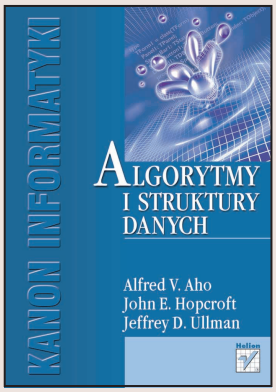
Algorytmy i struktury danych



Autorzy: Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
Tłumaczenie: Andrzej Grażyński
ISBN: 83-7361-177-0
Tytuł oryginału: Data Structures and Algorithms
Format: B5, stron: 442

W niniejszej książce przedstawiono struktury danych i algorytmy stanowiące podstawę współczesnego programowania komputerów. Algorytmy są niczym przepis na rozwiązanie postawionego przed programistę problemu. Są one nierozerwalnie związane ze strukturami danych — listami, rekordami, tablicami, kolejkami, drzewami... podstawowymi elementami wiedzy każdego programisty.
Książka obejmuje szeroki zakres materiału, a do jej lektury wystarczy znajomość dowolnego języka programowania strukturalnego (np. Pascala). Opis klasycznych algorytmów uzupełniono o algorytmy związane z zarządzaniem pamięcią operacyjną i pamięciami zewnętrznymi.

Książka przedstawia algorytmy i struktury danych w kontekście rozwiązywania problemów za pomocą komputera. Z tematyką rozwiązywania problemów powiązano zagadnienie zliczania kroków oraz złożoności czasowej — wynika to z głębokiego przekonania autorów tej książki, iż wraz z pojawianiem się coraz szybszych komputerów, pojawiać się będą także coraz bardziej złożone problemy do rozwiązywania i — paradoksalnie — złożoność obliczeniowa używanych algorytmów zyskiwać będzie na znaczeniu.

W książce omówiono m.in.:

- Tradycyjne struktury danych: listy, kolejki, stosy
- Drzewa i operacje na strukturach drzew
- Typy danych oparte na zbiorach, słowniki i kolejki priorytetowe wraz ze sposobami ich implementacji
- Grafy zorientowane i niezorientowane
- Algorytmy sortowania i poszukiwania mediany
- Asymptotyczne zachowanie się procedur rekurencyjnych
- Techniki projektowania algorytmów: „dziel i rządź”, wyszukiwanie lokalne i programowanie dynamiczne
- Zarządzanie pamięcią, B-drzewa i struktury indeksowe

Każdemu rozdziałowi towarzyszy zestaw ćwiczeń, o zróżnicowanym stopniu trudności, pomagających sprawdzić swoją wiedzę. „Algorytmy i struktury danych” to doskonały podręcznik dla studentów informatyki i pokrewnych kierunków, a także dla wszystkich zainteresowanych tą tematyką.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

12

1
Projektowanie i analiza algorytmów

Stworzenie programu rozwiązującego konkretny problem jest procesem wieloetapowym. Proces ten rozpoczyna się od sformułowania problemu i jego specyfikacji, po czym następuje projektowanie rozwiązania, które musi zostać następnie zapisane w postaci konkretnego programu, czyli zaimplementowane. Konieczne jest udokumentowanie oraz przetestowanie implementacji, a rozwiązanie wyprodukowane przez działający program musi zostać ocenione i zinterpretowane. W niniejszym rozdziale zaprezentowaliśmy nasze podejście do wymienionych kroków, następne rozdziały poświęcone są natomiast algorytmom i strukturom danych, składającym się na większość programów komputerowych.

1.1. Od problemu do programu

Wiedzieć, jaki problem tak naprawdę się rozwiązuje — to już połowa sukcesu. Większość problemów, w swym oryginalnym sformułowaniu, nie ma klarownej specyfikacji. Faktycznie, niektóre (wydawałoby się) oczywiste problemy, jak „sporządzenie smakowitego deseru" czy też „zachowanie pokoju na świecie" wydają się niemożliwe do sformułowania w takiej postaci, która przynajmniej otwierałaby drogę do ich rozwiązania za pomocą komputerów. Nawet jednak, gdy dany problem wydaje się (w sposób mniej lub bardziej oczywisty) możliwy do rozwiązania przy użyciu komputera, pozostaje jeszcze trudna zazwyczaj kwestia jego *parametryzacji*. Niekiedy bywa i tak, że rozsądne wartości parametrów mogą być określone jedynie w drodze eksperymentu.

Jeżeli niektóre aspekty rozwiązywanego problemu dają się wyrazić w kategoriach jakiegoś formalnego modelu, okazuje się to zwykle wielce pomocne, gdyż również w tych kategoriach poszukuje się rozwiązania, a dysponując konkretnym programem można określić (lub przynajmniej spróbować określić), czy faktycznie rozwiązuje on dany problem. Także — abstrahując od konkretnego programu — mając określoną wiedzę o modelu i jego właściwościach, można na tej podstawie podjąć próbę znalezienia dobrego rozwiązania.

Na szczęście, każda niemal dyscyplina naukowa daje się ująć w ramy modelu (modeli) z jakiejś dziedziny (dziedzin). Możliwe jest modelowanie wielu problemów natury wybitnie numerycznej w postaci układów równań liniowych (w ten sposób oblicza się m.in. natężenia prądu w poszczególnych gałęziach obwodu elektrycznego czy naprężenia w układzie połączonych belek) bądź równań różniczkowych (tak prognozuje się wzrost populacji i tak znajduje się stosunki ilościowe, w jakich łączą się substraty reakcji chemicznych). Rozwiązywanie problemów natury tekstowej czy symbolicznej odbywa się zazwyczaj na podstawie rozmaitych gramatyk wspomaganych zazwyczaj

obfitym repertuarem procedur wykonujących różne operacje na łańcuchach znaków (w taki sposób dokonuje się przecież kompilacja programu w języku wysokiego poziomu, tak również dokonuje się wyszukiwania konkretnych fraz tekstowych w obszernych bibliotekach).

Algorytmy

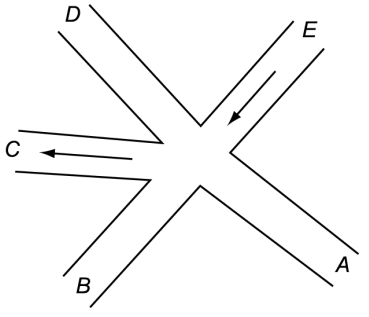
Kiedy już dysponujemy odpowiednim modelem matematycznym dla naszego problemu, możemy spróbować znaleźć rozwiązanie wyrażające się w kategoriach tegoż modelu. Naszym głównym celem jest poszukiwanie rozwiązania w postaci *algorytmu* — pod tym pojęciem rozumiemy skończoną sekwencję instrukcji, z których każda ma klarowne znaczenie i może być wykonana w skończonym czasie przy użyciu skończonego wysiłku. Dobrym przykładem klarownej instrukcji, wykonywalnej „w skończonym czasie i skończonym wysiłkiem", jest instrukcja przypisania w rodzaju `x := y + z`. Oczywiście, poszczególne instrukcje algorytmu mogą być wykonywane wielokrotnie i niekoniecznie w kolejności, w której zostały zapisane. Wynika stąd kolejne wymaganie stawiane algorytmowi — musi się on zatrzymywać po wykonaniu *skończonej* liczby instrukcji, niezależnie od danych wejściowych. Nasz program zasługuje więc na miano „algorytmu" tylko wtedy, gdy jego wykonywanie nigdy (czyli dla żadnych danych wejściowych) nie prowadzi do „ugrzęźnięcia" sterowania w nieskończonej pętli.

Co najmniej jeden aspekt powyższych rozważań nie jest do końca oczywisty. Określenie „klarowne znaczenie" ma mianowicie charakter na wskroś relatywny, ponieważ to, co jest oczywiste dla jednej osoby, może być wątpliwe dla innej. Ponadto wykonywanie się każdej z instrukcji w skończonym czasie może nie wystarczyć do tego, by „algorytm kończył swe działanie po wykonaniu skończonej liczby instrukcji". Za pomocą serii argumentów i kontrargumentów można jednak w większości przypadków osiągnąć porozumienie odnośnie tego, czy dana sekwencja instrukcji faktycznie może być uważana za algorytm. Zazwyczaj ostateczne wyjaśnienie wątpliwości w tym względzie jest powinnością osoby, która uważa się za autora algorytmu. W jednym z kolejnych punktów niniejszego rozdziału przedstawiliśmy sposób szacowania czasu wykonywania konstrukcji powszechnie spotykanych w językach programowania, dowodząc tym samym ich „skończoności czasowej".

Do zapisu algorytmów używać będziemy języka Pascal „wzbogaconego" jednakże o nieformalne opisy w języku naturalnym — programy zapisywane w powstałym w ten sposób „pseudojęzyku" nie nadają się co prawda do wykonania przez komputer, lecz powinny być intuicyjnie zrozumiałe dla Czytelnika, a to jest przecież w tej książce najważniejsze. Takie podejście umożliwia również ewentualne zastąpienie Pascala innym językiem, bardziej wygodnym dla Czytelnika, na etapie tworzenia „prawdziwych" programów. Pora teraz na pierwszy przykład, w którym zilustrowaliśmy większość etapów naszego podejścia do tworzenia programów komputerowych.

Przykład 1.1. Stworzymy model matematyczny, opisujący funkcjonowanie sygnalizacji świetlnej na skomplikowanym skrzyżowaniu. Ruch na takim skrzyżowaniu opisać można przez rozłożenie go na poszczególne „zakręty" z konkretnej ulicy w inną (dla przejrzystości przejazd *na wprost* również uważany jest za „zakręt"). Jeden cykl obsługi takiego skrzyżowania obejmuje pewną liczbę faz, w każdej fazie mogą być równocześnie wykonywane te zakręty (i tylko te), które ze sobą nie kolidują. Problem polega więc na określeniu poszczególnych faz ruchu i przyporządkowaniu każdego z możliwych zakrętów do konkretnej fazy w taki sposób, by w każdej fazie każda para zakrętów byłą parą niekolidującą. Oczywiście, rozwiązanie optymalne powinno wyznaczać *najmniejszą możliwą* liczbę faz.

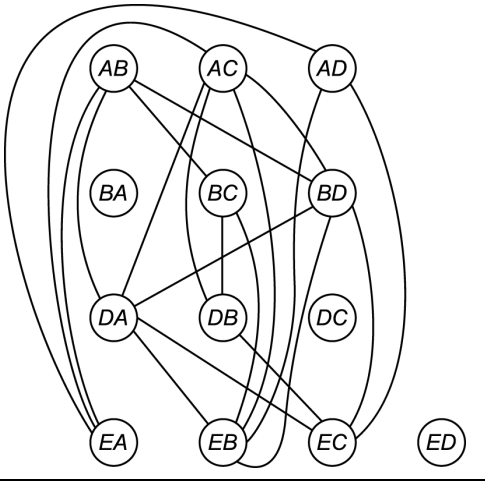
Rozpatrzmy konkretne skrzyżowanie, którego schemat przedstawiono na rysunku 1.1. Skrzyżowanie to znajduje się niedaleko Uniwersytetu Princeton i znane jest z trudności manewrowania,



RYSUNEK 1.1. Przykładowe skrzyżowanie

zwłaszcza przy zawracaniu. Ulice *C* i *E* są jednokierunkowe, pozostałe są ulicami dwukierunkowymi. Na skrzyżowaniu tym można wykonać 13 różnych „zakrętów"; niektóre z nich, jak *AB* (czyli z ulicy *A* w ulicę *B*) i *EC*, mogą być wykonywane równocześnie, inne natomiast, jak *AD* i *EB*, kolidują ze sobą i mogą być wykonane tylko w różnych fazach.

Opisany problem daje się łatwo ująć („modelować") w postaci struktury zwanej *grafem*. Każdy graf składa się ze zbioru *wierzchołków* i zbioru linii łączących wierzchołki w pary — linie te nazywane są *krawędziami*. W naszym grafie modelującym sterowanie ruchem na skrzyżowaniu wierzchołki reprezentować będą poszczególne zakręty, a połączenie dwóch wierzchołków krawędzią oznaczać będzie, że zakręty reprezentowane przez te wierzchołki kolidują ze sobą. Graf odpowiadający skrzyżowaniu pokazanemu na rysunku 1.1 przedstawiony jest na rysunku 1.2, natomiast w tabeli 1.1 widoczna jest jego reprezentacja *macierzowa* — każdy wiersz i każda kolumna reprezentuje konkretny wierzchołek; jedynka na przecięciu danego wiersza i danej kolumny wskazuje, że odpowiednie wierzchołki połączone są krawędzią.



RYSUNEK 1.2. Graf reprezentujący skrzyżowanie pokazane na rysunku 1.1

Rozwiązanie naszego problemu bazuje na koncepcji zwanej *kolorowaniem* grafu. Polega ona na przyporządkowaniu każdemu wierzchołkowi konkretnego koloru w taki sposób, by wierzchołki *połączone* krawędzią miały *różne* kolory. Nietrudno się domyślić, że optymalne rozwiązanie naszego problemu sprowadza się do znalezienia takiego kolorowania grafu z rysunku 1.2, które wykorzystuje jak najmniejszą liczbę kolorów.

TABELA 1.1.
Macierzowa reprezentacja grafu z rysunku 1.2

	AB	*AC*	*AD*	*BA*	*BC*	*BD*	*DA*	*DB*	*DC*	*EA*	*EB*	*EC*	*ED*
AB					1	1	1			1			
AC						1	1	1		1	1		
AD										1	1	1	
BA													
BC	1							1			1		
BD	1	1					1				1	1	
DA	1	1				1					1	1	
DB		1			1							1	
DC													
EA	1	1	1										
EB		1	1		1	1	1						
EC			1			1	1	1					
ED													

Zagadnienie kolorowania grafu studiowane było przez dziesięciolecia i obecna wiedza na temat algorytmów zawiera wiele propozycji w tym względzie. Niestety, problem kolorowania dowolnego grafu przy użyciu najmniejszej możliwej liczby kolorów zalicza się do ogromnej klasy tzw. problemów *NP-zupełnych*, dla których jedynymi znanymi rozwiązaniami są rozwiązania typu „sprawdź wszystkie możliwości". W przełożeniu na kolorowanie grafu oznacza to sukcesywne próby wykorzystania jednego koloru, potem dwóch, trzech itd. tak długo, aż w końcu liczba użytych kolorów okaże się wystarczająca (czego stwierdzenie również wymaga „sprawdzenia wszystkich możliwości"). Co prawda wykorzystanie specyfiki konkretnego grafu może ten proces nieco przyspieszyć, jednakże w ogólnym wypadku nie istnieje alternatywa dla oczywistego „sprawdzania wszystkich możliwości".

