i<2*rozr-1; i++)
    z[i]=0;           // zerowanie rezultatu
    // wielomian z=x*y
for(i=0; i<rozr; i++)
    for(j=0; j<rozr; 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:

$$(2x^{1600} + 3x^{900}) \times (3x^{85} + 1).$$

Owszem, można zarczerwować tablice o rozmiarach 1600, 85 i 1600+85 (na wynik), ale biorąc pod uwagę, że będą się one składały głównie z zer, nie jest to najrozsądzniejszym pomysłem...

Na pomoc przychodzi tutaj reprezentacja wielomianu przy pomocy listy jednokierunkowej; wybierzemy 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
    // reszta czynników (jeśli istnieja):
}

```

```

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 funkcji:

```

void main()
{
    WSPOLCZYNNIKI_PTR pw1,pw2,pw3,pwtemp;
    pw1=pw2=pw3=pwtemp=NULL;
    // wielomian pw1=5·x1700+6·x700+10·x50+5:
    pw1=wstaw(pw1,5,1700);
    pw1=wstaw(pw1,6,700);
    pw1=wstaw(pw1,10,50);
    pw1=wstaw(pw1,5,0);
    // wielomian pw2=6·x1800-6·x700+5·x50+15:
    pw2=wstaw(pw2,6,1800);
    pw2=wstaw(pw2,-6,700);
    pw2=wstaw(pw2,5,50);
    pw2=wstaw(pw2,15,0);
    // dodajemy pw1 i pw2:
    pw3=dodaj(pw1,pw2);
    // wielomian pw3=6·x1800+5·x1700+15·x50+20:
}

```

Omawiając system kodowania danych *RSA*, napotkaliśmy na niedogodność związaną z operacjami na bardzo dużych liczbach całkowitych. Abytrzymać ciąg kodowy powstały na podstawie pewnego tekstu M ⁷, musimy obliczyć dość makabryczne wyrażenie:

$$\{\text{ciąg kodowy}\} = M^p \bmod N,$$

⁷ Pamiętajmy, że po zamianie każdej litery tego tekstu na pewną liczbę (np. w kodzie ASCII), całość możemy traktować jako jedną, bardzo dużą liczbę M .

Podnoszenie do potęgi może być zrealizowane poprzez zwykłe mnożenie, ale co zrobić z obliczaniem funkcji *modulo*? Jak sobie, na przykład, poradzić z wyliczeniem:

$$12\ 9876\ 0002\ 6000\ 0000\ 0054 \bmod N?$$

Jeśli wszakże przedstawimy powyższą liczbę jako wielomian o podstawie $x=10^3$, to otrzymamy znacznie prostsze wyrażenie:

$$12(x^4 \bmod N) + 9876(x^3 \bmod N) + 2(x^2 \bmod N) + 6(x \bmod N) + 54.$$

Wartości w nawiasach są stałymi, które można wyliczyć tylko raz i „na sztywno” wpisać do programu kodującego!

13.2. Kompresja danych metodą Huffmana

Kod, który 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 - 1).

Tabela 13 - 1.
Przykład kodowania
znaków pewnego alfa-
betu 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... Teksty zapisywane są na liściach pewnej odpornej na działanie pogody rośliny. W celu szybkiej komunikacji, został wymyślony system szybkiego przesyłania wiadomości przy pomocy sygnałów trąb niosących dźwięk na bardzo długie dystanse.

Dwa krótkie sygnały oznaczają znak  (krótki), dwa długie sygnały oznaczają znak  (długie) itd., zgodnie z przedstawioną wyżej tabelką (0 - sygnał długiego, 1 - krótkiego). Jest godne docenienia, iż mamy przed sobą niewątpliwie kod... binarny! (Nawet, jeśli ów tajemniczy lud nie zdaje sobie z tego sprawy).

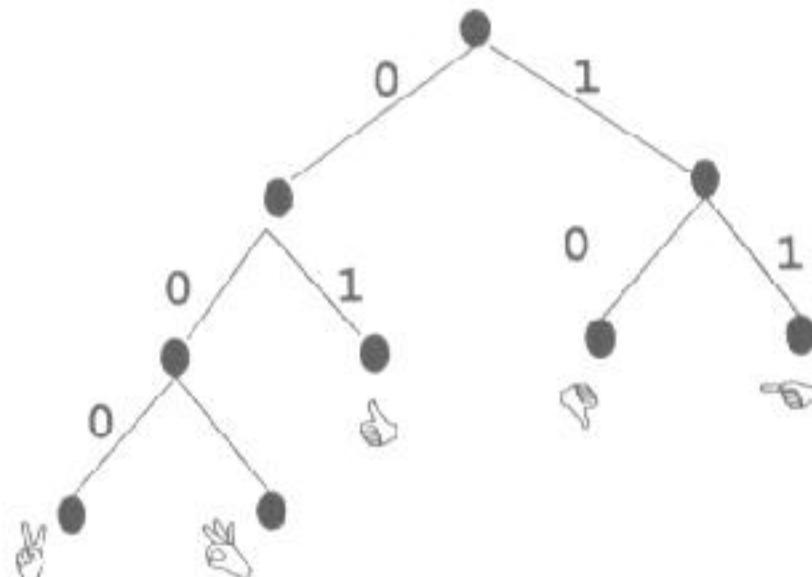
Załóżmy, że pewnego dnia odebrano następujący sygnał: 011110000001 (nadawca wysłał wiadomość:    ), czyli „doślijcie świeże melony”. Czy możliwe jest nieprawidłowe odtworzenie wiadomości, tzn. ewentualne pomyłenie jednego znaku z innym? Spróbujmy:

- 0 – znakiem może być:  lub  lub .
- 01 – już wiemy, że jest to !
- 0+1 – znakiem może być:  lub .
- 0+11 – już wiemy, że jest to !
- 0++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 X , czy też może do znaku Y .

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 - 3.

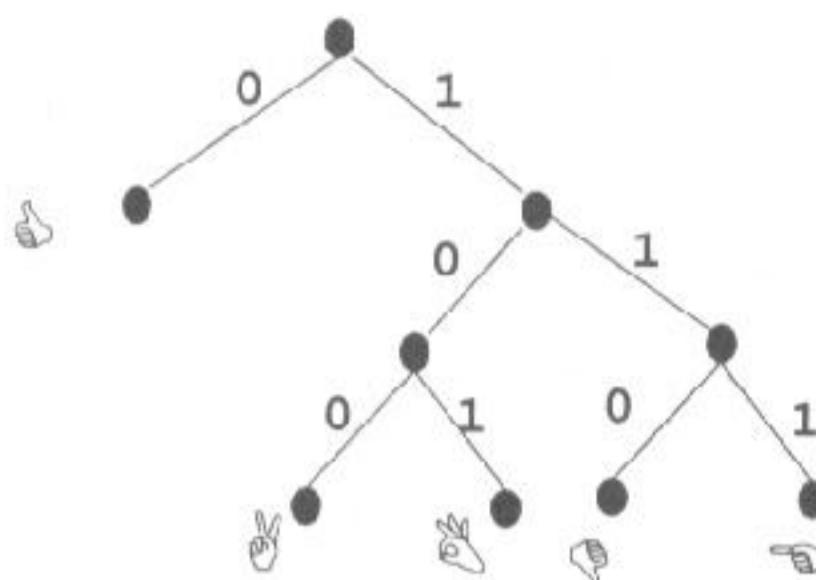
Rys. 13 - 3.
Przykład drzewa kodowego (1).



Przechadzając się po drzewie (poczynając od jego korzenia aż do liści), odwiedzamy gałęzie oznaczone etykietami 0 („lewe”) lub 1 („prawe”). Po dotarciu do danego listka, 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 - 4.

Rys. 13 - 4.

Przykład drzewa kodowego (2).



Powstaje naturalne 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. Zwrócić 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...).

Dla zainteresowanych podaję tabelkę zawierającą dane na temat języka polskiego (przytaczam za [CR90]).

Algorytm Huffmana korzysta w bezpośredni sposób z tabelek takich jak 13 - 2. 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.

Tabela 13 - 2.
Prawdopodobieństwa występowania liter w języku polskim.

Litera	Prawdop.	Litera	Prawdop.	Litera	Prawdop.	Litera	Prawdop.
spacja	0,140	f	0,011	n	0,043	u	0,018
a	0,078	g	0,012	ń	0,001	w	0,037
ą	0,10	h	0,011	o	0,061	y	0,031
b	0,012	i	0,077	ó	0,007	z	0,055
c	0,036	j	0,018	p	0,025	ż	0,008
ć	0,004	k	0,025	r	0,037	ź	0,001
d	0,029	l	0,017	s	0,047		
e	0,064	ł	0,024	ś	0,006		
ę	0,013	m	0,023	t	0,029		

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. Dodać 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.

