

*Kazimierz Jakubczyk*

# **WPROWADZENIE DO ALGORYTMÓW I STRUKTUR DANYCH**



***Recenzenci:***

prof. dr hab. inż. **Stanisław KOWALIK**, Politechnika Śląska w Gliwicach

prof. dr hab. inż. **Tadeusz KRUPA**, Politechnika Warszawska

Copyright © by Politechnika Radomska, Wydawnictwo (2005, 2007)  
26-600 Radom, ul. Malczewskiego 20A,  
tel. (048) 3617033 fax (048) 3617034

e-mail: [wydawnictwo@pr.radom.pl](mailto:wydawnictwo@pr.radom.pl)

[www.pr.radom.pl](http://www.pr.radom.pl)

**ISBN 83-7351-193-3** (wyd. I – 2005)

**ISBN 978-83-7351-223-8** (wyd. II – 2007)

Utwór w całości ani we fragmentach nie może być powielany ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych, kopiujących, nagrywających i innych bez pisemnej zgody posiadacza praw autorskich.

## Spis treści

<b>Wstęp.....</b>	<b>5</b>
<b>Rozdział 1. Algorytmy, ich analiza i metody tworzenia .....</b>	<b>9</b>
1.1. Rozsądny czas wykonania algorytmu.....	10
1.2. Dokładność algorytmów numerycznych .....	12
1.3. Reprezentacje algorytmów .....	15
1.4. Pomiar czasu wykonania programu.....	28
1.5. Złożoność obliczeniowa algorytmów .....	29
1.6. Przeszukiwanie sekwencyjne i binarne .....	32
1.7. Notacja asymptotyczna.....	38
1.8. Rekurencja a iteracja .....	42
1.9. Metody tworzenia algorytmów.....	48
Ćwiczenia .....	51
<b>Rozdział 2. Sortowanie .....</b>	<b>57</b>
2.1. Sortowanie przez selekcję .....	59
2.2. Sortowanie przez wstawianie .....	61
2.3. Sortowanie przez wstawianie binarne .....	63
2.4. Sortowanie szybkie.....	66
2.5. Sortowanie kopcowe .....	71
2.6. Porównanie algorytmów sortowania wewnętrznego .....	80
2.7. Znajdowanie mediany .....	88
2.8. Algorytmy sortowania zewnętrznego .....	91
Ćwiczenia .....	97
<b>Rozdział 3. Listowe struktury danych.....</b>	<b>101</b>
3.1. Listy, stosy, kolejki.....	102
3.2. Implementacja tablicowa listy .....	105
3.3. Implementacja dowiązaniowa listy .....	109
3.4. Implementacje w Delphi.....	115
3.5. Kolejki priorytetowe.....	124
3.6. Zbiory .....	134
3.7. Słowniki.....	136
3.8. Mieszanie (haszowanie) .....	140
Ćwiczenia .....	143
<b>Rozdział 4. Algorytmy grafowe.....</b>	<b>147</b>
4.1. Podstawowe pojęcia .....	148
4.2. Sposoby reprezentowania grafów.....	151