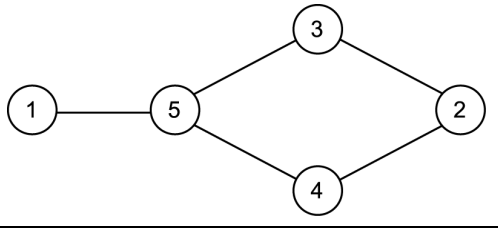
Okazuje się więc, że znalezienie optymalnego pokolorowania grafu jest w ogólnym przypadku procesem bardzo kosztownym i dla dużych grafów może okazać się zupełnie nie do przyjęcia, niezależnie od tego, jak efektywnie skonstruowany byłby program rozwiązujący zagadnienie. Zastosujemy w związku z tym trzy możliwe podejścia. Po pierwsze, dla małych grafów duży koszt czasowy nie będzie zbytnią przeszkodą i „sprawdzenie wszystkich możliwości" będzie zadaniem wykonalnym w rozsądnym czasie. Drugie podejście polegać będzie na wykorzystaniu specjalnych własności grafu, pozwalających na rezygnację a priori z dość dużej liczby sprawdzeń. Trzecie podejście odwoływać się będzie natomiast do znanej prawdy, że rozwiązanie problemu można znacznie przyspieszyć, rezygnując z poszukiwania rozwiązania *optymalnego* i zadowalając się rozwiązaniem *dobrym*, możliwym do zaakceptowania w konkretnych warunkach i często niewiele gorszym od optymalnego — tym bardziej, że wiele rzeczywistych skrzyżowań nie jest aż tak skomplikowanych, jak pokazane na rysunku 1.1. Takie algorytmy, szybko dostarczające dobrych, lecz niekoniecznie optymalnych rozwiązań, nazywane są algorytmami *heurystycznymi*.

Jednym z heurystycznych algorytmów kolorowania grafu jest tzw. algorytm „zachłanny" (ang. *greedy*). Kolorujemy *pierwszym kolorem* tyle wierzchołków, ile tylko możemy. Następnie wybieramy *kolejny kolor* i wykonać należy kolejno następujące czynności:

(1) Wybierz dowolny niepokolorowany jeszcze wierzchołek i przyporządkuj mu aktualnie używany kolor.

(2) Przejrzyj listę wszystkich niepokolorowanych jeszcze wierzchołków i dla każdego z nich sprawdź, czy jest połączony krawędzią z jakimś wierzchołkiem mającym aktualnie używany kolor. Jeżeli nie jest, opatrz go tym kolorem.

„Zachłanność" algorytmu wynika stąd, że w każdym kroku (tj. przy każdym z używanych kolorów) stara się on pokolorować *jak największą* liczbę wierzchołków, niezależnie od ewentualnych negatywnych konsekwencji tego faktu. Nietrudno wskazać przypadek, w którym mniejsza zachłanność prowadzi do zastosowania mniejszej liczby kolorów. Graf przedstawiony na rysunku 1.3 można pokolorować przy użyciu tylko dwóch kolorów; jednego dla wierzchołków *1*, *3* i *4*, drugiego dla pozostałych. Algorytm zachłanny, analizujący wierzchołki w kolejności wzrastającej numeracji, rozpocząłby natomiast od przypisania pierwszego koloru wierzchołkom *1* i *2*, po czym wierzchołki *3* i *4* otrzymałyby drugi kolor, a dla wierzchołka *5* konieczne byłoby użycie *trzeciego* koloru.



RYSUNEK 1.3. Przykładowy graf, dla którego nie popłaca zachłanność algorytmu kolorowania

Zastosujmy teraz „zachłanne" kolorowanie do grafu z rysunku 1.2. Rozpoczynamy od wierzchołka *AB*, nadając mu pierwszy z kolorów — niebieski; kolorem tym możemy także opatrzyć[1] wierzchołki *AC*, *AD* i *BA*, ponieważ są one „osamotnione". Nie możemy nadać koloru niebieskiego wierzchołkowi *BC*, gdyż jest on połączony z (niebieskim) wierzchołkiem *AB*; z podobnych względów nie możemy także pokolorować na niebiesko wierzchołków *BD*, *DA* i *DB*, możemy to jednak zrobić z wierzchołkiem *DC*. Wierzchołkom *EA*, *EB* i *EC* nie można przyporządkować koloru niebieskiego — w przeciwieństwie do „osamotnionego" wierzchołka *ED*.

Weźmy kolejny kolor — czerwony. Przyporządkowujemy go wierzchołkom *BC* i *BD*; wierzchołek *DA* łączy się krawędzią z *BD*, więc nie może mieć koloru czerwonego, podobnie jak wierzchołek *DB* łączący się z *BC*. Spośród niepokolorowanych jeszcze wierzchołków tylko *EA* można nadać kolor czerwony.

Pozostały cztery niepokolorowane wierzchołki: *DA*, *DB*, *EB* i *EC*. Jeżeli przypiszemy *DA* kolor zielony, możemy go także przyporządkować *DB*, jednakże dla *EB* i *EC* musimy wtedy użyć kolejnego (żółtego) koloru. Ostateczny wynik kolorowania przedstawiamy w tabeli 1.2. Dla każdego koloru w kolumnie *Ekstra* prezentujemy te wierzchołki, które nie są połączone z żadnym wierzchołkiem w tym kolorze i przy innej kolejności przechodzenia zachłannego algorytmu przez wierzchołki mogłyby ten właśnie kolor uzyskać.

Zgodnie z wcześniejszymi założeniami, wszystkie wierzchołki w danym *kolorze* (kolumna *Wierzchołki*) odpowiadają zakrętom „uruchamianym" w danej *fazie*. Oprócz nich można także uruchomić inne zakręty, które z nimi nie kolidują (mimo że reprezentujące je wierzchołki mają inny kolor), są one wyszczególnione w kolumnie *Ekstra*.

Tak więc wedle otrzymanego rozwiązania, sygnalizacja sterująca ruchem na skrzyżowaniu pokazanym na rysunku 1.1 powinna być sygnalizacją *czterofazową*. Po doświadczeniach z grafem przedstawionym na rysunku 1.3 nie można nie zastanawiać się, czy jest rozwiązanie optymalne, tzn. czy nie byłoby możliwe rozwiązanie problemu przy użyciu sygnalizacji trój- czy dwufazowej.

[1] Zgodnie z kolejnością przyjętą w tabeli na rysunku 1.3 — *przyp. tłum.*

TABELA 1.2.
Jeden ze sposobów pokolorowania grafu z rysunku 1.2 za pomocą „zachłannego" algorytmu

Kolor	Wierzchołki	Ekstra
niebieski	*AB*, *AC*, *AD*, *BA*, *DC*, *ED*	—
czerwony	*BC*, *BD*, *EA*	*BA*, *DC*, *ED*
zielony	*DA*, *DB*	*AD*, *BA*, *DC*, *ED*
żółty	*EB*, *EC*	*BA*, *DC*, *EA*, *ED*

Rozstrzygniemy tę wątpliwość za pomocą elementarnych wiadomości z teorii grafów. Nazwijmy *k-kliką* zbiór *k* wierzchołków, w których *każde dwa* połączone są krawędziami. Jest oczywiste, że nie da się pokolorować *k*-kliki przy użyciu mniej niż *k* kolorów. Gdy spojrzymy uważnie na rysunek 1.2 spostrzeżemy, że wierzchołki *AC*, *DA*, *BD* i *EB* tworzą 4-klikę, wykluczone jest zatem pokolorowanie grafu wykorzystując mniej niż 4 kolory, zatem nasze rozwiązanie jest optymalne pod względem liczby faz. □

Pseudojęzyki i stopniowe precyzowanie

Gdy tylko znajdziemy odpowiedni model matematyczny dla rozwiązywanego problemu, możemy przystąpić do formułowania algorytmu w kategoriach tego modelu. Początkowa wersja takiego algorytmu składa się zazwyczaj z bardzo ogólnych instrukcji, które w kolejnych krokach należy uściślić, czyli zastąpić instrukcjami bardziej precyzyjnymi. Przykładowo, sformułowanie „wybierz dowolny niepokolorowany jeszcze wierzchołek" jest intuicyjnie zrozumiałe dla osoby studiującej algorytm, lecz aby z algorytmu tego stworzyć program możliwy do wykonania przez komputer, sformułowania takie muszą zostać poddane stopniowej formalizacji. Postępowanie takie nazywa się *stopniowym precyzowaniem* (ang. *stepwise refinement*) i prowadzi do uzyskania programu w pełni zgodnego ze składnią konkretnego języka programowania.

Przykład 1.2. Poddajmy więc stopniowemu precyzowaniu „zachłanny" algorytm kolorowania grafu, a dokładniej jego część związaną z konkretnym kolorem. Zakładamy wstępnie, że mamy do czynienia z grafem *G*, którego niektóre wierzchołki mogą być pokolorowane. Poniższa procedura tworzy zbiór wierzchołków *newclr*, którym należy przyporządkować odnośny kolor. Procedura ta wywoływana jest cyklicznie tak długo, aż wszystkim węzłom grafu przyporządkowane zostaną kolory. Pierwsze, najbardziej ogólne sformułowanie wspomnianej procedury może wyglądać tak, jak na listingu 1.1.

LISTING 1.1.
Początkowa wersja procedury greedy

```
    procedure greedy ( var G: GRAPH; var newclr: SET );
        { greedy zapisuje w newclr zbiór wierzchołków, którym należy
          przyporządkować ten sam kolor }
        begin
{1}         newclr:= ∅;[2]
```

[2] Symbol ∅ oznacza zbiór pusty.

```
{2}         for <v reprezentujący każdy niepokolorowany jeszcze wierzchołek w G> do
{3}             if <v nie jest połączony krawędzią z żadnym wierzchołkiem wchodzącym
                    w skład newclr > then begin
{4}                 oznacz v jako pokolorowany;
{5}                 dodaj v do newclr
                end
        end; {greedy}
```

Na listingu 1.1 widoczne są podstawowe elementy zapisu algorytmu w pseudojęzyku. Po pierwsze, widzimy zapisane tłustym drukiem i małymi literami słowa kluczowe, które odpowiadają zastrzeżonym słowom języka Pascal. Po drugie, słowa pisane w całości wielkimi literami — GRAPH i SET[3] są nazwami abstrakcyjnych typów danych. W procesie stopniowego precyzowania typy te muszą zostać zdefiniowane w postaci typów danych pascalowych oraz procedur wykonujących na tych danych określone operacje. Abstrakcyjnymi typami danych zajmiemy się szczegółowo w dwóch następnych punktach.

Konstrukcje strukturalne Pascala, jak: **if**, **for** i **while** są użyte w naszym pseudojęzyku w oryginalnej postaci, jednakże już np. warunek w instrukcji **if** (w wierszu {3}) zapisany jest w sposób zupełnie nieformalny, podobnie zresztą jak wyrażenie stojące po prawej stronie instrukcji przypisania w wierszu {1}. Nieformalnie został także zapisany zbiór, po którym iterację przeprowadza instrukcja **for** w wierszu {2}.

Nie będziemy szczegółowo opisywać procesu przekształcania przedstawionej procedury do poprawnego programu pascalowego, zaprezentujemy jedynie transformację instrukcji **if** z wiersza {3} do bardziej precyzyjnej postaci.

Aby określić, czy dany wierzchołek *v* połączony jest krawędzią z którymś z wierzchołków należących do zbioru *newclr*, rozpatrzymy każdy element *w* tego zbioru i będziemy sprawdzać, czy w grafie *G* istnieje krawędź łącząca wierzchołki *v oraz w*. Wynik sprawdzenia zapisywać będziemy w zmiennej boolowskiej *found* — jeśli rzeczona krawędź istnieje, zmiennej tej nadana zostanie wartość true.

Zmieniona wersja procedury przedstawiona została na listingu 1.2.

LISTING 1.2.
Rezultat częściowego sprecyzowania procedury greedy

```
    procedure greedy ( var G: GRAPH; var newclr: SET );
        { greedy zapisuje w newclr zbiór wierzchołków, którym należy
          przyporządkować ten sam kolor }
        begin
{1}         newclr := ∅;
{2}         for <v reprezentujący każdy niepokolorwany jeszcze wierzchołek w G>
            do begin
{3.1}           found := false;
{3.2}           for <w reprezentującego każdy wierzchołek w zbiorze newclr> do
{3.3}               if <istnieje w grafie G krawędź łącząca wierzchołki v i w> then
{3.4}               found := true;
{3.5}           if found = false then begin
                { v nie łączy się z żadnym wierzchołkiem ze zbioru newclr }
{4}                 oznacz v jako pokolorowany;
{5}                 dodaj v do newclr
```

[3] Odróżniamy tu abstrakcyjny typ danych SET od typu zbiorowego set języka Pascal.

```
{4}                 end
                end
            end; {greedy}
```

Zredukowaliśmy tym samym nasz algorytm do zbioru operacji wykonywanych na dwóch zbiorach wierzchołków. Pętla zewnętrzna (wiersze od {2} do {5}) stanowi iterację po niepokolorowanych wierzchołkach grafu *G*; pętla wewnętrzna (wiersze od {3.2} do {3.4}) iteruje natomiast po wierzchołkach znajdujących się aktualnie w zbiorze *newclr*. Instrukcja w wierszu {5} dodaje nowy wierzchołek do zbioru *newclr*.

Język Pascal oferuje wiele możliwości reprezentowania zbiorów danych — niektórymi z nich zajmiemy się dokładniej w rozdziałach 4. i 5. Na potrzeby reprezentowania zbioru wierzchołków w naszym przykładzie wykorzystamy inny abstrakcyjny typ danych, LIST, który może być zaimplementowany jako lista liczb całkowitych (integer), zakończona specjalną wartością *null* (która może być reprezentowana przez wartość 0). Lista ta może mieć postać tablicy, lecz — jak zobaczymy w rozdziale 2. — istnieje wiele innych sposobów jej reprezentowania.

Zastąpmy instrukcję **for** w wierszu 3.2 na listingu 1.2 przez pętlę, w której zmienna *w* reprezentuje kolejne wierzchołki zbioru *newclr* (w kolejnych obrotach pętli). Identycznemu zabiegowi poddamy pętlę rozpoczynającą się w wierszu {2}. W rezultacie otrzymamy nową, bardziej precyzyjną wersję procedury, która zaprezentowana jest na listingu 1.3.