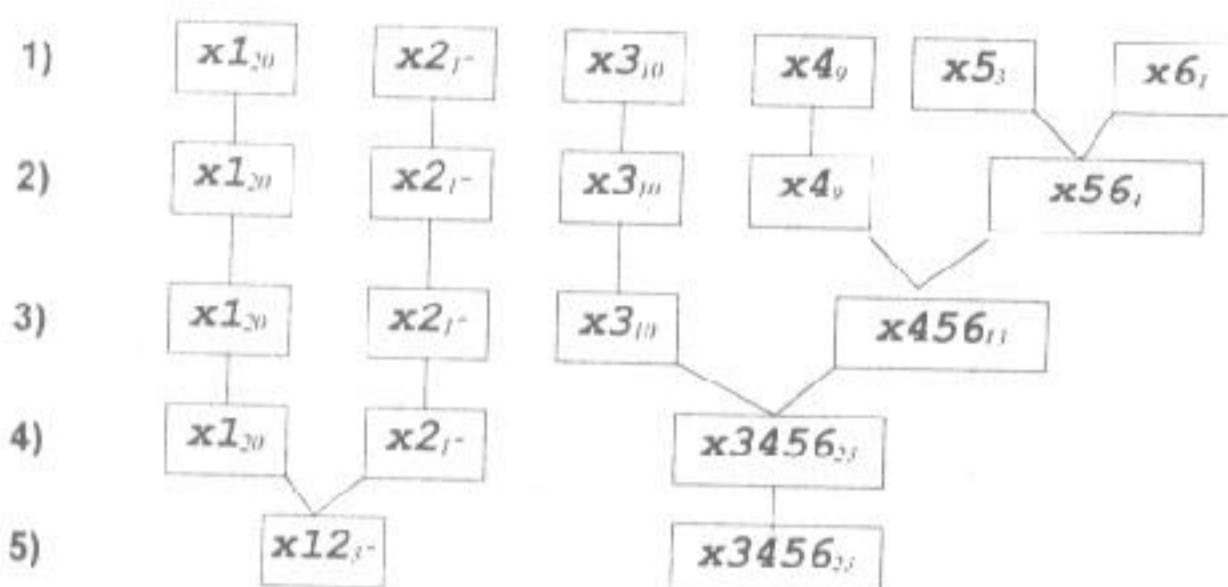
Załóżmy, że dysponujemy alfabetem składającym się z sześciu znaków: x_1, x_2, x_3, x_4, x_5 i x_6 . Otrzymujemy do zakodowania tekst długości 60 znaków, których częstotliwości występowania są następujące: 20, 17, 10, 9, 3 i 1. Aby zakodować sześć różnych znaków, potrzeba minimum 3 bity ($6 < 2^3$), zatem zakodowany

tekst zająłby $3 \times 60 = 180$ 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 - 5).

Rys. 13 - 5.

Konstrukcja kodu Huffmana – faza redukcji.

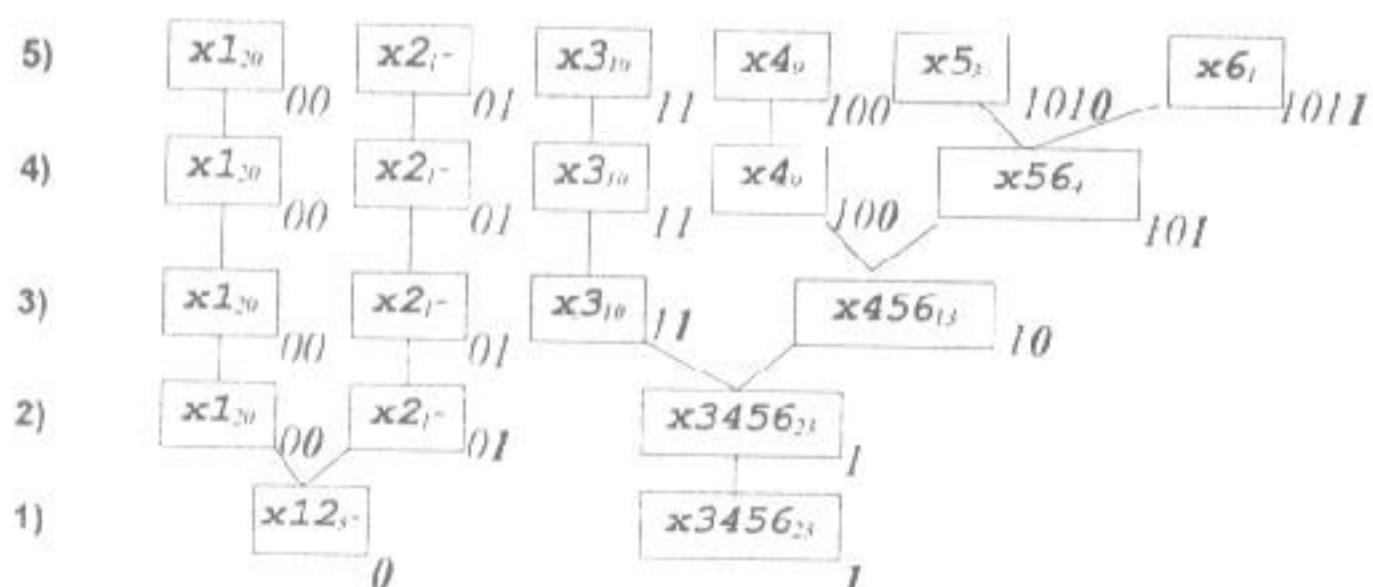


Rysunek przedstawia 6 etapów redukcji kodowanego alfabetu (Proszę nie sugerować się postacią rysunku, to jeszcze nie jest drzewo binarne!). Znaki x_5 i x_6 występują najrzadziej, zatem redukujemy je do zastępczego znaku, który oznaczamy jako x_{56} . Podobnie czynimy w każdym kolejnym etapie, aż dochodzimy do momentu, w którym zostają nam tylko dwa znaki alfabetu (zastępcze). Faza **redukcji** została zakończona i możemy przejść do fazy **konstrukcji kodu**. Patrzmy w tym celu na rysunek 13 - 6.

Rysunek przedstawia 6 etapów redukcji kodowanego alfabetu (Proszę nie sugerować się postacią rysunku, to jeszcze nie jest drzewo binarne!).

Rys. 13 - 6.

Konstrukcja kodu Huffmana – faza tworzenia kodu.



Znaki x_5 i x_6 występują najrzadziej, zatem redukujemy je do zastępczego znaku, który oznaczamy jako x_{56} . Podobnie czynimy w każdym kolejnym etapie, aż dochodzimy do momentu, w którym zostają nam tylko dwa znaki alfabetu (zastępcze). Faza **redukcji** została zakończona i możemy przejść do fazy **konstrukcji kodu**. Patrzmy w tym celu na rysunek 13 - 6.

Bity 0 i 1, które są dokładane na danym etapie do zredukowanych znaków alfabetu, są **wytłuszczone**. Mam nadzieję, że czytelnik nie będzie miał zbytnich kłopotów z odtworzeniem sposobu konstruowania kodu, tym bardziej, że nasz przykład bazuje na krótkim alfabetie.

Przy klasycznej metodzie kodowania binarnego, komunikat o długości 60 (napisany z użyciem 6-znakowego alfabetu) zakodowalibyśmy przy pomocy $60 \times 3 = 180$ bitów. Przy użyciu kodu Huffmana, ten sam komunikat zająłby odpowiednio: $20 \times 2 + 17 \times 2 + 10 \times 2 + 9 \times 3 + 3 \times 4 + 1 \times 4 = 137$ 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... Podczas kodowania nie należy również zapominać, aby wraz z kodowanym komunikatem posłać jego... kod Huffmana! (Inaczej odbiorca miałby pewne problemy z odczytaniem wiadomości).

Problemów technicznych 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! Popatrzymy 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 alfabetie
{
    wstaw wszystkie znaki ciągu s do sterty H
    stosownie do ich częstotliwości;
    dopóki H nie jest pusta wykonuj
    {
        jeśli H zawiera tylko jeden znak X wówczas
            X staje się korzeniem drzewa Huffmana T;
        w przeciwnym wypadku
        {
            ...
        }
    }
}
```

```
    • 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 często-
      tliwość 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!

Uwaga:

Na dyskietce dołączonej do książki, w katalogu HUFFMAN znajdują się programy HUF.C i UNHUF.C autorstwa Shaun Case. Są to programy typu *public domain*,ściągnięte przez *ftp* z sieci Internet. Autor prezentuje gotowe procedury kodujące i dekodujące pliki binarne. 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). Oczywiście, nie mogę ręczyć, że działają one poprawnie, ale taka już jest idea oprogramowania *public domain*...

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 rozwiązywanie 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.