4.3. Przeszukiwanie grafu w głąb .....	154
4.4. Przeszukiwanie grafu wszerz .....	157
4.5. Drzewo rozpinające grafu .....	159
4.6. Spójne składowe grafu nieskierowanego .....	167
4.7. Znajdowanie najkrótszych ścieżek w grafie .....	168
4.8. Implementacja algorytmu Dijkstry w Delphi .....	176
Ćwiczenia .....	186
<b>Rozdział 5. Algorytmy rekurencyjne .....</b>	<b>189</b>
5.1. Drzewko .....	190
5.2. Krzywe Sierpińskiego .....	194
5.3. Problem plecakowy .....	204
5.4. Problem ośmiu hetmanów .....	210
5.5. Wypłacalność kwoty .....	219
5.6. Interpretator wyrażeń arytmetycznych .....	223
Ćwiczenia .....	234
<b>Dodatek. Programowanie wizualne w Delphi .....</b>	<b>241</b>
D.1. Elementy i konfiguracja środowiska .....	241
D.2. Klasy, obiekty, komponenty .....	244
D.3. Paleta komponentów i jej wykorzystanie .....	246
D.4. Inspektor obiektów, właściwości i zdarzenia .....	248
D.5. Pliki tworzone przez środowisko .....	253
D.6. Kod źródłowy aplikacji i jego edycja .....	256
D.7. Liczby rzymskie .....	263
D.8. Kreślenie hipocykloidy .....	276
<b>Literatura .....</b>	<b>289</b>
<b>Skorowidz .....</b>	<b>291</b>
<b>Summary (streszczenie w języku angielskim) .....</b>	<b>301</b>

## Wstęp

Algorytmy i struktury danych są podstawą współczesnego programowania komputerowego. Rozwiązanie jakiegokolwiek problemu za pomocą komputera wymaga użycia odpowiedniego programu, w którym korzysta się z jakiegoś algorytmu i mniej lub bardziej skomplikowanych struktur danych. Zazwyczaj algorytm jest rozumiany jako przepis rozwiązywania określonego typu zadania przez komputer, aczkolwiek rozwiązującym zadanie może być również człowiek, oczywiście pod warunkiem, że wystarczy mu czasu na wykonanie krok po kroku wszystkich operacji przewidzianych w algorytmie. Program jest konkretnym sformułowaniem (zapisem w języku „zrozumiałym dla komputera”) algorytmu działającego na danych elementarnych (liczby, znaki, łańcuchy, wskaźniki) lub strukturach danych reprezentowanych w pamięci komputera za pomocą tablic i rekordów.

Rozwój metod tworzenia i analizy algorytmów sprawił, że programowanie przeszło metamorfozę od rzemiosła do dyscypliny naukowej. Istnieje niezbyt liczna grupa algorytmów i struktur danych, które są przydatne niemal we wszystkich programach. Ich opanowanie jest szczególnie ważne, ponieważ na nich opiera się umiejętność układania programów, one też służą do budowania bardziej złożonych algorytmów i struktur danych przy rozwiązywaniu większych problemów. Grupa ta obejmuje przede wszystkim algorytmy przeszukiwania i sortowania oraz struktury podstawowe, takie jak listy, kolejki, zbiory, słowniki, drzewa i grafy reprezentowane w postaci tablicowej lub dowiązaniowej<sup>1</sup>.

Niniejsza książka stanowi wprowadzenie w dziedzinę algorytmów i struktur danych. Jej tematyką są podstawowe zagadnienia algorytmiczne uważane obecnie za klasyczne. Zestaw omówionych zagadnień może wydawać się dość wybiórczy, jednak jest na tyle reprezentatywny, że ich opanowanie wystarczy do tworzenia większości efektywnych programów i pozwoli na uniknięcie „odkrywania tego, co już zostało odkryte”. Jako narzędzie opisu algorytmów i struktur danych wykorzystany został pseudojęzyk zbliżony do Pascala, stanowiący mieszaninę konstrukcji pascalowych i wyrażeń w języku naturalnym. Przyjęte podejście ma tę zaletę, że zapisy algorytmów w pseudojęzyku są przejrzyste i zrozumiałe, a ponadto dają się łatwo przekształcić do poprawnych programów w języku Pascal, które można skompilować i wykonać. Wiele algorytmów zostało zaprezentowanych w postaci pełnych programów opracowanych w środowisku Delphi firmy Borland, ponieważ jest ono oparte na Pascalu i zawiera szereg mechanizmów obiektowych ułatwiają-

---

<sup>1</sup> Struktura dowiązaniowa jest zbiorem rekordów, z których każdy zawiera dane i jeden lub większą liczbę wskaźników (dowiązań) do innych rekordów.

cych realizację wielu rozważanych struktur danych. Algorytmy te można łatwo zaimplementować w innym języku programowania, np. w C, C++ lub Java.