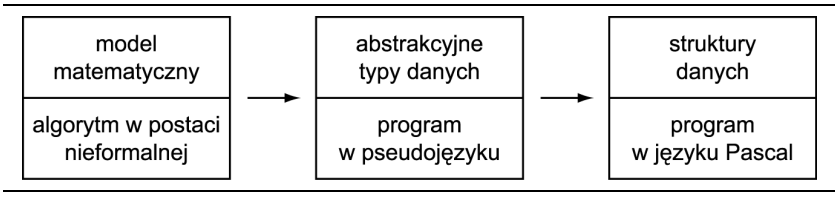
LISTING 1.3.
Kolejny etap sprecyzowania procedury *greedy*

```
procedure greedy (var G: GRAPH; var newclr: LIST);
    { greedy zapisuje w newclr zbiór wierzchołków, którym należy
      przyporządkować ten sam kolor }
    var
        found : boolean;
        v, w : integer;
    begin
        newclr := 0;
        v := <pierwszy niepokolorowany wierzchołek w grafie G>;
        while v <> null do begin
            found := false;
            w := <pierwszy wierzchołek ze zbioru newclr>;
            while w <> null do begin
                if <istnieje w grafie G krawędź łącząca wierzchołki v i w> then
                    found := true;
                w := <następny wierzchołek ze zbioru newclr>;
            end;
            if found = false do begin
                oznacz v jako pokolorowany;
                dodaj v do newclr
            end;
            v := <następny niepokolorowany wierzchołek w grafie G>;
        end
    end; { greedy }
```

Procedurze pokazanej na listingu 1.3 sporo jeszcze brakuje do tego, by została zaakceptowana przez kompilator Pascala. Poprzestaniemy jednak na tym etapie jej precyzowania, gdyż chodzi nam raczej o prezentację określonego sposobu postępowania niż ostateczny wynik. □

Podsumowanie

Na rysunku 1.4 przedstawiamy schemat procesu tworzenia programu, zgodnie z ujęciem w niniejszej książce. Proces ten rozpoczyna się etapem modelowania, na którym wybrany zostaje odpowiedni model matematyczny dla danego problemu (np. *graf*). Na tym etapie opis algorytmu ma zazwyczaj postać wybitnie nieformalną.



RYSUNEK 1.4. Proces rozwiązywania problemu za pomocą komputera

Kolejnym krokiem jest zapisanie algorytmu w pseudojęzyku stanowiącym mieszankę instrukcji pascalowych i nieformalnych opisów wyrażonych w języku naturalnym. Realizacja tego etapu polega na stopniowym precyzowaniu ogólnych, nieformalnych opisów do bardziej szczegółowej postaci. Niektóre fragmenty zapisu algorytmu mogą mieć już wystarczająco szczegółową postać do tego, by można było wyrazić je w kategoriach konkretnych operacji wykonywanych na konkretnych danych, w związku z czym nie muszą być już bardziej precyzowane. Po odpowiednim sprecyzowaniu instrukcji algorytmu, definiujemy abstrakcyjne typy danych dla wszystkich struktur używanych przez algorytm (z wyjątkiem być może struktur skrajnie elementarnych, jak: liczby całkowite, liczby rzeczywiste czy łańcuchy znaków). Z każdym abstrakcyjnym typem danych wiążemy zestaw odpowiednio nazwanych procedur, z których każda wykonuje konkretną operację na danych tego typu. Każda nieformalnie zapisana operacja zostaje następnie zastąpiona wywołaniem odpowiedniej procedury.

W trzecim kroku wybieramy odpowiednią implementację dla każdego typu danych, w szczególności dla związanych z tym typem procedur wykonujących konkretne operacje. Zastępujemy także istniejące jeszcze nieformalne zapisy „prawdziwymi" instrukcjami języka Pascal. W efekcie otrzymujemy program, który można skompilować i uruchomić. Po cyklu testowania i usuwania błędów (mamy nadzieję — krótkiego) otrzymamy poprawny program dostarczający upragnione rozwiązanie.

1.2. Abstrakcyjne typy danych

Większość omawianych dotychczas zagadnień powinna być znana nawet początkującym programistom. Jedynym istotnym *novum* mogą być abstrakcyjne typy danych, celowe więc będzie uświadomienie sobie ich roli w szeroko rozumianym procesie projektowania programów. W tym celu posłużymy się analogią — dokonamy mianowicie wyszczególnienia wspólnych cech abstrakcyjnych typów danych i *procedur* pascalowych.

Procedury, jako podstawowe narzędzie każdego języka algorytmicznego, stanowią tak naprawdę *uogólnienie operatorów*. Uwalniają one od kłopotliwego ograniczenia do podstawowych operacji (w rodzaju dodawania czy mnożenia liczb), pozwalając na dokonywanie operacji bardziej zaawansowanych, jak np. mnożenie macierzy.

Inną użyteczną cechą procedur jest *enkapsulacja* niektórych fragmentów kodu. Określony fragment programu, związany ściśle z pewnym aspektem funkcjonalnym programu, zamykany jest w ramach ściśle zlokalizowanej sekcji. Jako przykład posłuży procedura dokonująca wczytywania i weryfikacji danych. Jeżeli w pewnym momencie okaże się, że program powinien (powiedzmy)

oprócz liczb dziesiętnych honorować także liczby w postaci szesnastkowej, niezbędne zmiany trzeba będzie wprowadzić jedynie w ściśle określonym fragmencie kodu, bez ingerowania w inne fragmenty programu.

Definiowanie abstrakcyjnych typów danych

Abstrakcyjne typy danych, często dla prostoty oznaczane skrótem *ADT* (ang. *Abstract Data Types*), mogą być traktowane jak model matematyczny, z którym związano określoną kolekcję operacji. Przykładem prostego ADT może być zbiór liczb całkowitych, w stosunku do którego określono operacje sumy, iloczynu i różnicy (w rozumieniu teorii mnogości). *Argumentami* operacji związanych z określonym ADT mogą być także dane innych typów, dotyczy to także *wyniku* operacji. Przykładowo, wśród operacji związanych z ADT reprezentującym zbiór liczb całkowitych znajdować się mogą procedury badające przynależność określonego elementu do zbioru, zwracające liczbę elementów czy też tworzące nowy zbiór złożony z podanych parametrów. Tak czy inaczej, macierzysty ADT wystąpić musi jednak *co najmniej raz* jako argument którejś ze swych procedur.

Dwie wspomniane wcześniej cechy procedur — uogólnienie i enkapsulacja — stosują się jako żywo do abstrakcyjnych typów danych. Tak jak procedury stanowią uogólnienie elementarnych operatorów (+, -, *, / itd.), tak abstrakcyjne typy danych są uogólnieniem elementarnych typów danych (`integer`, `real`, `boolean` itd.). Odpowiednikiem enkapsulacji kodu przez procedury jest natomiast enkapsulacja typu danych — w tym sensie, że definicja struktury konkretnego ADT *wraz z towarzyszącymi tej strukturze operacjami* zamknięta zostaje w ściśle zlokalizowanym fragmencie programu. Każda zmiana postaci czy zachowania ADT uskuteczniana jest wyłącznie w jego definicji, natomiast przez pozostałe fragmenty ów ADT traktowany jest — to ważne — jako *elementarny* typ danych. Pewnym odstępstwem od tej reguły jest sytuacja, w której dwa różne ADT są ze sobą powiązane, czyli procedury jednego ADT uzależnione są od pewnych szczegółów drugiego. Enkapsulacja nie jest wówczas całkowita i niektórych zmian dokonać trzeba na ogół w definicjach obydwu ADT.

By dokładniej zrozumieć podstawowe idee tkwiące u podstaw koncepcji abstrakcyjnych typów danych, przyjrzyjmy się dokładniej typowi `LIST` wykorzystywanemu w procedurze *greedy* na listingu 1.3. Reprezentuje on listę liczb całkowitych (`integer`) o nazwie *newclr*, a podstawowe operacje wykonywane w kontekście tej listy są następujące:

(1) Usuń wszystkie elementy z listy.
(2) Udostępnij pierwszy element listy lub wartość *null*, jeżeli lista jest pusta.
(3) Udostępnij kolejny element listy lub wartość *null*, jeżeli wyczerpano zbiór elementów.
(4) Wstaw do listy nowy element (liczbę całkowitą).

Istnieje wiele struktur danych, które mogłyby posłużyć do efektywnej implementacji typu `LIST` — zajmiemy się nimi dokładniej w rozdziale 2. Gdybyśmy w zapisie algorytmu na listingu 1.3, używającym powyższych opisów, zastąpili je wywołaniami procedur:

```
(1) MAKENULL(newclr);
(2) w := FIRST(newclr);
(3) w := NEXT(newclr);
(4) INSERT(v, newclr);
```

od razu widoczna stałaby się podstawowa zaleta abstrakcyjnych typów danych. Otóż program korzystający z ADT ogranicza się tylko do wywoływania związanych z nim procedur — ich implementacja jest dla niego obojętna. Dokonując zmian w tej implementacji nie musimy ingerować w wywołania procedur.

Drugim abstrakcyjnym typem danych, używanym przez procedurę *greedy*, jest GRAPH reprezentujący graf, z następującymi operacjami podstawowymi:

(1) Udostępnij pierwszy niepokolorowany wierzchołek.
(2) Sprawdź, czy dwa podane wierzchołki połączone są krawędzią.
(3) Przyporządkuj kolor wierzchołkowi.
(4) Udostępnij kolejny niepokolorowany wierzchołek.

Wymienione cztery operacje są co prawda wystarczające w procedurze *greedy*, lecz oczywiście zestaw podstawowych operacji na grafie powinien być nieco bogatszy i powinien obejmować na przykład: dodanie krawędzi między podanymi wierzchołkami, usunięcie konkretnej krawędzi, usunięcie kolorowania z wszystkich wierzchołków itp. Istnieje wiele struktur danych zdolnych do reprezentowania grafu w tej postaci — przedstawimy je dokładniej w rozdziałach 6. i 7.

Należy wyraźnie zaznaczyć, że nie istnieje ograniczenie liczby operacji podstawowych związanych z danym modelem matematycznym. Każdy zbiór tych operacji definiuje odrębny ADT. Przykładowo, zestaw operacji podstawowych dla abstrakcyjnego typu danych SET mógłby być następujący:

(1) MAKENULL(*A*) — procedura usuwająca wszystkie elementy ze zbioru *A*,
(2) UNION(*A*, *B*, *C*) — procedura przypisująca zbiorowi *C* sumę zbiorów *A* i *B*,
(3) SIZE(*A*) — funkcja zwracająca liczbę elementów w zbiorze *A*.

Implementacja abstrakcyjnego typu danych polega na zdefiniowaniu jego odpowiednika (jako typu) *w kategoriach konkretnego języka programowania* oraz zapisaniu (również w tym języku) procedur implementujących jego podstawowe operacje. „Typ" w języku programowania stanowi zazwyczaj kombinację typów elementarnych tego języka oraz obecnych w tym języku *mechanizmów agregujących*. Najważniejszymi mechanizmami agregującymi języka Pascal są *tablice* i *rekordy*. Na przykład abstrakcyjny zbiór SET zawierający liczby całkowite może być zaimplementowany jako *tablica* liczb całkowitych.

Należy także podkreślić jeden istotny fakt. Abstrakcyjny typ danych jest kombinacją modelu matematycznego *i zbioru operacji*, jakie można na tym modelu wykonywać, dlatego dwa identyczne modele, połączone z różnymi zbiorami operacji, określają różne ADT. Większość materiału zawartego w niniejszej książce poświęcona jest badaniu podstawowych modeli matematycznych, jak zbiory i grafy, i znajdowaniu najodpowiedniejszych (w konkretnych sytuacjach) zestawów operacji dla tych modeli.

Byłoby wspaniale, gdybyśmy mogli tworzyć programy w językach, których elementarne typy danych i operacje są jak najbliższe używanym przez nas modelom i operacjom abstrakcyjnych typów danych. Język Pascal (pod wieloma względami) nie jest najlepiej przystosowany do odzwierciedlania najczęściej używanych ADT, w dodatku nieliczne języki, w których abstrakcyjne typy danych deklarować można bezpośrednio, nie są powszechnie znane (o niektórych z nich wspominamy w notce bibliograficznej).

1.3. Typy danych, struktury danych i ADT

Mimo że określenia „typ danych" (lub po prostu „typ"), „struktura danych" i „abstrakcyjny typ danych" brzmią podobnie, ich znaczenie jest całkowicie odmienne. W terminologii języka programowania „typem danych" nazywamy zbiór wartości, jakie przyjmować mogą zmienne tego typu. Na przykład zmienna typu boolean przyjmować może tylko dwie wartości: true i false. Poszczególne języki

programowania różnią się od siebie zestawem elementarnych typów danych; elementarnymi typami danych języka Pascal są: liczby całkowite (`integer`), liczby rzeczywiste (`real`), wartości boolowskie (`boolean`) i znaki (`char`). Także mechanizmy agregacyjne, za pomocą których tworzy się typy złożone z typów elementarnych, różne są w różnych językach — niebawem zajmiemy się mechanizmami agregacyjnymi Pascala.

Abstrakcyjnym typem danych (*ADT*) jest model, z którym skojarzono zestaw operacji podstawowych. Jak już wspominaliśmy, możemy formułować algorytmy w kategoriach ADT, chcąc jednak zaimplementować dany algorytm w konkretnym języku programowania, musimy znaleźć sposób reprezentowania tych ADT w kategoriach typów danych i operatorów właściwych temu językowi. Do reprezentowania modeli matematycznych składających się na poszczególne ADT służą *struktury danych*, które stanowią kolekcje zmiennych (być może różnych typów) połączonych ze sobą na różne sposoby.

Podstawowymi blokami tworzącymi struktury danych są *komórki* (ang. *cells*). Komórkę można obrazowo opisać jako skrzynkę, w której można przechowywać pojedynczą wartość należącą do danego typu (elementarnego lub złożonego). Struktury danych tworzy się przez nadanie nazwy agregatom takich komórek i (opcjonalnie) przez zinterpretowanie zawartości niektórych komórek jako połączenia (czyli wskaźnika) między komórkami.

Najprostszym mechanizmem agregującym, obecnym w Pascalu i większości innych języków programowania, jest (jednowymiarowa) *tablica* (ang. *array*) stanowiąca sekwencję komórek zawierających wartości określonego typu, zwanego często *typem bazowym*. Pod względem matematycznym można postrzegać tablicę jako odwzorowanie zbioru indeksów (którymi mogą być liczby całkowite) w typ bazowy. Konkretna komórka w ramach konkretnej tablicy może być identyfikowana w postaci nazwy, której towarzyszy konkretna wartość indeksu. W języku Pascal indeksami mogą być m.in. liczby całkowite z określonego przedziału (noszącego nazwę *typu okrojonego*) oraz wartości typu wyliczeniowego, jak np. typ (*czarny*, *niebieski*, *czerwony*, *zielony*). Typ bazowy tablicy może być w zasadzie dowolnym typem, tak więc deklaracja:

```
name: array [indextype] of celltype;
```

określa tablicę o nazwie *name*, złożoną z elementów typu bazowego `celltype`, indeksowanych wartościami typu `indextype`.

Język Pascal jest skądinąd niezwykle bogaty pod względem możliwości wyboru typu indeksowego. Niektóre inne języki, jak Fortran, dopuszczają w tej roli jedynie liczby całkowite (z określonego przedziału). Chcąc w takiej sytuacji użyć np. znaków w roli typu indeksowego, musielibyśmy uciec się do jakiegoś ich odwzorowania w liczby całkowite, np. bazując na ich kodach ASCII.