14.1. Teksty zadań

Zad. 14-1

Tzw. sito Erastotenesa jest jedną ze starszych metod na otrzymywanie 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 wielokrotności liczby 2:

1, *, 3, *, 5, *, 7, *, 9, *, 11, *, 13, *, 15... N

- usunąć z nich wielokrotności liczby 3:

1, *, *, *, 5, *, 7, *, *, *, 11, *, 13, *, *... N

- usunąć z nich wielokrotności liczby 5, 7...

Algorytm ten można nieco uprościć, wiedząc że jeśli liczba n *nie jest* pierwsza, wówczas jest ona podzielna przez pewną liczbę pierwszą, taką że jest ona mniejsza lub równa całkowitej części \sqrt{n} (oznaczane dalej jako $\text{sqrt_int}(n)$).

Proszę napisać program, który:

- sprawdza metodą *brute-force*}, czy dana liczba jest liczbą pierwszą;
- wykorzystując metodę sita Erastotenesa, liczy wszystkie liczby pierwsze mniejsze od 100;
- wykorzystując metodę uproszczoną, liczy wszystkie liczby pierwsze mniejsze od 100.

Zad. 14-2

Napisać funkcję, która otrzymując na wejściu datę zakodowaną w postaci liczby całkowitej (np. 220744) wypisze słownie jej znaczenie (tutaj: „22 lipca 1944”).

Zad. 14-3

W operacjach macierzowych często są używane tablice z dużą ilością 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, nie zaprezentowane 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 zapełnienia tablicy zerami taka struktura danych jest opłacalna, jeśli chodzi o zużycie pamięci.

Zad. 14-4

Zaproponuj dwie wersje rekurencyjnego algorytmu obliczania funkcji x^n (rekurencja „naturalna” i rekurencja „z parametrem dodatkowym”).

Zad. 14-5

Spróbuj stworzyć *nierekurencyjną* funkcję, która na podstawie dwóch list posortowanych zwróci jako wynik listę posortowaną, zawierającą wszystkie ich elementy. Wymóg: nie wolno tworzyć nowych komórek pamięci, jedyne, co jest dozwolone, to manipulacja wskaźnikami.

Zad. 14-6

Napisz funkcję, która na podstawie ceny podanej w postaci liczby całkowitej typu, np. *long* wydrukuje ją w postaci słownej. Przykład: wywołanie *cena_słownie(12304)* powinno dać w rezultacie tekst: „dwanaście tysięcy trzysta cztery zł”.

Zad. 14-7

Napisz program, który realizuje permutację cykliczną danej tablicy wejściowej o zadaną liczbę pozycji. Spróbuj przeprowadzić dokładną analizę problemu, wybierając technikę programowania: rozwiążanie iteracyjne, rekurencyjne – jeśli tak, to jakiego typu?

Zad. 14-7

Napisać program liczący w najprostszy sposób ilość wystąpień danego słowa w tekście wejściowym.

Zad. 14-9

Napisać program obliczający w najprostszy możliwy sposób dane „statystyczne” tekstu wejściowego: ilość wystąpień każdej litery, słowa etc.

Zad. 14-10

Napisać 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).

Zad. 14-11

Napisać program sprawdzający, czy zdanie wejściowe zawiera „ukryte słowo”, np. tekst „Bronek alergicznie nie znosił makaronu z kaszą” ukrywa słowo „bramka”.

Zad. 14-12

Napisać 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).

14.2. Rozwiązania

Zad. 14-1

Do rozwiązania zadania (a) będziemy potrzebowali funkcji zwracającej nam pierwszą całkowitą liczbę x spełniającą warunek $x^2 < n$:

```
inline int sqrt_int(int n)
{ return (int)sqrt((double)n)/1; }
```

erastot.cpp

(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:

```
int pierwsza(int n)
{
    // czy n jest liczba pierwszą?
    int limes=sqrt_int(n);
    for(int i=2; n!=(n/i)*i && i<=limes; i++)
        if (i>limes)
            return 1; // tak, liczba pierwsza
        else
            return 0; // nie, "zwyczajna" 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 *tp=new int[n+1];
    for(int i=1;i<=n;i++) // zaznaczenie wszystkich
        tp[i]=i; // liczb naturalnych od 1 do n
    int cpt=1;
    while(cpt<n)
    {
        //szukamy pierwszego niezerowego elementu tablicy tp:
        for(cpt++;(tp[cpt]==0) && (cpt!=100); cpt++);
        // zerujemy wielokrotności tego elementu (cpt) w tp
        int k=2;
        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
}

```

Metoda będąca przedmiotem pytania (c) jest skomplikowana w zapisie, ale sama jej idea odpowiada dokładnie tej, zapowiedzianej w tekście zadania. Na samym początku możemy zauważyć, że ilość potencjalnie odnalezionych liczb pierwszych $< n$ na pewno jest mniejsza od \sqrt{n} . Ułatwi nam to optymalny przydział pamięci. Aby sprawdzić, czy pewna liczba n jest liczbą pierwszą, wystarczy zatem podzielić ją przez uprzednio wyliczone wartości:

```

int* liczby_pierwsze(int n)
{
    // zwraca tablice liczb pierwszych < n
    // ilość liczb pierwszych < n na pewno nie przekracza
    //  $\sqrt{n}$ , stad optymalny przydział pamięci:
    int limes_i,*lp=new int[sqrt_int(n)];
    lp[0]=1;lp[1]=2; //inicjacja tablicy liczb pierwszych
    for(int i=2;i<n;i++)
        lp[i]=0;
    int np=2;          // następna liczba pierwsza (tzn. jej
                       // indeks w tablicy
    for(i=3;i<n;i++)
    {
        limes_i=sqrt_int(i);
        for(int j=1; (i!=lp[j]* (i/lp[j])) &&
                         (lp[j]<=limes_i); j++);
        if (lp[j]>limes_i)
            lp[np++]=i;
    }
    return lp;
}

```

Zad. 14-3

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 przy pomocy równości: $k \cdot p = 2N^2$. Przykładowo dla $p=8$, $N=10$, dwudziesty szósty niezerowy element już przebiera miarkę. Praktycznie rzecz ujmując, typowa ilość niezerowych elementów powinna być znacznie mniejsza od $\frac{2N^2}{p}$ – programista musi sam podjąć decyzję, co do właściwej interpretacji wyrażenia „znacznie”...

Zad. 14-4

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));
}

void main()
{
    cout << "Dwa do potęgi trzeciej: \n";
    cout << "Metoda 1\t" << pot1(2, 3) << "\n";
    cout << "Metoda 2\t" << pot2(2, 3) << "\n";
}

```

Zad. 14-10

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;
// słowo jest (na razie) palindromem :
enum {TRUE, FALSE} test=TRUE;
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";
}
```

Zad. 14-12

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 wzorzec. Tym bardziej, że dla pewnych konfiguracji danych wyrażenie jest obliczane źle!