Książka jest adresowana do tych Czytelników, dla których algorytmika ma pełnić użytkową rolę w wykonywaniu zawodu. W szczególności może posłużyć jako podręcznik studentom informatyki, a także studentom kierunków nieinformatycznych, którzy są przygotowywani do rozwiązywania problemów za pomocą komputera podczas zajęć z przedmiotów **Algorytmy i struktury danych** lub **Algorytmika i programowanie**. Może też być pomocna dla uczniów liceów, którzy w ramach zajęć w szkole lub pozalekcyjnych w domu pogłębiają swoją wiedzę informatyczną, a nawet dla programistów, którym zdarza się marnować czas na pisanie programu opartego na złym algorytmie.

Czytelnik sięgający po tę książkę powinien mieć pewne podstawowe przygotowanie w zakresie matematyki dyskretnej, przynajmniej na poziomie licealnym, umieć programować w języku Pascal, najlepiej w środowisku Delphi lub Turbo Pascal, i znać system Windows z perspektywy użytkownika. Założenie, że potrafi układać programy w Pascalu, znajduje uzasadnienie. Wszak język ten jest ogólnie uznany jako wprowadzający w naukę programowania ze względu na przystępność i wysokie walory dydaktyczne. Jednocześnie trzeba podkreślić, że Czytelnik nieobeznany z Delphi nie musi się go obawiać, ponieważ przejście od popularnego środowiska Turbo Pascal do nowoczesnego Delphi z językiem programowania Object Pascal wydaje się szczególnie naturalne. Chociaż Delphi jest mocno rozbudowane, to jednak jest na tyle intuicyjne, że szybko nabywa się w nim wprawy, nawet już podczas tworzenia pierwszej aplikacji. Z uwagi na rozbudowany system pomocy w Delphi niewątpliwym atutem jest znajomość języka angielskiego.

W książce wykorzystano wersję 7. Delphi. Wszystkie prezentowane przykłady dadzą się skompilować i uruchomić w najuboższej, darmowej w przypadku wykorzystania do celów niekomercyjnych, dystrybucji Delphi 7 Personal. Nie powinno to jednak sugerować, że są one bezużyteczne w odniesieniu do innych wersji Delphi. Z pewnością powinny działać w wersjach nowszych, a także bez większych problemów w wersjach poprzednich, zwłaszcza 6 i 5. Książki nie należy traktować jako podręcznika programowania w Delphi. Jest ono tylko wygodnym środowiskiem programistycznym pozwalającym na zaprezentowanie części omawianych algorytmów i struktur danych w jednym z najpopularniejszych obecnie języków programowania wysokiego poziomu – Object Pascalu.

Książka składa się z pięciu rozdziałów, dodatku o Delphi, wykazu cytowanej literatury i skorowidza ułatwiającego wyszukiwanie potrzebnych informacji. Na końcu każdego rozdziału znajdują się ćwiczenia do samodzielnego rozwiązania, związane tematycznie z omawianymi zagadnieniami.

Rozdział 1. koncentruje się wokół pojęcia algorytmu i stanowi wprowadzenie do analizy algorytmów. Przedstawia podstawowe właściwości i sposoby reprezen-

towania algorytmów. Zawiera intuicyjne i formalne wprowadzenie do pojęcia złożoności obliczeniowej algorytmu i jej asymptotycznego zachowania. Ukazuje różnice pomiędzy rekurencją a iteracją i przedstawia podstawowe metody konstruowania efektywnych algorytmów. Omawiane pojęcia i techniki są poparte licznymi przykładami algorytmów, takimi jak algorytm Euklidesa, sortowanie bąbelkowe (ang. *bubble sort*), przeszukiwanie sekwencyjne (ang. *sequential search*) i binarne (ang. *binary search*) oraz obliczanie liczb Fibonacciego.