Innym powszechnie używanym mechanizmem agregującym są *rekordy*. Rekord jest komórką składającą się z innych komórek zwanych *polami*, mających na ogół różne typy. Rekordy często łączone są w tablice — typ rekordowy staje się wówczas typem bazowym tablicy. Deklaracja pascalowa:

```
var
    reclist: array[1..4] of record
        data: real;
        next: integer;
    end;
```

określa czteroelementową tablicę, której komórka jest rekordem zawierającym dwa pola: *data* i *next*.

Trzecim mechanizmem agregującym, dostępnym w Pascalu i niektórych innych językach, są *pliki*. Plik, podobnie jak jednowymiarowa tablica, stanowi sekwencję elementów określonego typu. W przeciwieństwie jednak do tablicy, plik nie podlega indeksowaniu; elementy dostępne są tylko w takiej kolejności w jakiej fizycznie występują w pliku. Poszczególne elementy tablicy (i poszczególne

pola rekordu) są natomiast dostępne w sposób bezpośredni, czyli szybciej niż w pliku. Plik odróżnia jednak od tablicy istotna zaleta — jego wielkość (liczba zawartych w nim elementów) może zmieniać się w czasie i jest potencjalnie nieograniczona.

Wskaźniki i kursory

Oprócz mechanizmów agregujących istnieją jeszcze inne sposoby ustanawiania relacji między komórkami — służą do tego *wskaźniki* i *kursory*. Wskaźnik jest komórką, której zawartość jednoznacznie identyfikuje inną komórkę. Fakt, że komórka *A* jest wskaźnikiem komórki *B*, zaznaczamy na schemacie struktury danych rysując strzałkę od *A* do *B*.

W języku Pascal to, że zmienna *ptr* może wskazywać komórkę o typie `celltype`, zaznaczamy w następujący sposób:

```
var
    ptr: ↑celltype;
```

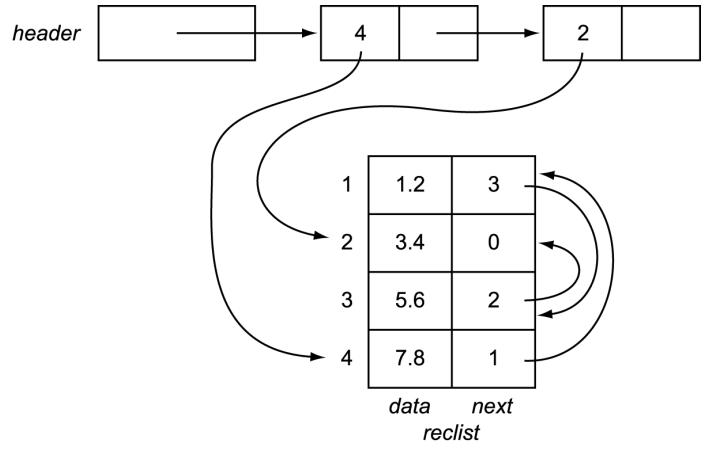
Strzałka poprzedzająca nazwę typu bazowego oznacza typ wskaźnikowy (czyli zbiór wartości stanowiących wskazania na komórkę o typie `celltype`). Odwołanie do komórki wskazywanej przez zmienną *ptr* (zwane także *dereferencją* wskaźnika) ma postać *ptr*↑ — strzałka występuje *za* nazwą zmiennej.

Kursor tym różni się od wskaźnika, że identyfikuje komórkę *w ramach konkretnej tablicy* — wartością kursora jest indeks odnośnego elementu. W samym zamyśle nie różni się on od wskaźnika — jego zadaniem jest także identyfikowanie komórki — jednak, w przeciwieństwie do niego, nie można za pomocą kursora identyfikować komórek „samodzielnych", które nie wchodzą w skład tablicy. W niektórych językach, jak np. Fortran i Algol, wskaźniki po prostu nie istnieją i jedyną metodą identyfikowania komórek są właśnie kursory. Należy także zauważyć, że w Pascalu nie jest możliwe utworzenie wskaźnika do konkretnej komórki tablicy, więc jedynie kursory umożliwiają identyfikowanie poszczególnych komórek. Niektóre języki, jak PL/I i C, są pod tym względem bardziej elastyczne i dopuszczają wskazywanie elementów tablic przez „prawdziwe" wskaźniki.

Na schemacie struktury danych kursory zaznaczane są podobnie jak wskaźniki, czyli za pomocą strzałek, a dodatkowo w komórkę będącą kursorem może być wpisana jej zawartość[4] dla zaznaczenia, iż nie mamy do czynienia z „typowym" wskaźnikiem.

Przykład 1.3. Na rysunku 1.5 pokazano strukturę danych składającą się z dwóch części: tablicy *reclist* (zdefiniowanej wcześniej w tym rozdziale) i kursorów do elementów tej tablicy; kursory te połączone są w listę łańcuchową. Elementy tablicy *reclist* są rekordami. Pole *next* każdego z tych rekordów jest kursorem do „następnego" rekordu i zgodnie z tą konwencją, na rysunku 1.5 rekordy tablicy uporządkowane są w kolejności 4, 1, 3, 2. Zwróć uwagę, że pole *next* rekordu 2 zawiera wartość 0, oznaczającą kursor *pusty*, czyli nie identyfikujący żadnej komórki, konwencja taka ma sens jedynie wtedy, gdy komórki tablicy indeksowane są począwszy od 1, nie od zera.

[4] Wynika to z jeszcze jednej, fundamentalnej różnicy między wskaźnikiem a kursorem. Otóż implementacja wskaźników w Pascalu (i wszystkich niemal językach, w których wskaźniki są obecne) bazuje na *adresie* komórki w przestrzeni adresowej procesu. Adres ten (a więc i konkretna wartość wskaźnika) ma sens *jedynie w czasie wykonywania* programu — nie istnieje więc jakakolwiek *wartość* wskaźnika, którą można by umieścić na schemacie. Kursor natomiast jest wielkością absolutną, pozostającą bez związku z konkretnymi adresami komórek — *przyp. tłum.*



RYSUNEK 1.5. Przykładowa struktura danych

Każda komórka łańcucha (w górnej części rysunku) jest rekordem o następującej definicji:

```
type
    recordtype = record
        cursor: integer;
        ptr: ↑recordtype
    end;
```

Pole *cursor* tego rekordu jest kursorem do jakiegoś elementu tablicy *reclist*, pole *ptr* zawiera natomiast wskaźnik do następnej komórki w łańcuchu. Wszystkie rekordy łańcucha są rekordami *anonimowymi*, nie mają nazw, gdyż każdy z nich utworzony został dynamicznie, w wyniku wywołania funkcji *new*. Pierwszy rekord łańcucha wskazywany jest natomiast przez zmienną *header*:

```
var
    header: ↑recordtype;
```

Pole *data* pierwszego rekordu w łańcuchu zawiera kursor do czwartego elementu tablicy *reclist*, pole *ptr* jest natomiast wskaźnikiem do drugiego rekordu. W drugim rekordzie pole *data* ma wartość 2, co oznacza kursor do drugiego elementu tablicy *reclist*; pole *ptr* jest natomiast *pustym* wskaźnikiem oznaczającym po prostu *brak* wskazania na cokolwiek. W języku Pascal „puste" wskazanie oznaczane jest słowem kluczowym **nil**. □

1.4. Czas wykonywania programu

Programista przystępujący do rozwiązywania jakiegoś problemu staje często przed wyborem jednego spośród wielu możliwych algorytmów. Jakimi kryteriami powinien się wówczas kierować? Otóż istnieją pod tym względem dwa, sprzeczne ze sobą, kryteria oceny:

(1) Najlepszy algorytm jest łatwy do zrozumienia, kodowania i weryfikacji.
(2) Najlepszy algorytm prowadzi do efektywnego wykorzystania zasobów komputera i jest jednocześnie tak szybki, jak to tylko możliwe.

Jeżeli tworzony program ma być uruchamiany tylko od czasu do czasu, wiążące będzie z pewnością pierwsze kryterium. Koszty związane z wytworzeniem programu są wówczas znacznie wyższe od kosztów wynikających z jego uruchamiania, należy więc dążyć do jak najefektywniejszego wykorzystania czasu programistów, bez szczególnej troski o obciążenie zasobów systemu. Jeżeli jednak program ma być uruchamiany często, koszty związane z jego wykonywaniem szybko się zwielokrotnią. Wówczas górę biorą względy natury efektywnościowej, gdzie liczy się szybki algorytm, bez względu na jego stopień komplikacji. Warto niekiedy wypróbować kilka różnych algorytmów i wybrać najbardziej opłacalny w konkretnych warunkach. W przypadku dużych złożonych systemów może okazać się także celowe przeprowadzanie pewnych symulacji badających zachowania konkretnych algorytmów. Wynika stąd, że programiści powinni nie tylko wykazać się umiejętnością optymalizowania programów, lecz także powinni umieć określić, czy w danej sytuacji zabiegi optymalizacyjne są w ogóle uzasadnione.

Pomiar czasu wykonywania programu

Czas wykonywania konkretnego programu zależy od szeregu czynników, w szczególności:

(1) danych wejściowych,
(2) jakości kodu wynikowego generowanego przez kompilator,
(3) architektury i szybkości komputera, na którym program jest wykonywany,
(4) złożoności czasowej algorytmu użytego do konstrukcji programu.

To, że czas wykonywania programu zależeć może od danych wejściowych, prowadzi do wniosku, iż czas ten powinien być możliwy do wyrażenia w postaci pewnej funkcji wybranego aspektu tych danych — owym „aspektem" jest najczęściej *rozmiar* danych. Znakomitym przykładem wpływu danych wejściowych na czas wykonania programu jest proces *sortowania* danych w rozmaitych odmianach, którymi zajmiemy się szczegółowo w rozdziale 8. Jak wiadomo, dane wejściowe programu sortującego mają postać listy elementów. Rezultatem wykonania programu jest lista złożona z tych samych elementów, lecz uporządkowana według określonego kryterium. Przykładowo, lista 2, 1, 3, 1, 5, 8 po uporządkowaniu w kolejności rosnącej będzie mieć postać 1, 1, 2, 3, 5, 8. Najbardziej intuicyjną miarą rozmiaru danych wejściowych jest liczba elementów w liście, czyli *długość* listy wejściowej. Kryterium długości listy wejściowej jako rozmiaru danych jest adekwatne w przypadku wielu algorytmów, dlatego w niniejszej książce będziemy je stosować domyślnie — poza sytuacjami, w których wyraźnie będziemy sygnalizować odstępstwo od tej zasady.

Przyjęło się oznaczać przez $T(n)$ czas wykonywania programu, gdy rozmiar danych wejściowych wynosi n. Przykładowo, niektóre programy wykonują się w czasie $T(n) = cn^2$, gdzie c jest pewną stałą. Nie precyzuje się jednostki, w której wyraża się wielkość $T(n)$, wygodnie jest przyjąć, że jest to liczba instrukcji wykonywanych przez hipotetyczny komputer.

Dla niektórych programów czas wykonania może jednak zależeć od szczególnej postaci danych, nie tylko od ich rozmiaru. W takiej sytuacji $T(n)$ oznacza *pesymistyczny* czas wykonania (tzw. najgorszy przypadek — ang. *worst case*), czyli maksymalny czas wykonania dla (statystycznie) wszystkich możliwych danych o rozmiarze n. Ponieważ najgorszy przypadek stanowi sytuację skrajną, definiuje się także *średni* czas wykonania oznaczany przez $T_{avg}(n)$ i stanowiący wynik (statystycznego) uśrednienia czasu wykonania wszystkich możliwych danych rozmiaru n. To, że $T_{avg}(n)$ stanowi miarę bardziej obiektywną niż czas pesymistyczny, staje się niekiedy źródłem błędnego założenia, że wszystkie możliwe postaci danych wejściowych są jednakowo prawdopodobne. W praktyce określenie średniego czasu wykonania bywa znacznie trudniejsze, niż określenie czasu pesymistycznego zarówno ze względu na trudności związane z „matematycznym podejściem" do problemu, jak i z powodu

niezbyt precyzyjnego znaczenia określenia „średni". Z tego właśnie względu przy szacowaniu złożoności czasowej algorytmów będziemy raczej bazować na czasie pesymistycznym, ale będziemy uwzględniać czas średni w sytuacjach, w których da się to uczynić.

Zatrzymajmy się teraz nad punktami 2. i 3. przedstawionej listy, zgodnie z którymi czas wykonania programu zależny jest zarówno od użytego kompilatora, jak i konkretnego komputera. Zależność ta uniemożliwia określenie czasu wykonania w sposób konwencjonalny, np. w sekundach, możemy to uczynić jedynie w kategoriach proporcjonalności, mówiąc na przykład, że „czas sortowania bąbelkowego proporcjonalny jest do n^2". Stała będąca współczynnikiem tej proporcjonalności pozostaje wielką niewiadomą, zależną i od kompilatora, i od komputera oraz kilku innych czynników.

Notacja „dużego O" i „dużej omegi"

Wygodnym środkiem wyrażania funkcyjnej złożoności czasowej algorytmu jest tzw. notacja „dużego O". Mówimy na przykład, że złożoność programu $T(n)$ jest rzędu „dużego O od n-kwadrat" i zapisujemy to w postaci $T(n) = O(n^2)$. Formalnie oznacza to, że istnieją takie stałe dodatnie c i n_0, że dla każdego $n \geq n_0$ zachodzi $T(n) \leq cn^2$.

Przykład 1.4. Załóżmy, że $T(0) = 1$, $T(1) = 4$ i ogólnie $T(n) = (n+1)^2$. Widzimy więc, że $T(n) = O(n^2)$ oraz $n_0 = 1$ i $c = 4$. Istotnie, dla $n \geq 1$ mamy $(n+1)^2 \leq 4n^2$, co Czytelnik może łatwo sprawdzić. Zauważ, że nie można przyjąć $n_0 = 0$, gdyż $T(0) = 1$ nie jest mniejsze od $c0^2 = 0$ dla żadnego c. □

Przyjmujemy, że funkcje złożoności czasowej określone są w dziedzinie *nieujemnych liczb całkowitych*, a ich wartości są także nieujemne, chociaż niekoniecznie całkowite. Mówimy, że $T(n) = O(f(n))$, jeżeli istnieją takie stałe c i n_0, że dla każdego $n \geq n_0$ zachodzi $T(n) \leq cf(n)$. O programie, którego złożoność czasowa jest $O(f(n))$ mówimy, że ma on tempo wzrostu $f(n)$.