wyraz.cpp

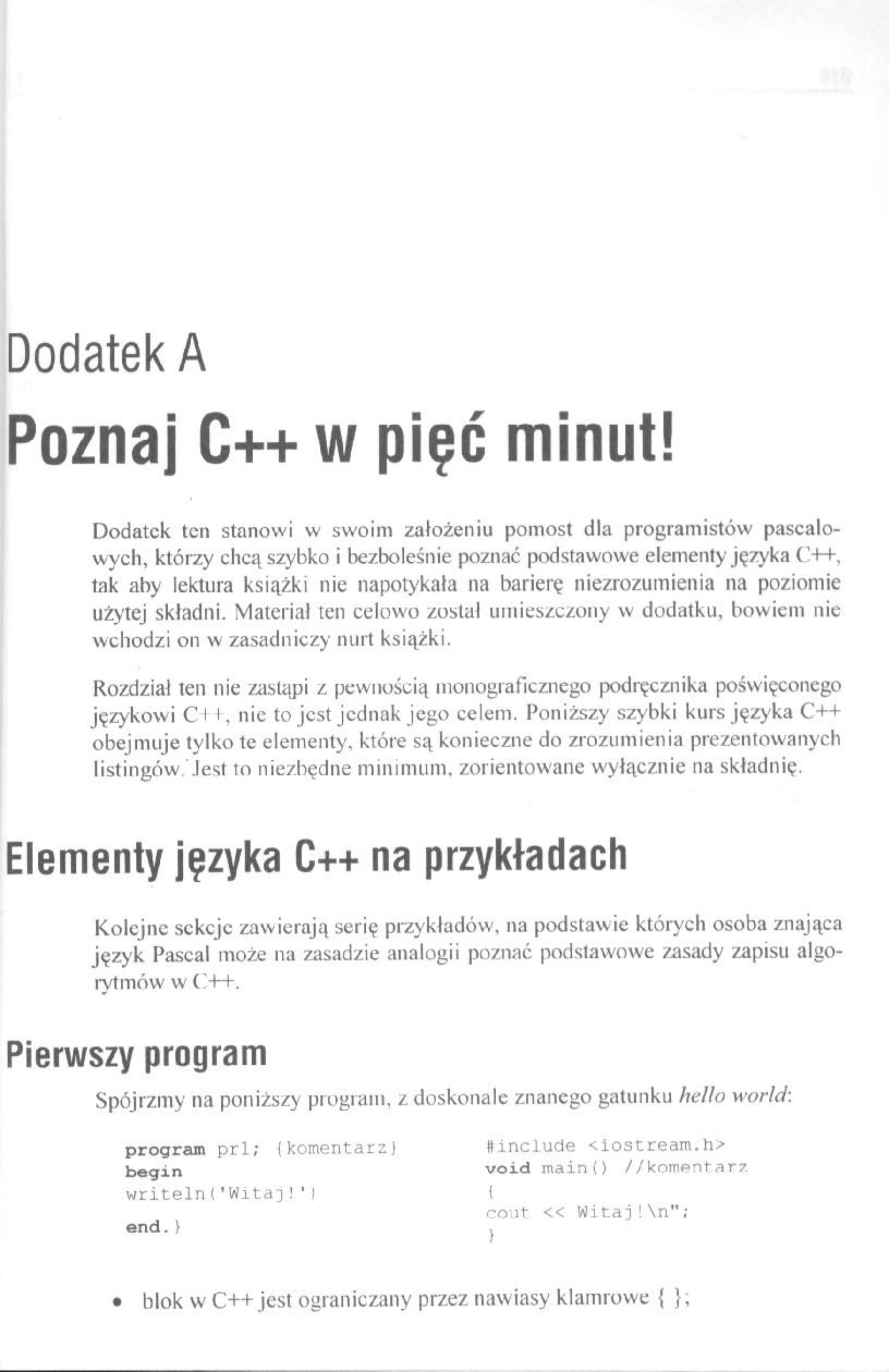
```
double transl(char *s)
// zamienia ciągi znaków typu 1+1 na 2, 1+1+2*5 na 12 etc.
// uwaga funkcja nie analizuje operacji dzielenia i
// badania przypadku dzielenia przez zero!
{
    int n=strlen(s);
    char *s1=new char[n+1]; //kopia robocza tekstu wejściowego
    strcpy(s1,s);

    // szukamy znaków + i *
    for(int i=0;i<n;i++)
        if(s[i]=='+'||s[i]=='*')
        {
            s1[i]='\0'; // "ucinamy" kopię roboczą tekstu z prawej strony
            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"
}
```

```
void 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 stanowi w swoim założeniu pomoż dla programistów pascalowych, którzy chcą szybko i bezboleśnie poznać podstawowe elementy języka C++, tak aby lektura książki nie napotykała na barierę 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.

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 prl; {komentarz}
begin
writeln('Witaj!')
end.}                                #include <iostream.h>
                                         void main() //komentarz
                                         {
                                         cout << Witaj!\n";
                                         }
```

- blok w C++ jest ograniczany przez nawiasy klamrowe { };

- słowo *void* oznacza procedurę, czyli funkcję nie zwracającą wartości w sensie arytmetycznym;
- działanie programu rozpoczyna się od funkcji o nazwie *main*.

W C++ komentarz `// „działa”`, aż do końca linii. Chcąc coś napisać w komentarzu pomiędzy instrukcjami, użyj raczej `/* komentarz */` niż `//komentarz`.

Linia `#include <iostream.h>` oznacza dołączenie¹ pliku *iostream.h* do pliku z programem. Plik ten jest obowiązkowy, jeśli zamierzamy używać standardowych 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.

Uwaga: W dalszych przykładach dyrektywa ta będzie omijana.

Pliki z rozszerzeniem *h* (lub *hxx*) zawierają zazwyczaj deklarację często używanych stałych i typów. Oczywiście, rozszerzenie pliku nie ma dla kompilatora najmniejszego znaczenia, warto się jednak trzymać jakiejś określonej konwencji.

Tekst w C++ można wypisać, wysyłając ciąg znaków ograniczony przez cudzysłów ("tekst") do standardowego wyjścia. Sekwencja `\x` oznacza znak specjalny, np. `\n` jest to skok do nowej linii podczas wypisywania tekstu na ekranie, `\t` - znak tabulacji etc.

Operacje arytmetyczne

Niewielkie różnice dotyczą pewnych operatorów, które w Pascalu nazywają się nieco inaczej niż w C++. To, co może uderzyć nas przy pierwszym spojrzeniu na język C++, to nasycenie programów skrótami w zapisie operacji arytmetycznych, czyniące listingi dość często pozornie mało czytelnymi. Mam tu przede wszystkim na myśli operatory `++`, `--` oraz całą rodzinę wyrażeń typu:

zmienna OPERATOR = wyrażenie

Należy podkreślić, iż stosowanie tych form nie jest obowiązkowe, tym niemniej 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;
```

```
const float pi=3.14;
// lub double
int a,b,c;
void main()
{
  a=1;
```

¹ Jeszcze przed właściwą komplikacją.

```

a:=a+1; {inkrementacja}
b:=b-2
end.
b=1;
a++; //inkrementacja
b-=2; // ŚREDNIK!
}

```

- miejsce deklarowania zmiennych w C++ jest dowolne. Można to uczyć się *przed, za i w ciele* niektórych instrukcji;
- przy deklaracji stałej, opuszczenie typu w deklaracji oznacza, że będzie to domyślnie *int*;
- przypisanie wartości zmiennej odbywa się za pomocą `=`, a nie `:=`;
- znanym z Pascal'a *div* i *mod*, odpowiadają w C++ odpowiednio `/` i `%`.

Zwróćmy uwagę na często używane w C++ operatory inkrementacji/dekrementacji (`++/-`). Zastosowane w wyrażeniu mają one priorytet², jeśli są użyte przedrostkowo, natomiast w przypadku użycia przyrostkowego priorytet ma wyrażenie.

Przykład:

```

a=2;
b=5;
n=a+b++; // n=7 (priorytet ma dodawanie)
b=5;
k=a+++b; // k=8 (priorytet ma inkrementacja)

```

- zmienna θ wyrażenie jest równoważne klasycznemu zapisowi: *zmienna=zmienna θ wyrażenie*, gdzie θ oznacza pewien operator dwuargumentowy.

Operacje logiczne

Podobnie jak arytmetyczne, operacje logiczne także mają swoje osobliwości. Na szczęście nie jest ich aż tak wiele. Programiści pascalowi 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!`

```

Program pr3;
var a:boolean;
begin
  a:=true;

```

```

int a;
void main()
{
  a=-5;
}

```

² Tzn. są uwzględniane w pierwszej kolejności.

```

if a=true then
writeln('true')
else
writeln('false')
end.

if (a==0)
cout << "true \n";
else
cout << "false\n";
}

```

W C++ typ *boolean* nie istnieje: „symuluje” się go na ogół za pomocą *int*, przy czym *zero* oznacza *false*, a wszystkie inne wartości – *true*.

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 (patrz tabela A-1).

Tabela A - I.
Porównanie operatorów
Pascalu i C++.

Pascal	C++
-	==
not	!
<>	!=
OR	
AND	&&

Zmienne dynamiczne

C++ stanowi ciekawy melanż mechanizmów o wysokim poziomie abstrakcji (jest to przecież tzw. język strukturalny) z możliwościami zbliżającymi go do języka asemblera. Umiejętnie 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.

void main()
{
float *p;
// albo: double *p
p=new float;
*p=3.14;
delete p;
}

```

- W C++ operacje wskaźnikowe (na adresach) nie są ograniczone do zmiennych dynamicznych.

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;
begin
  t[3]:=11;
  t[4]:=t[3]+1;
end.
```

```
typedef int tab[3];
tab t;
void main()
{
  t[0]=11;
  *(t+1)=t[0]+1;
}
```

- Język C++ nie zapewnia kontroli przekroczenia granic tablic podczas dostępu do nich przy pomocy indeksowania, ufając niejako programiście. Radą na to jest zastosowanie mechanizmów obiektowych, ale w wersji pierwotnej trzeba po prostu uważać, aby nie znaleźć się „w malinach”;
- 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 cell=
  record
    c:char;
    a,b,d:integer;
  end;
```

```
struct cell
{
  char c;
  int a,b,d;
};
```

³ Tutaj mogłyby to oznaczać „złe adresy”...

```

end;
var x:cell;
begin
  x.c:= 'a';
  x.a:=1
end.