Przykład 1.5. Funkcja $T(n) = 3n^2+2n^2$ jest $O(n^3)$. Istotnie, załóżmy $n_0 = 0$ i $c = 5$. Łatwo pokazać, że $3n^3+2n^2 \leq 5n^3$ dla każdego $n \geq 0$. Zauważ, że $O(n^4)$ byłoby dla tej funkcji również oszacowaniem poprawnym, jednak mniej precyzyjnym[5].

Udowodnimy, że funkcja 3^n *nie* jest $O(2^n)$. Załóżmy mianowicie, że istnieją takie stałe n_0 i c, że dla każdego $n \geq n_0$ zachodzi $3^n \leq c2^n$. Musiałoby wówczas zachodzić $c \geq (3/2)^n$ dla każdego $n \geq n_0$, gdy tymczasem wielkość $(3/2)^n$ rośnie nieograniczenie wraz ze wzrostem n, nie istnieje więc stała ograniczająca ją od góry. □

Stwierdzenie „$T(n)$ jest $O(f(n))$" albo „$T(n) = O(f(n))$" oznacza, że funkcja $f(n)$ stanowi górne ograniczenie tempa wzrostu $T(n)$. Na oznaczenie *dolnego* ograniczenia złożoności czasowej wprowadzono notację „dużej omegi"[6]. Mówimy, że „$T(n)$ jest omega od $g(n)$" i zapisujemy to $T(n) = \Omega(g(n))$, co formalnie oznacza, że istnieje taka stała dodatnia c, że dla nieskończenie wielu wartości n zachodzi $T(n) \geq cg(n)$.

---

[5] Kwestia ta została szczegółowo wyjaśniona na stronach 17 – 18 książki *Algorytmy i struktury danych z przykładami w Delphi*, wyd. Helion 2000 — *przyp. tłum.*

[6] Zwróć uwagę na asymetrię między definicjami „dużego O" i „dużej omegi" — w pierwszym przypadku mówi się o *wszystkich* $n \geq n_0$, w drugim o *nieskończenie wielu*. Asymetria ta bywa bardzo często użyteczna, ponieważ istnieją szybkie algorytmy, które jednak dla szczególnych wartości danych wejściowych stają się bardzo powolne. Przykładowo, algorytm sprawdzający, czy długość listy danych wejściowych wyraża się liczbą pierwszą, wykonuje się bardzo szybko, jeżeli długość ta jest liczbą parzystą; w tej sytuacji nie możemy podać dolnego ograniczenia obowiązującego dla wszystkich $n \geq n_0$, możemy jednak ustalić takie, które obowiązuje dla nieskończenie wielu z nich.

Przykład 1.6. Aby sprawdzić, czy funkcja $T(n) = n^3+2n^2$ jest $\Omega(n^3)$, załóżmy $c = 1$; wtedy $T(n) \geq cn^3$ dla $n = 0, 1, \ldots$

Załóżmy teraz, że $T(n) = n$ dla n nieparzystych i $T(n) = n^2/100$ dla n parzystych. Jeżeli przyjmiemy $c = 1/100$, to dla wszystkich parzystych n (czyli dla nieskończonej ich ilości) otrzymamy $T(n) = cn^2$, czyli $T(n) \leq cn^2$, więc $T(n)$ jest $\Omega(n^2)$. □

Tyraniczne tempo wzrostu

Przy porównywaniu złożoności czasowej dwóch programów ignoruje się zazwyczaj stałą proporcjonalności; przy tym założeniu program o złożoności $O(n^2)$ jest „lepszy" od rozwiązującego ten sam problem programu o złożoności $O(n^3)$. Jak wcześniej stwierdziliśmy, stałą proporcjonalności w funkcji złożoności czasowej odzwierciedla konsekwencje użycia określonego kompilatora i uruchomienia programu na komputerze o określonej szybkości. Nie jest to jednak do końca prawdą, ponieważ swój wkład do owej stałej proporcjonalności wnoszą też… same algorytmy. Załóżmy na przykład, że dwa różne programy rozwiązują ten sam problem rozmiaru[7] n w czasie (odpowiednio) $100n^2$ i $5n^3$ milisekund (na tym samym komputerze, przy użyciu tego samego kompilatora). Czy drugi z tych programów może być lepszy od pierwszego?

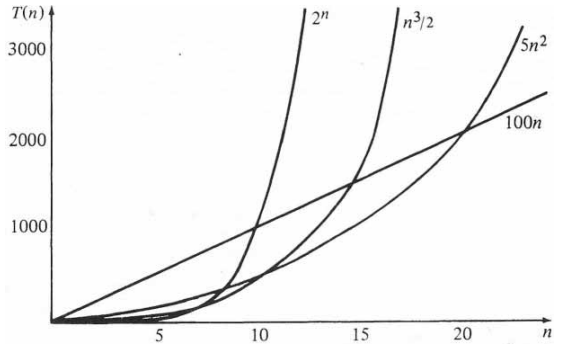
Odpowiedź na to pytanie zależy od *spodziewanego* rozmiaru danych, jakie przyjdzie przetwarzać obydwu programom. Nierówność $5n^3 < 100n^2$ spełniona jest dla $n < 20$, zatem dla danych niewielkich rozmiarów drugi program będzie zdecydowanie lepszy — zwłaszcza przy częstym uruchamianiu — mimo większego tempa wzrostu. Gdy jednak rozmiar danych zaczyna wzrastać, wykładniki potęg stają się coraz bardziej istotne — drugi program jest przecież $5n^3/100n^2 = n/20$ razy wolniejszy, czyli przewaga pierwszego programu pod względem szybkości obliczeń jest proporcjonalna do rozmiaru danych. Tłumaczy to preferowanie samego tempa wzrostu programu i przywiązywanie znacznie mniejszej wagi do stałych proporcjonalności.

Innym ważnym czynnikiem, który przemawia przynajmniej za poszukiwaniem algorytmów o jak najmniejszym tempie wzrostu, jest określenie, z jakiego rozmiaru danymi jest w stanie skutecznie poradzić sobie dany algorytm w danych warunkach, czyli na określonym komputerze przy użyciu określonego kompilatora. Nieustanny postęp technologiczny i zwiększające się wciąż szybkości komputerów są również przyczyną powierzania komputerom coraz bardziej złożonych zagadnień. Jak za chwilę zobaczymy, jedynie w przypadku algorytmów o niewielkiej złożoności w rodzaju $O(n)$ czy $O(n \log n)$ zwiększenie szybkości komputera ma znaczący wpływ na wzrost rozmiaru problemu możliwego do rozwiązania (w określonym czasie).

Przykład 1.7. Na rysunku 1.6 pokazano graficzne przedstawienie czasu (mierzonego w sekundach) wykonania programów o różnej złożoności czasowej (kompilowanych tym samym kompilatorem i uruchamianych na tym samym komputerze). Przypuśćmy, że mamy do dyspozycji 1000 sekund. Jak duży rozmiar danych zdolny jest w tym czasie przetworzyć każdy z programów?

Jak łatwo odczytać z przedstawionego wykresu, rozmiar problemów możliwych do rozwiązywania przez każdy z programów w czasie 1000 sekund jest mniej więcej taki sam. Załóżmy teraz, że bez żadnych dodatkowych kosztów udało nam się wygospodarować czas 10 razy większy lub, co na jedno wychodzi, otrzymaliśmy dziesięciokrotnie szybszy komputer. Jak w tej sytuacji zmieni się maksymalny rozmiar problemu możliwego do rozwiązywania?

[7] Określenie „problem rozmiaru n" jest tu wygodnym skrótem określającym „dane rozmiaru n przetwarzane przez program rozwiązujący problem" — *przyp. tłum.*



RYSUNEK 1.6. Złożoność czasowa czterech programów

Zestawienie widoczne w tabeli 1.3 nie pozostawia żadnych wątpliwości. Przewaga programu o złożoności $O(n)$ jest wyraźnie widoczna, jego możliwości wzrastają dziesięciokrotnie. Pozostałe programy prezentują się pod tym względem mniej imponująco, ponieważ wielkość problemów możliwych do rozwiązywania przez nie wzrasta tylko nieznacznie. Szczególnie program o złożoności $O(2^n)$ zdatny jest do rozwiązywania jedynie problemów o niewielkich rozmiarach.

TABELA 1.3.
Wzrost maksymalnego rozmiaru możliwego do rozwiązania problemu w rezultacie dziesięciokrotnego przyspieszenia komputera

Złożoność czasowa T(n)	Komputer oryginalny	Komputer dziesięciokrotnie szybszy	Względny przyrost rozmiaru problemu
$100n$	10	100	10,0
$5n^2$	14	45	3,2
$n^3/2$	12	27	2,3
2^n	10	13	1,3

Można w pewnym sensie odwrócić tę sytuację, porównując czas potrzebny na rozwiązanie (przez każdy z programów) problemu o rozmiarze dziesięciokrotnie większym (w porównaniu do tego, jaki rozwiązują one w czasie 1000 sekund).

Z tabeli 1.4 wynika niezbicie, że najszybsze nawet komputery nie są w stanie pomniejszyć znaczenia efektywnych algorytmów. □

TABELA 1.4.
Wzrost czasu wykonania programu odpowiadający dziesięciokrotnemu zwiększeniu rozmiaru problemu

Złożoność czasowa T(n)	Rozmiar problemu	Czas wykonania programu (s)	Względny przyrost czasu wykonania
$100n$	100	10 000	10
$5n^2$	140	100 000	100
$n^3/2$	120	1 000 000	1000
2^n	100	1030 (≈ 3∗1022 lat)	1029

Dochodzimy tym samym do nieco paradoksalnej konkluzji. Otóż w miarę, jak komputery stają się coraz szybsze, jednocześnie pojawiają się coraz bardziej złożone problemy (a raczej chęć wykorzystania komputerów do ich rozwiązania). To, w jakim stopniu wzrost mocy obliczeniowej

przekłada się na wzrost rozmiaru problemów możliwych do rozwiązania, zależy przede wszystkim od złożoności czasowej używanych algorytmów. Nieustający postęp technologiczny nie tylko więc nie zwalnia z poszukiwania efektywnych algorytmów, lecz nadaje mu coraz większe znaczenie! — nawet jeśli wydawałoby się, że powinno być odwrotnie.

Szczypta soli…

Mimo fundamentalnego znaczenia złożoności czasowej, istnieją przypadki, gdy nie jest ona jedynym ani też najważniejszym kryterium wyboru algorytmu. Oto kilka przykładowych sytuacji, w których główną rolę odgrywają inne czynniki.

(1) Jeżeli program ma być użyty tylko raz lub kilka razy bądź ma być wykorzystywany sporadycznie, koszty jego wytworzenia i przetestowania znacznie przewyższać będą koszt związany z jego uruchamianiem. O wiele ważniejsze od efektywności algorytmu jest wówczas jego poprawne zaimplementowanie w prosty sposób.
(2) Tempo wzrostu czasu wykonania programu, wyrażone przez notację „dużego O" algorytmu staje się czynnikiem krytycznym w przypadku *asymptotycznym*, czyli wtedy, gdy rozmiar problemu zaczyna zmierzać do nieskończoności. Dla małych rozmiarów istotne stają się współczynniki proporcjonalności (w funkcji złożoności czasowej) — duży współczynnik proporcjonalności może „zrekompensować" szybkość wynikającą z małego wykładnika potęgi. Istnieje wiele algorytmów, np. algorytm Schonhagego-Strassena mnożenia liczb całkowitych [1971], które pod względem złożoności asymptotycznej są zdecydowanie lepsze od konkurentów, lecz duże współczynniki proporcjonalności dyskwalifikują je pod względem przydatności do rozwiązywania problemów pojawiających się w praktyce.
(3) Skomplikowane algorytmy niełatwo zrozumieć, a tworzone na ich bazie programy są trudne do konserwacji i rozwijania dla osób, które nie są ich autorami. Jeżeli szczegóły danego algorytmu są powszechnie znane w danym środowisku programistów, można bez obaw algorytm ten wykorzystywać. W przeciwnym razie może się okazać, że opóźnienia i straty wynikłe z niezrozumienia lub, co gorsza, z błędnego zrozumienia algorytmu mogą zniweczyć wszelkie korzyści wynikające z jego efektywności.
(4) Znane są sytuacje, w których efektywne czasowo algorytmy wymagają znacznych ilości pamięci. Wiążąca się z tym konieczność wykorzystania pamięci zewnętrznych (np. do implementacji pamięci wirtualnej) może drastycznie obniżyć szybkość realizacji programu, niezależnie od efektywności samego algorytmu.
(5) W algorytmach numerycznych względy dokładności i stabilności są zwykle nieporównywalnie ważniejsze od szybkości obliczeń.

1.5. Obliczanie czasu wykonywania programu

Określenie czasu wykonania danego programu, nawet tylko z dokładnością do stałego czynnika, może stanowić skomplikowany problem matematyczny. Na szczęście, w większości przypadków spotykanych w praktyce wystarczające okazuje się zastosowanie kilku niezbyt skomplikowanych reguł. Zanim przejdziemy do ich omówienia, zatrzymajmy się na chwilę na zagadnieniu pomocniczym — dodawaniu i mnożeniu na gruncie notacji „dużego O".

Załóżmy, że $T_1(n) = O(f(n))$ i $T_2(n) = O(g(n))$ są czasami wykonania dwóch fragmentów programu, odpowiednio, P_1 i P_2. Łączny czas wykonania obydwu fragmentów, kolejno P_1 i P_2, wynosi wówczas $T_1(n)+T_2(n) = O(\max(f(n), g(n)))$. Aby przekonać się, że jest tak istotnie, zdefiniujmy cztery stałe: c_1, c_2, n_1, i n_2 takie, że dla $n \geq n_1$ zachodzi $T_1(n) \leq c_1 f(n)$ i, odpowiednio, dla $n \geq n_2$ zachodzi $T_2(n) \leq c_2 f(n)$. Niech $n_0 = \max(n_1, n_2)$. Jeżeli $n \geq n_0$, wtedy $T_1(n)+T_2(n) \leq c_1 f(n)+c_2 g(n) \leq (c_1+c_2)\times \max(f(n), g(n))$, czyli dla każdego $n \geq n_0$ zachodzi $T_1(n)+T_2(n) \leq c_0 \times \max(f(n), g(n))$, gdzie $c_0 = c_1+c_2$. Oznacza to (bezpośrednio na podstawie definicji), że $T_1(n)+T_2(n) = O(\max(f(n), g(n)))$.