```

```

cell x;
void main()
{
  x.c= 'a';
  x.a=1;
}

```

- rekordy w C++ są zwane *strukturami*, dostęp do nich jest podobny jak w przypadku Pascalu;
- nie można wprost zadeklarować rekordu z wariantami;
- jest możliwe, podobnie jak w Pascalu, „włożenie” tablicy do rekordu i odwrotnie;
- pole *nazwa_pola* rekordu dynamicznego, wskazywanego przez zmienną *x* nie jest dostępne poprzez *x.nazwa_pola*, lecz przez *x->nazwa_pola*.

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* (ang. *przerwij*). Ominięcie go, spowodowałoby wykonanie instrukcji znajdujących się dalej, aż do napotkania jakiegoś innego *break* lub końca instrukcji *switch*.

```

program pr7;
var w:integer;
begin
  w:=2;
  case w of
    1: writeln('1');
    2: writeln('2');
    otherwise:
      writeln('?');
  end
end.

```

```

int w;
void 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

```

```

int i,j;
void main()

```

```

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.
}

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++ dopóty, dopóki wyrażenie *v* jest różne od *zero*⁴.
- 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.)

```

Podprogramy

W języku C++, podobnie zresztą jak i w klasycznym C, wszystkie podprogramy są nazywane *funkcjami*. Odpowiednikiem znanej z Pascala *procedury* jest specjalna funkcja „zwieracząca” typ o nazwie *void*.

Procedury

„Oto przykład użycia procedury w Pascalu i C++:

<pre> program pr9; procedure procl(a,b:integer; var m:integer) {zmienna lokalna:} </pre>	<pre> void procl(int a, int b, int& m.) { //zmienna lokalna: </pre>
---	--

⁴ Porównaj np. z *repeat... until*.

```

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.

```

```

int c;
c=a+b;
cout << c << endl;
m=c*a*b;
}

int i,j,k;
void main()
{
  i=10;
  j=20;
  procl(i,j,k);
}

```

- 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,...)*;
- nie jest możliwe przekazanie przez referencję tablicy. 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*. Podeczas wywołania funkcji *Fun*, tablica *tab* jest przekazywana poprzez swój adres i jej zawartość może być fizycznie zmodyfikowana.

Funkcje

Zasadnicze różnice pomiędzy funkcjami w C++ i w Pascalu dotyczą sposobu zwracania wartości:

```

program pr10;
function
plus2(a:integer):integer;
begin
  plus2:=a+2
end;
var i:integer;
begin
  i:=10;
  writeln(plus2(i))
end.

```

```

int plus2(int a)
{
  return a+2;
}

int i;
void main()
{
  i=10;
  cout<<plus2(i)<<endl;
}

```

- w C++ instrukcja `return(v)` powoduje natychmiastowy powrót z funkcji z wartością `v`. Przykładowo, po instrukcji `if(v) return(val)` nie trzeba używać `else5` – 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, co do typów parametrów wejściowych. 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]) {...}
```

to tuż za dyrektywami `#include` możemy dopisać linię:

```
void f(int, char*[]); // tu średnik!
```

(Zwróć uwagę na tradycyjny średnik!). Celowo piszę *możemy*, bowiem użycie nagłówków jest wymuszone tylko i wyłącznie zdrowym rozsądkiem programisty. Pozwala ono już na etapie wstępnych komplikacji uniknąć wielu błędów związanych z wywołaniem funkcji ze złymi parametrami. Notabene niektóre kompilatory z założenia nie tolerują ich braku.

Struktury rekurencyjne

Przykład następny pokazuje sposób deklarowania rekurencyjnych struktur danych.

```
Program prll;
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;
}TYP_X,TYP_WSK_DO_X;

void main()
{
    TYP_WSK_DO_X p;
    p=new TYP_X;
    cin >> p->wartosc;
    p->nastepny=NULL;
}
```

- Odpowiednikiem `nil` w C++ jest `NULL`.

⁵ Ale, oczywiście nie jest to zabronione.

- Adres dowolnej zmiennej w C++ może być z niej „pobrany” poprzez poprzedzenie jej nazwy operatorem &.
- W C++ nie ma pozaskładniowych ograniczeń, co do operacji na adresach, zmiennych wskaźnikowych, dynamicznych przydziałach pamięci etc.
- Wartość zmiennej, na którą wskazuje pewna zmienna wskaźnikowa *wsk*, może być z niej „wyłuskana” poprzez poprzedzenie jej nazwy operatorem * (gwiazdka).

Przykład:

```
int k=12;  
int *wsk=&k;  
cout << *wsk << endl; // program wypisze 12
```

Programowanie obiektowe w C++

Cała siła i piękno języka C++ zawiera się nie w cechach odziedziczonych od swojego przodka⁶, 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 się stało z C++. Niegdyjsza 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!

Dla formalności tylko, przypomnę jeszcze „ostrzeżenie” zawarte we wstępie: cały ten rozdział służy wyłącznie nauczeniu programisty pascalowego *czytania i rozumienia* listingów napisanych w C++. Ograniczona objętość książki, w konfrontacji z rozpiętością tematyki, nie pozwala na omówienie wszystkiego. Tym niemniej, cytowane tu przykłady zostały wybrane ze względu na ich dużą reprezentatywność. Osoby głębiej zainteresowane programowaniem obiektowym w C++ mogą skorzystać, np. z [Poh89], [WF92] lub [Wró94] w celu poszerzenia swojej wiedzy.

Terminologia

Typowe pojęcia związane z programowaniem obiektowym poglądownie zgrupowano na rysunku A - 1.

⁶ Którym jest oczywiście język C!

Rys. A - 1.
Terminologia
w programowaniu
obiektowym.



- Zmienna tego nowego typu danych zwana jest *obiektem*;
- *Metody* są to zwykłe funkcje lub procedury operujące polami, stanowiące jednak własność klasy⁷.

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 konstruktora 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 zajętą 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 ona 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

⁷ Tzn. mogą z nich korzystać obiekty danej klasy – inne, „zewnętrzne” funkcje programu już nie!

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:                                // początek sekcji publicznej
    Complex(double x,double y) // konstruktor klasy
    {
        Re=x;
        Im=y;
    }
    void wypisz(); // nagłówek funkcji która wypisuje
                    // liczbę urojoną
    double Czesc_Rzecz(); // zwraca część rzeczywista
    {
        return Re;
    }
    double Czesc_Uroj (); // zwraca część urojoną
    {
        return Im;
    }
    // nagłówek funkcji, która przeddefiniowuje operator
    // + (plus) aby umożliwić dodawanie licz zespolonych:
    friend Complex& operator +(Complex,Complex);
    // nagłówek funkcji, która przeddefiniowuje operator
    // << aby umożliwić wypisywanie licz 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 dodawania⁸ (mnożenia, dzielenia etc.) oraz wypisywać ją w pewnej określonej postaci⁹:

- 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 „wewnętrz” deklaracji klasy ograniczonej nawiasami klamrowymi { }. Decyzja o miejscu definicji jest najczęściej podyktowana długością kodu: jeśli metoda ma pokaźną objętość¹⁰, 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, popatrzymy, jak zrealizować brakujące metody.

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 poprzez 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.

⁸ Przykład ogranicza się tylko do dodawania – pozostałe operacje arytmetyczne Czytelnik może z łatwością dopisać samodzielnie.

⁹ Reprezentacja za pomocą modułu i fazy zostaje pozostawiona do realizacji Czytelnikowi jako proste ćwiczenie programistyczne.

¹⁰ Powszechnie zalecaną regułą jest nieprzekraczanie jednej strony przy konstrukcji procedury – tak aby całość mogła zostać objęta wzrokiem bez konieczności gorączkowego przerzucania kartek.

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 standar-dowych*, 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 przedefiniowania operatora dwuargumentowego jest użycie do tego celu tzw. *funkcji zaprzyjaź-nionej*; jest to specjalna funkcja, która nie będąc metodą¹¹ 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: $(a+j*b)+(c+j*d) = (a+c)+(b+d)*j$. Jest to matematyka elementarna, zawarta w programie nauczania szkoły średniej. Po utworzeniu, nowy obiekt jest zwracany przez referencję – czyli jako w pełni adresowalny obiekt, który może być przypisany innemu obiekowi, na którego rzecz 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* przy pomocy *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).

¹¹ W konsekwencji nie mogą być wywoływanie za pomocą notacji „z kropką”!

Podobnie jak w przypadku operatora +, celowe mogłoby być przedefiniowanie operatora <<, który wysyła sformatowane dane do strumienia wyjściowego. W C++ służy do tego celu klasa o nazwie *ostream*. Bez wnikania w szczegóły¹², proponuję zapamiętać zastosowaną poniżej sztuczkę:

```
ostream& operator << (ostream &str, Complex x)
{
    str << x.Czesc_Rzecz() << "+j*" << x.Czesc_Uroj();
    return str;
}
```

Spójrzmy wreszcie na program przykładowy, który tworzy obiekty i manipuluje nimi:

```
#include "complex.h"
void 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 unikalne 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. *pola statyczne*.