Przykład 1.8. Opisaną powyżej *regułę sum* wykorzystać można do obliczenia złożoności czasowej sekwencji fragmentów programu, zawierającego pętle i rozgałęzienia. Załóżmy mianowicie, że czasy wykonania trzech fragmentów wynoszą, odpowiednio, $O(n^2)$, $O(n^3)$ i $O(n \log n)$. Łączny czas wykonania dwóch pierwszych kroków wyniesie wówczas $O(\max(n^2, n^3)) = O(n^3)$, zaś łączny czas wykonania wszystkich trzech — $O(\max(n^3, n \log n)) = O(n^3)$. □

W ogólnym przypadku czas wykonania ustalonej sekwencji kroków równy jest, z dokładnością do stałego czynnika, czasowi kroku wykonywanego najdłużej. Niekiedy zdarza się, że dwa kroki programu (lub większa ich liczba) są w stosunku do siebie *niewspółmierne* pod względem czasu wykonania. Czas ten nie jest ani równy, ani też systematycznie większy w przypadku któregoś z kroków. Dla przykładu rozpatrzmy dwa kroki o czasie wykonania, odpowiednio, $f(n)$ i $g(n)$, gdzie:

$$f(n) = \begin{cases} n^4 & \text{dla } n \text{ parzystych} \\ n^2 & \text{dla } n \text{ nieparzystych} \end{cases} \qquad f(n) = \begin{cases} n^2 & \text{dla } n \text{ parzystych} \\ n^3 & \text{dla } n \text{ nieparzystych} \end{cases}$$

Regułę sum należy zastosować w sposób bezpośredni. Łączny czas wykonania wspomnianych kroków równy będzie $O(\max(n^4, n^2)) = O(n^4)$ dla n parzystych oraz $O(\max(n^2, n^3) = O(n^3)$ dla n nieparzystych.

Innym pożytecznym wnioskiem z reguły sum jest to, że w sytuacji, w której $g(n) \leq f(n)$ dla wszystkich $n \geq n_0$, $O(f(n)+g(n))$ jest tym samym, co $O(f(n))$. Na przykład $O(n^2+n)$ oznacza to samo, co $O(n^2)$.

W podobny sposób formułuje się *regułę iloczynów*. Jeżeli $T_1(n) = O(f(n))$ i $T_2(n) = O(g(n))$, to $T_1(n)\ T_2(n)$ równe jest $O(f(n)g(n))$, co dociekliwy Czytelnik zweryfikować może samodzielnie w taki sam sposób, jak zrobiliśmy to w przypadku reguły sum. W szczególności $O(cf(n))$ jest tym samym, co $O(f(n))$ (c jest tu dowolną stałą dodatnią).

Przed przystąpieniem do omawiania ogólnych reguł analizy czasu wykonywania programów, przyjrzyjmy się prostemu przykładowi ilustrującemu wybrane elementy rzeczywistego procesu.

Przykład 1.9. Na listingu 1.4 widoczna jest procedura *bubble* sortująca tablicę liczb rzeczywistych w kolejności rosnącej, metodą *sortowania bąbelkowego* (ang. *bubblesort*). Każde wykonanie pętli wewnętrznej powoduje przesunięcie najmniejszych elementów w kierunku początku tablicy.

LISTING 1.4.
Procedura sortowania bąbelkowego

```
    procedure bubble( var A: array [1..n] of integer );
        { bubble sortuje tablicę A w kolejności rosnącej }
    var
        i, j, temp: integer;
        begin
{1}         for i := 1 to n-1 do
{2}             for j := n downto j+1 do
{3}                 if A[j-1] > A[j] then begin
```

```
                        {zamień miejscami A[j-1] z A[j]}
                        temp := A[j-1];
{4}                     A[j-1] := A[j];
{5}                     A[j] := temp;
{6}                 end;
        end; {bubble}
```

Rozmiar sortowanej tablicy, czyli liczba elementów do posortowania, jest tu niewątpliwie wiarygodną miarą rozmiaru problemu. Oczywiście, każda instrukcja przypisania wymaga pewnego stałego czasu, niezależnego od danych wejściowych; każda z instrukcji {4}, {5} i {6} wykonuje się więc w czasie $O(1)$, czyli $O(0)$ ($O(c) = O(0)$ dla dowolnej stałej c). Łączny czas wykonania tych instrukcji równy jest — na mocy reguły sum — $O(\max(1, 1, 1)) = O(1) = O(0)$.

Przyjrzyjmy się teraz instrukcjom pętli i instrukcji warunkowej. Instrukcje te są zagnieżdżone, musimy więc zacząć analizę od najbardziej wewnętrznej z nich. Testowanie warunku instrukcji **if** wymaga $O(1)$ czasu. Jeżeli chodzi o ciało tej instrukcji, czyli instrukcje uwarunkowane {4}, {5} i {6}, nie sposób z góry przewidzieć, ile razy zostaną wykonane. Założymy więc pesymistycznie, że będą wykonywane *zawsze*, w czasie $O(1)$. Zatem koszt wykonania całej instrukcji **if** wynosi $O(1)$.

Przemieszczając się na zewnątrz, natrafiamy na wewnętrzną pętlę **for** (wiersze {2} – {6}). Ogólnie rzecz ujmując, czas wykonania pętli for jest sumą czasu wykonania poszczególnych jej „obrotów", przy czym musimy jeszcze dodać $O(1)$ czasu w każdym obrocie, potrzebnego na zwiększenie i testowanie zmiennej sterującej oraz skok do początku pętli. Każdy obrót wspomnianej pętli wymaga więc $O(1)$ czasu, liczba obrotów równa jest natomiast $O(n–i)$ (i jest wartością zmiennej sterującej pętli zewnętrznej). Daje to łączny czas wykonania pętli $O((n–i)\times 1) = O(n–i)$.

Dochodzimy wreszcie do pętli zewnętrznej (wiersze {1} – {6}). Wykonuje się ona n–1 razy, tak więc łączny czas jej wykonania wynosi $O(k)$, gdzie k równe jest:

$$\sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = n^2/2 - n/2$$

Procedura *bubble* wykonuje się w czasie $O(n^2)$. W rozdziale 8. przedstawimy inne algorytmy sortowania, działające w czasie $O(n \log n)$, a zatem efektywniejsze[8], gdyż $\log n$ jest mniejsze od n. □

Zanim przejdziemy do omawiania ogólnych reguł analizowania złożoności obliczeniowej, przypomnijmy, że precyzyjne określenie górnego ograniczenia tej złożoności, choć czasami bardzo łatwe, to w wielu wypadkach może stanowić prawdziwe wyzwanie intelektualne.

Nie sposób zresztą podać jakiegoś kompletnego zbioru reguł tego procesu. Ograniczymy się tylko do wymienienia pewnych wskazówek i koncepcji ilustrowanych stosownymi przykładami. Czas wykonania określonej instrukcji lub grupy instrukcji uzależniony jest od rozmiaru danych wejściowych i opcjonalnie od wartości jednej zmiennej lub wielu zmiennych. Natomiast jedynym dopuszczalnym parametrem wpływającym na czas wykonania *całego programu* jest rozmiar danych wejściowych.

(1) Czas wykonania instrukcji przypisania, wczytywania lub wypisywania danych przyjmuje się jako $O(1)$. Od tej zasady istnieje kilka wyjątków. Po pierwsze, niektóre języki, w tym Pascal, dopuszczają kopiowanie całych tablic (być może dość dużych) za pomocą pojedynczej instrukcji przypisania. Po drugie, prawa strona instrukcji przypisania zawierać może wywołanie funkcji, której czasu wykonania nie można pominąć.

[8] Wszystkie używane w tej książce logarytmy mają podstawę 2, chyba że wyraźnie zaznaczona została inna. Dla notacji „dużego o" podstawa logarytmu i tak nie ma znaczenia, ponieważ zmiana podstawy logarytmu równoznaczna jest z pomnożeniem jego wartości przez stały czynnik: $\log_a n = c\log_b n$, gdzie $c = \log_a b$.

(2) Czas wykonania sekwencji instrukcji określony jest przez regułę sum z dokładnością do stałego czynnika. Jest on równy czasowi wykonania tej instrukcji, która wykonuje się najdłużej.
(3) Na czas wykonania instrukcji **if** składa się czas wykonania instrukcji uwarunkowanej i czas wartościowania wyrażenia warunkowego — ten ostatni przyjmuje się zwykle jako $O(1)$. Czas wykonania instrukcji **if-then-else** oblicza się natomiast jako sumę czasu wartościowania wyrażenia warunkowego oraz czasu wykonania tej spośród sekcji uwarunkowanych, która wykonuje się dłużej.
(4) Czas wykonania pętli jest sumą wykonania wszystkich jej obrotów. Na czas wykonania jednego obrotu składa się czas wykonania ciała pętli oraz czas czynności pomocniczych związanych z obsługą zmiennej sterującej i skokiem do początku pętli (zakłada się, że owe czynności pomocnicze wykonywane są w czasie $O(1)$). Często czas wykonania pętli oblicza się, mnożąc liczbę wykonywanych obrotów przez największy możliwy czas wykonania pojedynczego obrotu, ale czasami nie można z góry ustalić liczby wykonywanych obrotów, zwłaszcza w przypadku pętli *nieskończonej*.

Wywołania procedur

Jeżeli mamy do czynienia z programem, w którym żadna z procedur nie jest rekurencyjna[9], możemy rozpocząć analizowanie czasu wykonania od tych procedur, które nie wywołują żadnych innych procedur (oczywiście, wywołanie funkcji w wyrażeniu również uważane jest tu za „wywołanie procedury"), a co najmniej jedna taka procedura musi istnieć, skoro wszystkie są nierekurencyjne[10].

[9] Tzn. nie odwołuje się do samej siebie ani bezpośrednio, ani pośrednio.

[10] Warunek nierekurencyjności procedur jest warunkiem za słabym, by opisane postępowanie mogło się udać, czego przykładem jest chociażby sekwencja:

```
var
    ster: boolean;
...
procedure A;
begin
    ...
    if ster
    then
        B;
end;

procedure B;
begin
    ...
    if not ster
    then
        A;
end;

begin
    ...
    if ster then A else B;
end.
```

Wbrew pozorom nie mamy tutaj do czynienia z rekurencją, lecz nie ma procedury, która „nie wywoływałaby żadnych innych". Autorzy pisząc o rekurencji mieli zapewne na myśli także jej pozory, jak powyższy przykład — *przyp. tłum.*

W kolejnych krokach analizujemy te procedury, które odwołują się wyłącznie do procedur o obliczonej już złożoności, i proces ten kontynuujemy aż do przeanalizowania wszystkich procedur.

Kiedy niektóre (lub wszystkie) procedury programu są rekurencyjne, nie jest możliwe ustawienie ich w opisanym powyżej porządku. Należy zatem uznać złożoność czasową $T(n)$ danej procedury za niewiadomą funkcję zmiennej n, po czym sformułować i rozwiązać równanie rekurencyjne określające tę funkcję. Istnieje wiele sposobów analizowania różnego rodzaju zależności rekurencyjnych. Niektórymi z nich zajmiemy się w rozdziale 9., natomiast w tym miejscu zaprezentujemy jedynie obliczanie złożoności prostego programu rekurencyjnego.

Przykład 1.10. Widoczna na listingu 1.5 funkcja *fact* dokonuje rekursywnego obliczania funkcji $f(n) = n!$ ($n!$ to iloczyn kolejnych liczb naturalnych od 1 do n).

LISTING 1.5.
Rekurencyjne obliczanie funkcji silnia

```
    function fact( n: integer) : integer;
        { fact(n) równe jest n! }
        begin
{1}         if n <= 1 then
{2}             fact := 1
            else
{3}             fact := n * fact(n-1)
        end; {fact}
```

Miarą rozmiaru problemu dla tej funkcji jest oczywiście jej argument — oznaczmy zatem przez $T(n)$ czas wykonywania się tej funkcji dla argumentu n. Instrukcje w wierszach {1} i {2} wykonują się oczywiście w czasie $O(1)$, zaś instrukcja w wierszu {3} w czasie $O(1)+T(n–1)$. Dla pewnych stałych c i d mamy więc:

$$T(n) = \begin{cases} c + T(n-1) & \text{dla } n > 1 \\ d & \text{dla } n \le 1 \end{cases} \tag{1.1}$$

Zakładając, że $n > 2$, możemy rozwinąć wzór (1.1) do postaci:

$$T(n) = 2c+T(n–2) \quad (\text{dla } n > 2)$$

ponieważ $T(n–1) = c+T(n–2)$, o czym można się przekonać, zmieniając we wzorze (1.1) n na $n–1$. Korzystając z kolei z zależności:

$$T(n–2) = c+T(n–3) \quad (\text{dla } n > 3)$$

możemy napisać:

$$T(n) = 3c+T(n–3) \quad (\text{dla } n > 3)$$

i ogólnie:

$$T(n) = ic+T(n–i) \quad (\text{dla } n > i)$$

Ostatecznie, dla $i = n–1$ otrzymamy:

$$T(n) = c(n–1)+T(1) = c(n–1)+d \tag{1.2}$$

Zatem, zgodnie ze wzorem (1.2), $T(n) = O(n)$. Zauważ, że dla celów naszej analizy przyjęliśmy, że mnożenie dwóch liczb całkowitych realizuje się w czasie $O(1)$. Założenie to może nie być słuszne, jeżeli n będzie tak duże, że będziemy musieli używać *wielosłowowej* reprezentacji liczb całkowitych. □

Zaprezentowana tutaj ogólna metoda rozwiązywania równań rekurencyjnych polega na sukcesywnym zastępowaniu po prawej stronie równania wartości $T(k)$ wartościami $T(k–1)$ tak długo, aż dla jakiegoś argumentu x wartość $T(x)$ wyrażona zostanie nierekurencyjnie. Jeżeli nie będzie możliwe dokładne oszacowanie poszczególnych wartości $T(k)$, możemy posłużyć się ich *górnymi ograniczeniami*, a otrzymany wynik też będzie wówczas górnym ograniczeniem na $T(n)$.

Programy z instrukcjami GOTO

Omawiane dotychczas metody analizy złożoności czasowej przeznaczone są zasadniczo dla programów, których struktura opiera się wyłącznie na pętlach i rozgałęzieniach, ponieważ za pomocą reguły sum i reguły iloczynów można łatwo analizować ich sekwencje i zagnieżdżenia. Tę wygodną regularność zepsuć mogą (i zazwyczaj psują) instrukcje skoku (**goto**) znacznie utrudniające wszelką analizę kodu. Stanowi to argument przemawiający za unikaniem instrukcji **goto**, a przynajmniej za ich umiarkowanym używaniem. W wielu przypadkach okazują się one niezbędne, na przykład wtedy, gdy trzeba przedwcześnie zakończyć pętlę **while**, **for** i **repeat**. W języku Pascal bowiem, w przeciwieństwie do języka C, nie ma instrukcji `break` i `continue`[11].

Sugerujemy następujące podejście do analizowania instrukcji **goto** realizujących „wyskok" z pętli i prowadzących *bezpośrednio* za pętlę — to bodaj jedyny usprawiedliwiony sposób ich użycia. Z reguły instrukcja **goto** w pętli wykonywana jest *w sposób warunkowy*, dlatego możemy założyć, że odnośny warunek nigdy nie wystąpi i pętla wykona się w sposób kompletny. Jako że skok wywołany przez instrukcję **goto** prowadzi do instrukcji występującej bezpośrednio po pętli — czyli tej instrukcji, która wykonuje się jako pierwsza po „normalnym" wykonaniu pętli — założenie takie nigdy nie prowadzi do zaniżenia pesymistycznej złożoności czasowej. Niebezpieczeństwo takie pojawiłoby się jednak, gdyby instrukcja **goto** była faktycznie zakamuflowaną instrukcją pętli. Bywają natomiast (rzadkie co prawda) sytuacje, w których takie „zachowawcze" ignorowanie instrukcji **goto** prowadzi do zawyżenia złożoności pesymistycznej.

Nie chcielibyśmy stwarzać wrażenia, że instrukcja **goto** prowadząca *wstecz* sama z siebie czyni kod programu niezdatnym do analizy. Tak długo, jak „zapętlenia" powodowane przez instrukcje skoku mają „rozsądną" strukturę — to znaczy są zupełnie rozłączne bądź zagnieżdżone w sobie — opisane w tym rozdziale techniki analizy można z powodzeniem stosować (chociaż osoba dokonująca analizy musi upewnić się, czy owe „zapętlenia" nie kryją w sobie jakichś niespodzianek). Uwaga ta odnosi się szczególnie do języków pozbawionych pętli warunkowych, jak np. Fortran.

[11] Uwaga ta dotyczy wzorcowego języka Pascal. W 1992 roku ukazała się wersja 7. Turbo Pascala, zawierająca wspomniane instrukcje. Niezależnie jednak od tego istnieją sytuacje, w których „wyskok" z głęboko zagnieżdżonej pętli za pomocą instrukcji **goto** jest znacznie zgrabniejszy i czytelniejszy, niż mozolne „przedzieranie się" przez kilka poziomów wyrażeń warunkowych. Czytelników zainteresowanych problematyką rozsądnego używania instrukcji goto w Pascalu zachęcam do przeczytania 2. rozdziału książki *Delphi 7 dla każdego*, wyd. Helion 2003 — *przyp. tłum.*

Analizowanie pseudoprogramów

Techniki stosowane do analizy złożoności czasowej „prawdziwych" programów dają się także zastosować do podobnej analizy w stosunku do pseudoprogramów zawierających nieformalne opisy w języku naturalnym — oczywiście pod warunkiem, że zna się złożoność czasową czynności opisywanych przez te nieformalne „instrukcje". Kłopot polega na tym, że jest ona na ogół silnie uzależniona od implementacji, więc w przypadku procedur zaimplementowanych połowicznie lub niezaimplementowanych w ogóle, o ich złożoności czasowej możemy powiedzieć niewiele lub zgoła nic. Paradoksalnie nieznajomość ta kryje w sobie pewną zaletę. Decydując się mianowicie na użycie w programie pewnego abstrakcyjnego typu danych, nie determinujemy jeszcze złożoności czasowej tego programu — ta bowiem zależna będzie od (dokonanego później) wyboru konkretnej implementacji procedur związanych z ADT. Ta właśnie możliwość wyboru jest jednym z najważniejszych argumentów przemawiających za używaniem abstrakcyjnych typów danych.

Skoro czasy wykonania pewnych procedur wywoływanych w programie są jak na razie nieznane, sama złożoność czasowa tego programu może być traktowana jedynie jako funkcja, której parametrami są owe czasy wykonania, uzależnione z kolei od „rozmiarów" argumentów przekazywanych wspomnianym procedurom. Znaczenie słowa „rozmiar" jest zawsze sprawą konkretnego przypadku i interpretacji osoby dokonującej analizy, chociaż wybrany model matematyczny zazwyczaj wskazuje najbardziej trafne podejście w tym względzie. Jeżeli na przykład argument przekazany do procedury jest zbiorem danych (SET), najbardziej odpowiednią miarą jego „rozmiaru" jest liczba zawartych w nim elementów. W kolejnych rozdziałach książki przedstawimy wiele podobnych przykładów związanych z analizą rozmaitych pseudoprogramów.

1.6. Dobre praktyki programowania

Istnieje kilka dobrych i sprawdzonych praktyk, których warto przestrzegać przy projektowaniu algorytmów i ich implementowaniu w postaci programów. Niestety, w powszechnym odczuciu są one traktowane raczej jako banalne, a o ich prawdziwości można się dopiero przekonać w przypadku skutecznego rozwiązania jakiegoś rzeczywistego problemu, nie zaś drogą dociekań teoretycznych. Zamieszczamy je tutaj w głębokim przekonaniu, że zasługują na poważne potraktowanie, a Czytelnicy niniejszej książki bez trudu odnajdą ich odzwierciedlenie w prezentowanych przez nas programach. Mamy nadzieję, że także w swej codziennej praktyce programistycznej poszukiwać będą okazji do ich zastosowania.

(1) *Sporządź projekt programu*. Jak wspominaliśmy na początku rozdziału, każdy program może być początkowo zaprojektowany w postaci szkicu, za pomocą nieformalnych opisów wykonywanych czynności, by później, w wyniku procesu stopniowego precyzowania, doprowadzony został do formalnej postaci zdatnej do wykonania na komputerze (dokładniej, do przetłumaczenia przez kompilator). Strategia „od szkicu do szczegółów" prowadzi zazwyczaj do powstania programu dobrze zorganizowanego, łatwego do zrozumienia, testowania i konserwacji.

(2) *Zastosuj enkapsulację danych i kodu*. Abstrakcyjne typy danych i procedury wymuszają precyzyjne zlokalizowanie (w konkretnych miejscach programu) definicji określonych struktur danych i fragmentów kodu związanych z określonymi czynnościami. Jeżeli konieczne będzie dokonanie zmian w programie, łatwo będzie wskazać w kodzie odpowiednie miejsca do modyfikacji.

(3) *Wykorzystaj istniejące programy.* Jedna z głównych przyczyn nieefektywnego tworzenia programów wynika stąd, że tworzenie to często przypomina „powtórne odkrywanie Ameryki". Wykorzystanie istniejących programów i procedur (często już sprawdzonych i przetestowanych)

jest postępowaniem znacznie efektywniejszym, niż rozpoczynanie wszystkich działań od początku. Zasada ta działa także w drugą stronę, tworząc jakiś program, weź pod uwagę, że być może ktoś zechce go wykorzystać w przyszłości do swych potrzeb.

(4) *Bądź kowalem swych narzędzi*. W języku programistów narzędziami nazywane są programy przeznaczone do wielokrotnego użytku, w różnych zastosowaniach. Pisząc program, zastanów się, czy nie należałoby nadać mu bardziej ogólnej postaci i tym samym skonstruować narzędzie przydatne do różnych celów. Przykładowo, tworząc program układający harmonogram egzaminów, zastanów się, czy nie warto stworzyć ogólniejszego programu rozwiązującego problem kolorowania grafu przy użyciu najmniejszej liczby kolorów. Znalezienie optymalnego harmonogramu sesji mogłoby być wówczas tylko jednym z jego zastosowań, gdyż wierzchołki grafu reprezentowałyby grupy studentów, różne kolory odzwierciedlałyby różne terminy egzaminów, zaś połączenie krawędzią dwóch wierzchołków oznaczałoby, że w grupach reprezentowanych przez te wierzchołki znajdują się „wspólni" studenci. Zastosowanie uniwersalnego programu rozwiązującego problem optymalnego kolorowania grafu wykraczałoby natomiast daleko poza problem organizacji sesji egzaminacyjnej, czego przykład pokazaliśmy na początku rozdziału, poszukując optymalnego sposobu sterowania ruchem na skrzyżowaniu.

(5) *Wykorzystaj polecenia i programy systemu operacyjnego*. Polecenia i programy systemów operacyjnych zdolne są wykonać wiele pożytecznych zadań, działając jednak „w pojedynkę", ograniczają swe działanie do funkcji raczej elementarnych (do których w końcu zostały stworzone). Dobrze zaprojektowany system operacyjny udostępnia mechanizmy automatyzujące współpracę poszczególnych programów w taki sposób, by wyniki produkowane przez jeden z nich stawały się danymi wejściowymi drugiego. Zespół tak „połączonych" programów nazywamy *potokiem*, zaś poszczególne programy noszą nazwę *filtrów*. Jeżeli repertuar poleceń i programów danego systemu jest inteligentnie zaprojektowany, można w nim tworzyć potoki o zdumiewających możliwościach.

Przykład 1.11. Przykładem ciekawego potoku może być „program" `spell`, stworzony pierwotnie przez S.C. Johnsona wyłącznie z poleceń systemu UNIX[12]. Program ten wczytuje z pliku tekst w języku angielskim i wyprowadza wszystkie występujące w nim wyrazy, których brakuje w dołączonym niewielkim słowniku. Program `spell` przejawia co prawda tendencję do produkowania „fałszywych alarmów" w postaci sygnalizowania poprawnych słów jako słów błędnych[13], lecz generowane przez niego raporty są na tyle krótkie, że przy odrobinie inteligencji łatwo zorientować się, które z wydrukowanych słów są rzeczywiście błędne (treść niniejszej książki została zweryfikowana za pomocą programu *spell*).

Pierwszym filtrem użytym w potoku (którym w końcu jest program `spell`) jest polecenie `translate`, które, dzięki użyciu odpowiednich parametrów, służy do zamiany wielkich liter na ich małe odpowiedniki oraz do zastępowania tzw. „białych" znaków (*blanks*, czyli spacji i znaków tabulacji) znakami nowego wiersza. W rezultacie otrzymujemy wykaz słów występujących w tekście, a wszystkie z nich pisane są wyłącznie małymi literami i każde znajduje się w osobnym wierszu. Kolejne polecenie-filtr, `sort`, dokonuje posortowania tego wykazu w porządku alfabetycznym. Każde ze słów może występować w tekście wiele razy, dlatego we wspomnianym wykazie mogą się one powtarzać. Aby każde z nich występowało w wykazie tylko raz, wywoływane jest polecenie `unique` dokonujące eliminacji duplikatów w posortowanej liście. Tak „znormalizowany" wykaz jest następnie konfrontowany ze słownikiem w celu wychwycenia tych jego pozycji, które nie występują

[12] UNIX jest znakiem towarowym firmy Bell Laboratories.

[13] Im większy słownik, tym mniej „fałszywych alarmów". Można by użyć kompletnego słownika języka angielskiego, lecz nawet dla małych słowników większość raportowanych przez program wyrazów to rezultaty błędów ortograficznych lub tzw. literówek, a wyrazy te stwarzają niekiedy pozory przynależności do języka angielskiego, ale tak naprawdę nikt o nich nigdy nie słyszał.

w słowniku — to zadanie wykonuje polecenie *diff*, którego parametrem jest nazwa wspomnianego słownika. Oto więc kompletna treść programu-potoku *spell*:

```
spell: translate [A-Z] → [a-z]  blank → newline
       sort
       unique
       diff dictionary
```

Programowanie na poziomie potoków wymaga od programistów dyscypliny, a raczej wsparcia polegającego na tym, by jak najwięcej tworzonych programów mogło w razie potrzeby pełnić rolę filtrów. W ostatecznym rozrachunku, mierzonym stosunkiem osiągniętych efektów do włożonego wysiłku, postępowanie takie na dłuższą metę naprawdę się opłaca. □

1.7. Super Pascal

Większość prezentowanych w tej książce programów zapisanych zostało w języku Pascal. Aby uczynić je bardziej czytelnymi, użyliśmy trzech konstrukcji, które nie występują w standardowym Pascalu, łatwo je jednak mechanicznie przetłumaczyć na tę jego wersję. Pierwszą z tych konstrukcji są nienumeryczne etykiety — instrukcja **goto** *output* jest zdecydowanie bardziej intuicyjna niż **goto** 561. Każdą z takich instrukcji łatwo doprowadzić do „standardowej" postaci, zastępując każdą nienumeryczną etykietę jej unikalnym numerycznym odpowiednikiem[14] zarówno w deklaracji, jak i w miejscu wystąpienia i we wszystkich odwołujących się do niej instrukcjach **goto**.

Druga niestandardowa konstrukcja ma postać instrukcji **return** powodującej natychmiastowe zakończenie bieżąco wykonywanej procedury lub funkcji. W przypadku funkcji, parametr instrukcji specyfikuje zwracana wartość:

```
return(wyrażenie)
```

W standardowym Pascalu instrukcja **return** może być symulowana za pomocą skoku do etykiety znajdującej się bezpośrednio przed końcową dyrektywą **end**:

```
function Fibon(n:integer):integer;
    label
        999;
    begin
        if n <=0 then
        begin
            Fibon := 0;
            goto 999;
        end;

        if n < 3 then
        begin
            Fibon := 1;
            goto 999;
```

[14] Znakomita większość używanych obecnie implementacji języka Pascal — w tym popularne Turbo Pascal i Delphi — dopuszcza używanie etykiet nienumerycznych — *przyp. tłum.*

```
        end;

        Fibon := Fibon(n-1) + Fibon(n-2);

    999:
    end;
```

Poniższa postać funkcji jest jednak bardziej zwarta i czytelna[15]:

```
function Fibon(n:integer):integer;
    begin
        if n <=0 then
            return(0);

        if n < 3 then
            return(1);

        Fibon := Fibon(n-1) + Fibon(n-2);
        { albo: return(Fibon(n-1) + Fibon(n-2)) }
    end;
```

Przykład 1.12. Na listingu 1.6 widoczna jest funkcja *fact* zapisana z użyciem instrukcji **return**, na listingu 1.7 zawarta została natomiast jej postać dostosowana do wymogów standardowego Pascala. □

LISTING 1.6.
Funkcja fact korzystająca z instrukcji return…

```
function fact( n: integer ) : integer;
    begin
        if n <= 1 then
            return(1)
        else
            return(n * fact(n-1))
    end; {fact}
```

[15] W Turbo Pascalu instrukcja **return**(*expr*) może być symulowana przez sekwencję:

```
begin
    <nazwa funkcji> := expr;
    exit;
end;
```

W Delphi można to osiągnąć jeszcze bardziej uniwersalnie:

```
begin
    Result := expr;
    exit;
end;
```

— *przyp. tłum.*

LISTING 1.7.
…i po transformacji do postaci wymaganej przez standardowy Pascal

```
function fact( n: integer ) : integer;
    label
        999;
    begin
        if n <= 1 then
            begin
                fact := 1;
                goto 999;
            end
        else
            begin
                fact := n * fact(n-1);
                goto 999;
         end;
    999:
    end; {fact}
```

Trzecie z zastosowanych przez nas rozszerzeń wynika stąd, że język Pascal w pewnych sytuacjach wymaga użycia *identyfikatora* typu, nie tolerując jego definicji *in extenso*. Na przykład deklaracja:

```
function zap ( A: array[1..10] of integer ): ↑celltype;
```

musi być zapisana w następującej postaci:

```
type
    arrayoftenints = array[1..10] of integer;
    ptrtocell = ↑celltype;

function zap ( A: arrayoftenints ): ptrtocell;
```

by kompilator uznał ją za poprawną. Pierwotna postać, jakkolwiek niepoprawna syntaktycznie (choć przecież równoważna poprawnej postaci), jest jednak bardziej zwięzła i zrozumiała intuicyjnie, dlatego będziemy ją konsekwentnie stosować w prezentowanych programach, a jej ewentualna transformacja do wspomnianej postaci będzie dla Czytelnika zadaniem czysto mechanicznym.