Poprzedzenie w definicji klasy C, atrybutu x słowem *static*, spowoduje utworzenie właśnie tego typu zmiennej. Inicjacja takiego pola może nastąpić nawet przed

¹² Nie miejsce tu bowiem na omawianie dość złożonej hierarchii bibliotek klas dostarczanych z dobrymi kompilatorami C++. Początkującemu fana C++ taki opis mógłby dość skutecznie zanudzić...

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. Oczywistym 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 *stałej* (np. `void fun() const;`). Nazwa ta jest dość nieszczęśliwie wybrana, chodzi w istocie o metodę, która deklaruje się, że nigdy nie zmodyfikuje pól obiektu, na rzecz którego została zaktywowana.

Dziedziczenie własności

Założmy, że dysponujemy starannie opracowanymi klasami *A* i *B*. Dostaliśmy je w postaci *skompilowanych bibliotek*, tzn. oprócz kodu wykonywalnego mamy tylko do dyspozycji 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 śmieci? Powinno być oczywiste dla każdego, że nie zadawałbym tego pytania, gdyby odpowiedź nie brzmiała: *NIE*. Język C++ pozwala na bardzo łatwą „reutylizację” 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* (ang. *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...). Oznacza 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 *prywatne*: klasa dziedzicząca traci możliwość używania ich w swoich metodach przy pomocy nazwy. Ten brak dostępu można, oczywiście sprytnie ominąć, definiując wyspecjalizowane metody służące do *kontrolowanego* dostępu do pól klasy.

Dodateknie 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;
}
};

void main()
{
C3 ob(10);
ob.pisz_wszystko();
}

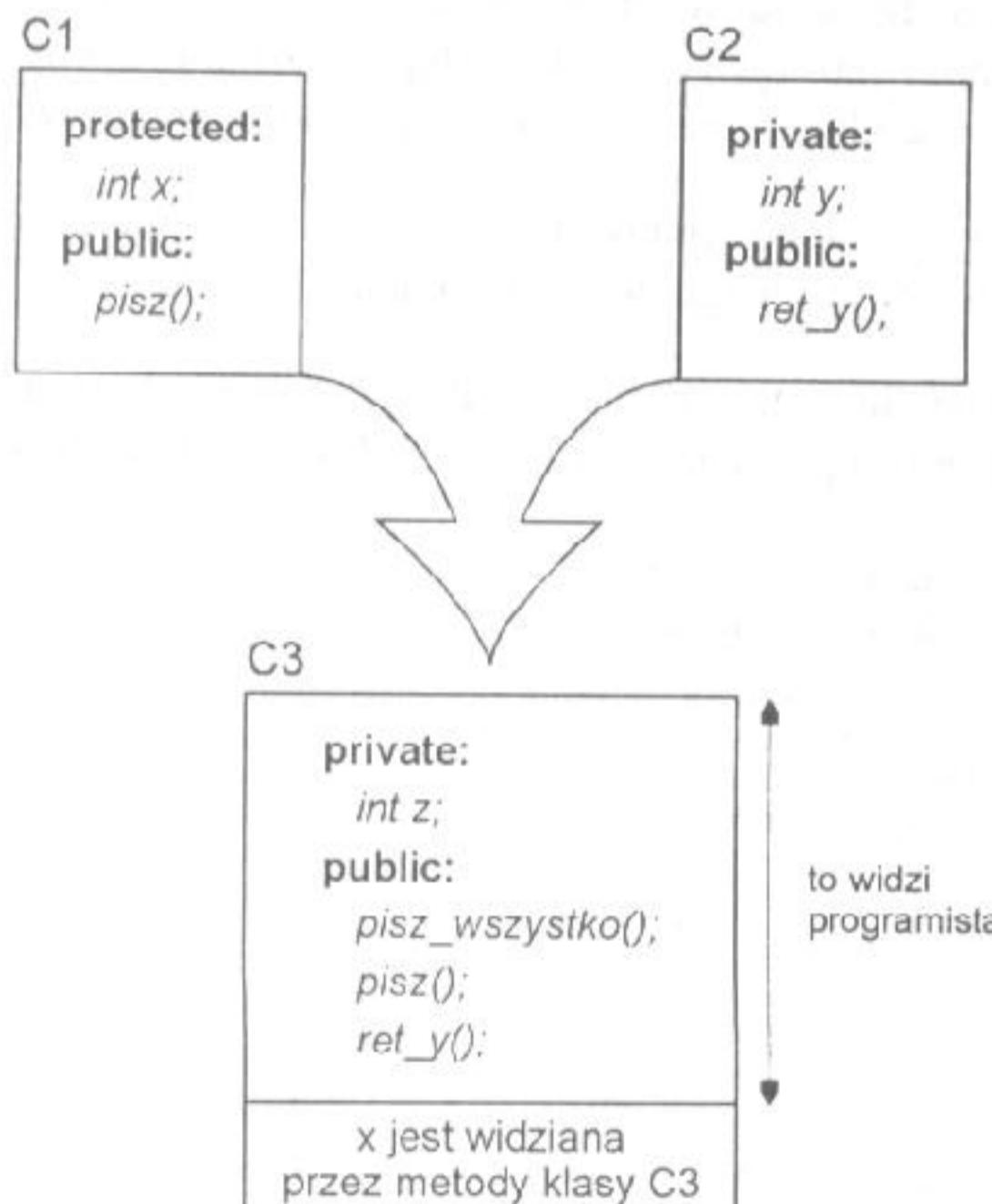
// wynik:
// Wszystkie pola:
// x=11
// y=9
// z=20

```

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.

Rys. A - 2.
Dziedziczenie
własności.



W C++ kilka różnych pod względem zawartości funkcji może nosić taką samą nazwę¹³ – „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. Założmy, 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 *C*, ale do innego celu. Uzupełniamy wówczas klasę *C3* o następującą definicję¹⁴:

```
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()*.

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. Tym niemniej, zaprezentowana powyżej wiedza jest już wystarczająca do tworzenia dość skomplikowanych aplikacji zorientowanych obiektywnie. Inne mechanizmy, takie jak np. bardzo ważne funkcje *wirtualne* i tzw. *klasy abstrakcyjne*, trzeba już pozostawić wyspecjalizowanym publikacjom – zachęcam Czytelnika do lektury.

¹³ Taka cecha jest zwana przeciążaniem.

¹⁴ Ponadto należy dodać linię *void pisz();* do sekcji publicznej klasy *C3*.

Literatura

- [AHU87] A. V. Aho, J. E. Hopcroft, i J. D. Ullmann. *Structures de données et algorithmes*. Addison-Wesley Europe/InterEditions, Paris, 1987. (Tłumaczenie z języka angielskiego).
- [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.
- [CP84] A. Couvert i R. Pendrano. *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.
- [DF89] E. Dijkstra i W. H. Feijen. *A Method of Programming*. Addison-Wesley Publishing Company, 1989.
- [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, Septembre 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.
- [Kla87] Praca zbiorowa pod redakcją Jerzego Klamki. *Laboratorium metod numerycznych*. Skrypt nr 1305 Politechniki Śląskiej w Gliwicach, Gliwice 1987.
- [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.
- [Kro89] D. Krob. Algorithmique et structures de données. Ellipses, 1989.
- [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. *Algorithms in C++*. Addison-Wesley Publishing Company, 1992.
- [Str78] B. Stroustrup. *Le langage C++ 2ème édition*. Addison-Wesley Publishing Company, 1978. (Tłumaczenie z języka angielskiego).
- [WF92] K. Weiskamp i B. Flaming. *The Complete C++ Primer – 2nd ed.* Academic Press, Inc., 1992.
- [Wró94] P. Wróblewski. *Język C++ dla programistów*. Helion, 1994

Spis ilustracji

Rys. 1-1	Etapy konstrukcji programu	22
Rys. 2-1	„Sprzątanie klocków”, czyli rekurencja w praktyce	30
Rys. 2-2	Drzewo wywołań funkcji silnia(3)	34
Rys. 2-3	Obliczanie fib(4)	35
Rys. 2-4	Ilość wywołań funkcji MacCarthy'ego w zależności od parametru wywołania	37
Rys. 2-5	Nieskończony ciąg wywołań rekurencyjnych	39
Rys. 2-6	Spirala narysowana rekurencyjnie	43
Rys. 2-7	Spirala narysowana rekurencyjnie – szkic rozwiązania	43
Rys. 2-8	Kwadraty „parzyste” ($n=2$)	44
Rys. 2-9	Przeszukiwanie binarne na przykładzie	48
Rys. 2-10	Trójkąty narysowane rekurencyjnie	49
Rys. 3-1	Zerowanie tablicy	62
Rys. 4-1	Sortowanie przez wstawianie (1)	82
Rys. 4-2	Sortowanie przez wstawianie (2)	83
Rys. 4-3	Sortowanie przez wstawianie (3)	83
Rys. 4-4	Sortowanie „bąbelkowe”	85
Rys. 4-5	Podział tablicy w metodzie Quicksort	87
Rys. 4-6	Zasada działania procedury Quicksort	87
Rys. 4-7	Budowa niezmiennika dla algorytmu Quicksort	88
Rys. 4-8	Sortowanie metodą Quicksort na przykładzie	90
Rys. 5-1	Typy rekordów używanych podczas programowania list	94
Rys. 5-2	Przykład listy jednokierunkowej (1)	95
Rys. 5-3	Przykład listy jednokierunkowej (2)	96
Rys. 5-4	Dołączanie elementu na jej początek	99
Rys. 5-5	Dołączanie elementu listy z sortowaniem	99
Rys. 5-6	Wstawianie nowego elementu do listy – analiza przypadków	100