Na koniec krótka uwaga na temat zastosowanej przez nas konwencji typograficznej w listingach prezentujących przykładowe programy. Zarezerwowane słowa języka Pascal wypisywane będą czcionką **pogrubioną**, nazwy typów czcionką prostą, zaś nazwy zmiennych, procedur i funkcji czcionką *pochyłą*. Rozróżniać będziemy także wielkie i małe litery[16].

[16] Rozróżnianie między wielkościami liter to w rzeczywistości czwarte z odstępstw od standardowego Pascala, bowiem żadna ze znanych implementacji tego języka nie odróżnia małych i wielkich liter w zapisie identyfikatorów i słów kluczowych. Czytelnik dokonujący konwersji prezentowanych programów na którąś z rzeczywistych implementacji Pascala musi więc uważać, by nie utożsamić ze sobą różnych (biorąc pod uwagę zastosowaną konwencję) identyfikatorów — *przyp. tłum.*

Ćwiczenia

1.1. Sześć drużyn: Szakale, Lwy, Orły, Bobry, Tygrysy i Skunksy, przygotowuje się do kolejnych rozgrywek ligi piłkarskiej. Szakale rozegrały już mecze z Lwami i Orłami, Lwy także grały już z Bobrami i Skunksami, zaś Tygrysy grały z Orłami i Skunksami. Każda drużyna rozgrywa tylko jeden mecz w tygodniu. Znajdź taki harmonogram rozgrywek, by każda drużyna grała z każdą i aby zajęło to jak najmniej czasu. *Wskazówka*. Stwórz graf, którego wierzchołki prezentować będą te pary, które jeszcze nie rozegrały meczu ze sobą. Jeżeli poszczególne kolory reprezentować będą kolejne tygodnie rozgrywek, jakie będzie znaczenie *krawędzi* w tym grafie?

***1.2.** Rozpatrz ramię robota zakotwiczone na jednym końcu. Ramię to ma dwa przeguby, z których każdy pozwala na obrót o 90° w górę lub w dół w płaszczyźnie pionowej. Jak mógłby wyglądać model matematyczny odzwierciedlający możliwe ruchy wolnego końca ramienia? Skonstruuj algorytm jego przesunięcia z jednej dozwolonej pozycji do innej.

***1.3.** Mamy obliczyć iloczyn czterech macierzy liczb rzeczywistych: $M_1 \times M_2 \times M_3 \times M_4$. Macierz M_1 ma rozmiar 10×20, M_2 — 20×50, M_3 — 50×1, a M_4 — 1×100. Załóżmy, że pomnożenie dwóch macierzy o wymiarach, odpowiednio, $p \times q$ i $q \times r$ wymaga pqr operacji elementarnych (zgodnie z klasycznym schematem macierzy). Znajdź taką kolejność mnożenia wspomnianych macierzy, która zminimalizuje całkowitą liczbę wykonanych operacji elementarnych. Jak można uogólnić rozwiązanie tego zadania na dowolną liczbę macierzy?

****1.4.** Dane jest 100 liczb rzeczywistych będących pierwiastkami kwadratowymi z kolejnych liczb naturalnych od 1 do 100. Należy posegregować te liczby na dwie grupy w taki sposób, by ich sumy w obydwu grupach były jak najbardziej zbliżone do siebie. Gdybyś dysponował dwiema minutami czasu komputera, jakich obliczeń dokonałbyś w tym czasie w celu ułatwienia sobie rozwiązania tego zadania?

1.5. Zaproponuj „zachłanny" algorytm gry w szachy. Jakich wyników spodziewałbyś się w przypadku jego zastosowania?

1.6. W punkcie „Abstrakcyjne typy danych" rozpatrywaliśmy abstrakcyjny typ `SET` z operacjami elementarnymi: `MAKENULL`, `UNION` i `SIZE`. Załóżmy dla wygody, że ograniczamy się do zbiorów stanowiących podzbiory zbioru `{0, 1, ..., 31}`. Uwzględniając to ograniczenie, zaproponuj implementacje (w języku Pascal) wspomnianych operacji elementarnych.

1.7. Największym wspólnym dzielnikiem dwóch liczb całkowitych p i q nazywamy największą liczbę całkowitą d taką, że dzieli ona bez reszty zarówno p, jak i q. Zamierzamy zaimplementować następujący algorytm obliczania największego wspólnego dzielnika p i q — niech r będzie resztą z dzielenia p przez q; jeżeli r równe jest zero, to q jest szukanym największym wspólnym dzielnikiem, w przeciwnym razie przypisujemy $p := q$, $q := r$ i powtarzamy dzielenie[17].

(a) Udowodnij, że przedstawiony algorytm faktycznie znajduje największy wspólny dzielnik dwóch liczb.
(b) Zapisz ten algorytm w pseudojęzyku.
(c) Przekształć nieformalny program stworzony w punkcie (b) w poprawny program w języku Pascal.

[17] Algorytm ten wymyślony został ponad 2300 lat temu przez Euklidesa i dziś jest powszechnie znany pod jego nazwiskiem — *przyp. tłum.*

1.8. Zamierzamy stworzyć program formatujący tekst, wyrównujący każdy z wierszy do lewej i prawej krawędzi. Program używa dwóch buforów: dla słów i dla wierszy. Początkowo obydwa bufory są puste. Program wczytuje słowo do bufora słów i sprawdza, czy słowo to zmieści się jeszcze w bieżącym wierszu (zapisanym tymczasowo w buforze wiersza). Jeżeli tak, słowo dopisywane jest do bufora wiersza i bufor słowa jest opróżniany; jeżeli nie, w buforze wiersza między poszczególne słowa zostają równomiernie wstawione dodatkowe spacje, by wyrównać zawartość wiersza na obydwu krawędziach, po czym bufor wiersza jest drukowany i opróżniany.

(a) Zapisz algorytm działania programu formatującego w pseudojęzyku.

(b) Przekształć ten zapis w poprawny program w języku Pascal.

1.9. Jest n miast i dana jest tabela odległości między każdą ich parą. Napisz pseudoprogram, który znajduje krótką ścieżkę rozpoczynającą się i kończącą w tym samym mieście, przechodzącą przez każde z miast dokładnie jeden raz[18]. Ponieważ jedyną znaną metodą znajdowania *najkrótszej* ścieżki spełniającej podane warunki jest wyczerpujące poszukiwanie, zaproponuj jakiś algorytm heurystyczny znajdujący ścieżkę o długości możliwej do zaakceptowania.

1.10. Rozpatrzmy następujące funkcje zmiennej n:

$$f_1(n) = n^2$$

$$f_2(n) = n^2 + 1000n$$

$$f_3(n) = \begin{cases} n & \text{dla } n \text{ nieparzystych} \\ n^3 & \text{dla } n \text{ parzystych} \end{cases}$$

$$f_4(n) = \begin{cases} n & \text{dla } n \le 100 \\ n^3 & \text{dla } n > 100 \end{cases}$$

Znajdź wszystkie takie pary (i,j), że $f_i(n) = O(f_j(n))$ oraz takie, że $f_i(n) = \Omega(f_j(n))$.

1.11. Rozpatrzmy następujące funkcje zmiennej n:

$$g_1(n) = \begin{cases} n^2 & \text{dla parzystych } n \ge 0 \\ n^3 & \text{dla nieparzystych } n \ge 1 \end{cases}$$

$$g_2(n) = \begin{cases} n & \text{dla } 0 \le n \ge 100 \\ n^3 & \text{dla } n > 100 \end{cases}$$

$$g_3(n) = n^{2,5}$$

Znajdź wszystkie takie pary (i,j), że $g_i(n) = O(g_j(n))$ oraz takie, że $g_i(n) = \Omega(g_j(n))$.

1.12. Wyraź w notacji „dużego O” pesymistyczną złożoność czasową (w funkcji n) następujących procedur:

[18] Zagadnienie to znane jest pod nazwą problemu komiwojażera — *przyp. tłum.*

(a)
```
procedure matmpy ( n: integer);
    var
        i, j, k: integer;
    begin
        for i := 1 to n do
            for j:= 1 to n do begin
                 C[i, j] := 0;
                 for k : = 1 to n do
                     C[i, j] := C[i, j] + A[i, k] * B[k, j]
            end
        end
```

(b)
```
procedure mystery ( n: integer );
    var
        i, j, k: integer;
    begin
        for i := 1 to n-1 do
        for j := i + 1 to n do
            for k := 1 to j do
                { jakaś instrukcja wykonywana w czasie O(1)}
    end
```

(c)
```
procedure veryodd ( n: integer );
    var
        i, j, x, y: integer;
    begin
        for i := 1 to n do
            if odd(i) then begin
                for j : = i to n do
                    x := x + 1;
                for j:= 1 to i do
                    y := y + 1
            end
        end
```

(*d)
```
function recursive (n: integer) : integer;
    begin
        if n <= 1 then
            return (1)
        else
            return (recursive(n-1) + recursive(n-1))
    end
```

1.13. Udowodnij prawdziwość poniższych równości.

(a) $17 = O(1)$.

(b) $n(n–1)/2 = O(n^2)$.

(c) $\max(n^3, 10n^2) = O(n^3)$.

(d) $\sum_{i=1}^{n} i^k = O(n^{k+1})$ oraz $\sum_{i=1}^{n} i^k = \Omega(n^{k+1})$ dla całkowitych k.

(e) Jeżeli $p(x)$ jest dowolnym wielomianem k-tego stopnia z dodatnim współczynnikiem przy najwyższej potędze, to $p(n) = O(n^k)$ oraz $p(n) = \Omega(n^k)$.

***1.14.** Załóżmy, że $T_1(n) = \Omega(f(n))$ oraz $T_2(n) = \Omega(g(n))$. Które z poniższych stwierdzeń jest prawdziwe?

(a) $T_1(n)+T_2(n) = \Omega(\max(f(n), g(n)))$.

(b) $T_1(n)T_2(n) = \Omega(f(n)g(n))$.

***1.15.** Niektórzy autorzy definiują wielkość „dużej omegi" w następujący sposób: $f(n) = \Omega(g(n))$, jeżeli istnieją takie stałe dodatnie n_0 i c, że dla wszystkich $n \geq n_0$ zachodzi $f(n) \geq cg(n)$.

(a) Czy zgodnie z tą definicją prawdą jest, że $f(n) = \Omega(g(n))$ wtedy i tylko wtedy, gdy $g(n) = O(f(n))$?

(b) Czy stwierdzenie wyrażone w punkcie a) jest prawdziwe w kontekście definicji $\Omega(n)$ przedstawionej w podrozdziale „Czas wykonywania programu"?

(c) Czy rozwiązanie zadania 1.14 pozostaje niezmienne, jeżeli przyjąć nową definicję „dużej omegi"?

1.16. Uporządkuj następujące funkcje pod względem tempa wzrostu:

(a) n

(b) $\sqrt{n}$

(c) $\log n$

(d) $\log \log n$

(e) $\log^2 n$

(f) $n/\log n$

(g) $\sqrt{n} \times \log^2 n$

(h) $(1/3)^n$

(i) $(3/2)^n$

(j) 17

1.17. Załóżmy, że parametr n poniższej procedury jest dodatnią potęgą liczby 2. Znajdź formułę wyrażającą wypisywaną wartość zmiennej *count* w funkcji n.

```
procedure mystery ( n: integer);
    var
        x, count: integer;
    begin
        count := 0;
        x := 2;
        while x < n do begin
            x := 2 * x;
            count := count + 1;
        end;
        writeln(count);
    end;
```

1.18. Dana jest tablica A i funkcja *max(i,n)* zwracająca największy element z ciągu elementów $A[i] \ldots A[i + n - 1]$. Przyjmujemy dla wygody, że n jest potęgą liczby 2.

```
function max ( i, n: integer) : integer;
    var
        m1, m2: integer;
    begin
        if n = 1 then
```

```
            return (A[i])
        else begin
            m1 := max(i, n div 2);
            m2 := max(i+n div 2, n div 2);
            if m1 < m2 then
                return (m2)
        else
            return (m1)
        end
    end
```

(a) Niech $T(n)$ będzie pesymistycznym czasem wykonania w zależności od drugiego parametru n oznaczającego liczbę przeszukiwanych elementów. Napisz równanie wyrażające zależność $T(n)$ od $T(j)$ dla jednej lub kilku wartości $j < n$ oraz podaj stałą (lub stałe) reprezentującą (reprezentujące) czasy wykonywania poszczególnych instrukcji funkcji *max*.

(b) Podaj (w notacji „dużego O") ścisłą wartość górnego ograniczenia $T(n)$. Wartość ta powinna być równa dolnemu ograniczeniu (wyrażonemu w notacji „dużej omegi") i powinna być jak najprostsza.

Uwagi bibliograficzne

Koncepcja abstrakcyjnego typu danych ma swą genezę w klasie (typie *class*) języka Simula 67 (Birtwistle i in., [1973]). Od tego czasu powstało wiele innych języków zawierających abstrakcyjne typy danych, między innymi *Alphard* (Shaw, Wulf i London [1977]), *C++* (Stroustrup [1982]), *MESA* (Geschke, Morris i Satterthwaite [1977]) i *Russel* (Demers i Donahue [1979]). Koncepcja ADT jest przedmiotem rozważań prac Gotliebów [1978] oraz Wulfa i in. [1981].

Dzieło Knutha [1968] jest pierwszą ze znaczących publikacji promujących systematyczne studia nad czasem wykonywania programów. Aho, Hopcroft i Ullman [1974] odnoszą złożoność czasową i pamięciową algorytmów do różnorodnych modeli obliczeniowych, jak maszyny Turinga i maszyny o dostępie swobodnym. Zobacz także uwagi bibliograficzne do rozdziału 9. zawierające więcej odsyłaczy do źródeł traktujących o analizie algorytmów i programów.

Jako lekturę uzupełniającą na temat programowania strukturalnego polecić można opracowania Hoarego, Dahla i Dijkstry [1972], Wirtha [1973], Kernighana i Plaugera [1974] oraz Yourdona i Constantine [1975]. Organizacyjne i psychologiczne aspekty realizacji dużych projektów programistycznych dyskutowane są przez Brooksa [1974] i Weinberga [1971]. Kernighan i Plauger [1981] demonstrują sposoby budowania użytecznych narzędzi programistycznych w Pascalu.