Rys. 5-7	Fuzja list na przykładzie	105
Rys. 5-8	Sortowanie listy bez przemieszczania jej elementów (1)	109
Rys. 5-9	Sortowanie listy bez przemieszczania jej elementów (2)	110
Rys. 5-10	Tablicowa implementacja listy	123
Rys. 5-11	Metoda „tablic równoległych” (1)	125
Rys. 5-12	Metoda „tablic równoległych” (2)	126
Rys. 5-13	Lista dwukierunkowa	127
Rys. 5-14	Usuwanie danych	128
Rys. 5-15	Lista cykliczna	128
Rys. 5-16	Stos i podstawowe operacje na nim	129
Rys. 5-17	Tablicowa realizacja kolejki FIFO	134
Rys. 5-18	Sterta i jej tablicowa implementacja	137
Rys. 5-19	Konstrukcja sterty na przykładzie	138
Rys. 5-20	Poprawne wstawianie nowego elementu do sterty	139
Rys. 5-21	Ilustracja procedury NaDol	142
Rys. 5-22	Drzewa binarne i wyrażenia arytmetyczne	144
Rys. 5-23	Tablicowa reprezentacja drzewa	146
Rys. 5-24	Tworzenie drzewa binarnego wyrażenia arytmetycznego	148
Rys. 5-25	Kompresja danych zaletą Uniwersalnej Struktury Słownikowej	154
Rys. 5-26	Reprezentacja słów w USS	155
Rys. 6-1	Wieże Hanoi – prezentacja problemu	170
Rys. 6-2	Wieże Hanoi – sposób rozwiązywania	171
Rys. 7-1	Użycie list do obsługi konfliktów dostępu	198
Rys. 7-2	Podział tablicy do obsługi konfliktów dostępu	199
Rys. 7-3	Obsługa konfliktów dostępu przez próbkowanie liniowe	201
Rys. 7-4	Utrudnione poszukiwanie danych przy próbkowaniu liniowym	202
Rys. 8-1	Algorytm typu brute-force przeszukiwania tekstu	208
Rys. 8-2	Fałszywe starty” podczas poszukiwania	209
Rys. 8-3	Wyszukiwanie optymalnego przesunięcia w algorytmie K-M-P	212
Rys. 8-4	„Przesuwanie się” wzorca w algorytmie K-M-P (1)	212
Rys. 8-5	„Przesuwanie się” wzorca w algorytmie K-M-P (2)	213
Rys. 8-6	Optymalne przesunięcia wzorca „ananas”	214
Rys. 8-7	Przeszukiwanie tekstu metodą Boyera i Moore'a	216
Rys. 9-1	Mnożenie macierzy	229
Rys. 9-2	Obliczanie wartości ciągu liczb Fibonacciego	241
Rys. 9-3	„Dwuwymiarowy” wzór rekurencyjny	242
Rys. 10-1	Przykład grafu	246
Rys. 10-2	Normalizowanie” grafu(1)	247

Rys. 10-3 „Normalizowanie” grafu(2)	247
Rys. 10-4 Tablicowa reprezentacja grafu	248
Rys. 10-5 Reprezentacja grafu przy pomocy słownika węzłów	249
Rys. 10-6 Przykładowe wykonanie algorytmu Roy-Warshalla	252
Rys. 10-7 Poszukiwanie drogi w grafie	254
Rys. 10-8 Algorytm Floyda (1)	255
Rys. 10-9 Algorytm Floyda (2)	256
Rys. 10-10 Przeszukiwanie grafu „w głąb”	258
Rys. 10-11 Przeszukiwanie grafu „wszerz”	259
Rys. 10-12 Zawartość kolejki podczas przeszukiwania grafu „wszerz”	260
Rys. 10-13 Problem doboru	262
Rys. 10-14 Listy rankingowe w problemie doboru	262
Rys. 11-1 Algorytm Newtona odszukiwania miejsc zerowych	268
Rys. 11-2 Interpolacja funkcji $f(x)$ przy pomocy wielomianu $F(x)$	271
Rys. 12-1 Problem konika szachowego (1)	282
Rys. 12-2 Problem konika szachowego (2)	283
Rys. 12-3 Przykład drzewa pewnej wyimaginowanej gry (2)	284
Rys. 12-4 Reguła mini-max	286
Rys. 12-5 Pojęcie linii otwartych w grze w „kółko i krzyżyk”	288
Rys. 12-6 Generowanie listy możliwych ruchów gracza na podstawie danego węzła	290
Rys. 12-7 Kodowanie listy węzłów potomnych przy użyciu tylko jednego węzła	290
Rys. 13-1 Algorytmiczny system kodujący	295
Rys. 13-2 System kodujący z kluczem publicznym	296
Rys. 13-3 Przykład drzewa kodowego (1)	303
Rys. 13-4 Przykład drzewa kodowego (2)	304
Rys. 13-5 Konstrukcja kodu Huffmana – faza redukcji	306
Rys. 13-6 Konstrukcja kodu Huffmana – faza tworzenia kodu	306
Rys. A-1 Terminologia w programowaniu obiektowym	327
Rys. A-2 Dziedziczenie własności	334

Spis tabelic

Tabela 2 - 1. Objasnenia instrukcji graficznych.....	44
Tabela 3 - 1. Czasy wykonania programow dla algorytmow róznej klasy.....	56
Tabela 3 - 2. Złożoność teoretyczna algorytmów – przykłady.....	59
Tabela 3 - 3. Czasy wykonania programów dla algorytmów róznej klasy.....	73
Tabela 5 - 1. Wady i zalety list jednokierunkowych.....	108
Tabela 5 - 2. Typy węzłów w drzewie opisującym wyrażenie arytmetyczne.....	145
Tabela 7 - 1. Kodowanie liter przy pomocy 5 bitów.....	194
Tabela 9 - 1. Przykładowe rozwiązania problemu plecakowego.....	236
Tabela 10 - 1. Problem doboru na przykładzie.....	264
Tabela 13 - 1. Przykład kodowania znaków pewnego alfabetu 5-znakowego.....	302
Tabela 13 - 2. Prawdopodobieństwa występowania liter w języku polskim.....	305
Tabela A - 1. Porównanie operatorów Pascalu i C++.....	320

Skorowidz

A

Abu Ja'far Mohammed ibn Mūsā al-Khowārizmī, 17
Adleman, L., 296
Aiken, H., 20
algorytm, 9
 A*, 285
 cechy, 18
 definicja, 17
 Euklidesa, 17
 Floyd, 254
 język asemblera, 23
 język prezentacji, 10
 opis słowny, 23
 poprawność, 25
 poziom abstrakcji opisu, 23
 Roy-Warshalla, 251
 sposób prezentacji, 23
 SSS*, 285
algorytm cięć α-β, 285
algorytm Newtona, 268
algorytm Roy-Warshalla, 252
algorytmik
 rozwój, 19
algorytmika, 9; 19
algorytmu
 kryteria wyboru, 53
algorytmy numeryczne, 267
algorytmy sortowania, 81
 kryteria wyboru, 90
arytmetyka dużych liczb, 297
ASCII, 302
assembly output, 215

B

Babbage, Ch., 19
Boyer, R. S., 210
breadth first, Patrz strategia "wszerz"

C

C++, 10; 24; 95
calloc, 95
całkowanie funkcji, 274
ciąg Fibonacciego, 35; 240
ciąg Fibonaciego, 77
Cook, S. A., 210
czas wykonania algorytmu, 58
czas wykonania programu, 54

D

debugger, 26
dekompozycja problemu, 42; 228
delete, 103
depth first, Patrz strategia „w głąb”
dereksywacja, 165
Diffie, W., 295
Dijkstra, E., 22; 26
drzewa, 143
drzewa binarne, 144
drzewo gry, 284
drzewo kodowe, 303

E

Eckert, J. P., 20
eliminacja zmiennych lokalnych, 177

end-recursion, *Patrz* wywołanie terminalne
Euler, L., 245

F

Floyd, R., 22
Forth, 149
funkcja
 modulo, 219
 nagłówek, 41
 parametry domyślne, 41
 wywołanie przez wartość, 39
 zaprzyjaźniona, 105
funkcja Ackermanna, 75
funkcja H, 193; 219
 suma modulo, 195
funkcja MacCarthy'ego, 36
funkcja *modulo*, 302
funkcja *O*, 59
funkcja odwrotna, 178
fuzja list, 105

G

Garmisch, 21
glouton, *Patrz* algorytmy żarłoczne
głowa, 94
GO, 285
Gödel, K., 20
Gosper, R. W., 210
gra w „kółko i krzyżyk”, 285
graf, 245
 diagonalny, 250
 nieśkierowany, 247
 operacje matematyczne, 249
 przechodni, 250
 reprezentacja, 248
 skierowany, 246
 węzel, 246
grafu
 domknięcie przechodnie, 250
grafy
 przeszukiwanie, 257
grafy stanów, 283
greedy, *Patrz* algorytmy żarłoczne
gry dwuosobowe, 283

H

Hellman, M., 295
heurystyka, 257
Hoare, C. A. R., 22
Hollerith, H., 19

I

IBM, 20
IFIP, 21
indukcja matematyczna, 26
interfejs użytkownika, 23
interpolacja funkcji, 271

J

Jacquard, J. M., 19
język programowania, 10
języki prezentacji programów, 24

K

Karp, R. M., 210
klasa
 destruktor, 103
klasa algorytmu, 73
klasa *O*, 60
Knuth, D. E., 210
kod
 nadmiarowy, 294
 nierównomierny, 294
 równomierny, 294
kod ASCII, 302
kod Huffmana, 294; 304
kodowanie, 194; 293
kodowanie danych, 293
kodowanie z *kluczem publicznym*, 294; 295
Koenigsberg, *Patrz* Euler, L.
kolejka priorytetowa, 136
Kolejki FIFO, 133
 tablicowa realizacja, 134
kompilacja, 9
kompilator, 13; 23; 166; 204
 Borland C++, 13; 215
 GNU C++, 13; 215
 kod wykonywalny, 23
kompresja danych, 154; 293
komputer
 BULL Gamma3, 20
 EDVAC, 20
 ENIAC, 20
 IBM 604, 20
 IBM 650, 20
 MARK I, 20
 UNIVAC I, 20

konferencja NATO, 21
konwencje typograficzne, 14
kryptosystem RSA, 296

L

- LISP, 24; 106
 lista cykliczna, 128
 lista dwukierunkowa, 127
 lista jednokierunkowa, 94
 reprezentacja tablicowa, 122
 struktura informacyjna, 95
 wady i zalety, 108

M

- macierz kierowania ruchem, 253
 malloc, 95
 Markow, A. A., 20
 maszyna analityczna, *Patrz* Babbage
 Mauchly, J. W., 20
 McCarthy, J., 21
 metoda „tablic równoległych”, 125
 metoda eliminacji Gaussa, *Patrz* rozwiązywanie układów równań
 metoda Floyda, 26
 metoda funkcji przeciwnych, 178
 metoda niezmienników, *Patrz* metoda Floyda
 metody programowania, *Patrz* techniki programowania
 miara złożoności obliczeniowej, 55
 mini-max, 285
 mnożenie macierzy, 229
 Moore, J. S., 210
 Morris, J. H., 210
 myślenie rekurencyjne, 42

N

- najbardziej czasochłonna operacja, 57; 64
następny, 94
 Neumann, Johannes von, 20
new, 95
 niezmiennik, 26
 NULL, 95
 NWD, 17

O

- obliczanie wartości funkcji, 269
 odpluskwanie, 21
 Odwrotna Notacja Polska, 148
ogon, 94
O-notacja, 61
 ONP, *Patrz* Odwrotna Notacja Polska

P

- Pascal, 10; 25
 podwójne kluczowanie, 203
 Postscript, 149
 poszukiwanie miejsc zerowych funkcji, *Patrz*
 algorytm Newtona
 Pratt, V. R., 210
private, 105
 problem doboru, 261
 problem plecakowy, 235
 program, 9
 etapy konstrukcji, 22
 warunki końcowe, 27
 warunki wstępne, 27
 wersja na dyskietce, 13
 PROLOG, 24; 106
protected, 105
 próbkowanie liniowe, 201
 przeciążanie funkcji, 116
 przeddefiniowanie operatora, 104
 przeszukiwanie, 189
 przeszukiwanie binarne, 48; 72; 190; 234
 przeszukiwanie grafów, 257
 strategia "w głąb", 257
 strategia "wszerz", 259
 przeszukiwanie liniowe, 189
 przeszukiwanie tekstów, 207
 algorytm Boyera i Moore'a, 216
 algorytm K-M-P, 210; 211
 algorytm Rabina i Karpa, 218
 brute-force, 208
 przypadek najgorszy, 66
 przypadek najlepszy, 66
 przypadek średni, 67
 przypadek typowy, *Patrz* przypadek średni

Q

- Quicksort, *Patrz* sortowanie szybkie, *Patrz*
 sortowanie szybkie

R

- Rabin, M. O., 210
 RC, *Patrz* równanie charakterystyczne
 rekurencja
 zajętość pamięci, 36
 rekurencja, 29; 223
 „naturalna”, 40
 „z parametrem dodatkowym”, 40
 drzewo wywołań, 34
 ilustracja, 29; 31
 niedogodności, 34

nieskończona ilości wywołań, 38
poprawność definicji, 39
poziom, 33; 34; 41
rozkład na problemy elementarne, 30
sposób wykonywania programu, 33
typy programów, 40
uwagi praktyczne, 45
zakończenie algorytmu, 30
rekurencja skrośna, 46
rekursja, *Patrz* rekurencja
REVERSI, 285
Rivets, R., 296
RO, *Patrz* rozwiązywanie ogólne
routing, *Patrz* macierz kierowania ruchem
rozkład „logarytmiczny”, 72
rozmiar danych wejściowych, 55
rozwiązywanie ogólne, 70
rozwiązywanie równania rekurencyjnego, 70
rozwiązywanie szczegółowe, 70
rozwiązywanie układów równań, 276
równanie charakterystyczne, 69
różniczkowanie funkcji, 272
RS, *Patrz* rozwiązywanie szczegółowe
RSA, *Patrz* kryptosystem RSA
ruchy dozwolone, 283

S

schemat Hornera, 269; 298
schematy derekursywacji
 typu *if... else*, 182
 typu *while*, 181
 z podwójnym wywołaniem rekurencyjnym, 185
shaker-sort, *Patrz* sortowanie przez wytrząsanie
Shamir, A., 296
silnia, 33; 40; 57; 173
sito Erazostenesa, 309
słownik węzłów, *Patrz* graf, reprezentacja
sortowanie
 bąbelkowe, 84
 przez wstawianie, 82
 przez wytrząsanie, 86
 szynkowe, 87
sortowanie danych, 82
sortowanie wewnętrzne, 81
sortowanie zewnętrzne, 81
SRL, *Patrz* szereg rekurencyjny liniowy
sterta, 136
stos, 128
Strassen, V., 230
strategia „w głąb”, 257
strategia „wszerz”, 257
strategia gry, 284

struktury danych, 93
system operacyjny, 23
 DOS, 13
 Unix, 13
szachy, 285
szereg rekurencyjny liniowy, 69
sztuczna inteligencja, 282

T

tablica przesunięć, 213
techniki programowania, 223
 algorytmy żarłoczne, 234
 programowanie dynamiczne, 238
 schemat typu „dziel-i-rządź”, 224
 uwagi bibliograficzne, 243
tekst (pojęcie), 207
teoria gier, 257
transformacja kluczowa, 191
 konflikty dostępu, 197
 zastosowania, 204
Turing, A. M., 20

U

Uniwersalna Struktura Słownikowa, 152
USS, *Patrz* Uniwersalna Struktura Słownikowa

W

wielomiany
 dodawanie, 299
 mnożenie, 299
wieże Hanoi, 170; 177; 233
Wirth, N., 22
wskaźniki do funkcji, 112
wywołanie terminalne, 169
wyznacznika Vandermonde'a, 271
wzór Simpsona, *Patrz* całkowanie funkcji
wzór Stirlinga, *Patrz* różniczkowanie funkcji

Z

zadania, 11
zajętość pamięci programu, 54
złożoność obliczeniowa algorytmów, 53; 57
złożoność praktyczna algorytmu, 58
złożoność teoretyczna algorytmu, 59
zmiana dziedziny równania rekurencyjnego, 74
zmienna, 9
zmienna globalna, 175
zmienna lokalna, 175