Rozdział 2. poświęcony jest prostym algorytmom sortowania wewnętrznego, takim jak sortowanie przez wybieranie (ang. *selection sort*) i sortowanie przez wstawianie (ang. *insertion sort*), a także wydajnym algorytmom sortowania, jak sortowanie szybkie (ang. *quick sort*) i sortowanie kopcowe (ang. *heap sort*), zwane również sortowaniem stogowym. Przedstawia też algorytmy znajdowania statystyk pozycyjnych i sortowania zewnętrznego.

Rozdział 3. dotyczy sposobu reprezentacji danych, który jest zdeterminowany zarówno możliwościami komputera, jak i operacjami wykonywanymi na danych. Omawia elementarne struktury listowe, takie jak listy, stosy i kolejki, jak również kolejki priorytetowe, zbiory i słowniki implementowane za pomocą list i tablic mieszających (haszujących). Przedstawia także metody ich reprezentowania w pamięci komputera i implementowania w środowisku Delphi.

Rozdział 4. stanowi wprowadzenie w obszerną tematykę grafów. Po zapoznaniu z podstawowymi pojęciami przedstawia sposoby reprezentowania grafów w komputerze. Opisuje algorytmy przeszukiwania grafów w głąb i wszerz, wyznaczania drzewa rozpinającego i spójnych składowych grafu, a także znajdowania najkrótszych ścieżek z danego wierzchołka (źródła) do pozostałych wierzchołków grafu (algorytm Dijkstry) i najkrótszych ścieżek między wszystkimi parami wierzchołków (algorytm Floyda-Warshalla).

Rozdział 5. ukazuje potęgę rekurencji. Zawiera kilka przykładów uzasadnionej rekurencji, która pozwala na znaczne uproszczenie zapisu algorytmów i poprawienie czytelności kodu programów. Dotyczą one kreślenia atrakcyjnych motywów graficznych (drzewko rekurencyjne, krzywe Sierpińskiego) i poszukiwania rozwiązania metodą prób i błędów, czyli tzw. algorytmów z powrotami (problem plecakowy, problem ośmiu hetmanów, problem wypłacalności kwoty z zawartości portmonetki). Rozdział kończy przykład budowy interpretatora elementarnego języka programowania.

Dodatek, stanowiący wprowadzenie do programowania wizualnego w Delphi, został przygotowany z myślą o grupie Czytelników, którzy nie znają tego środowiska. Oczywiście, nie zastąpi on pełnego podręcznika poświęconego Delphi, ale umożliwi lekturę tej książki. W dodatku omówiono elementy środowiska i zasady ich wykorzystania, zaprezentowano podstawowe idee programowania wizualnego oparte na komponentach, ich metodach, właściwościach i zdarzeniach, a także

zademonstrowano za pomocą dwóch przykładów technikę projektowania aplikacji okienkowych wyposażonych w graficzny interfejs użytkownika.

Wszystkie omawiane w książce programy w Delphi są zamieszczone na stronie internetowej autora pod adresem [www.kaj.pr.radom.pl/mat.html](http://www.kaj.pr.radom.pl/mat.html). Czytelnik może ściągnąć ich wersje źródłowe i wykonawcze oraz używać bez ograniczeń.

Jestem niezmiernie wdzięczny obu Recenzentom, Panu prof. Stanisławowi Kowalikowi i Panu prof. Tadeuszowi Krupie, za przychylne i wnikliwe recenzje. Ich cenne i rzeczowe uwagi przyczyniły się niewątpliwie do poprawienia jakości niniejszej książki, za co Im serdecznie dziękuję. Słowa wdzięczności należą się również Panu prof. Grzegorzowi Kiedrowiczowi i Pani mgr Elwirze Mortka, a także wszystkim tym, którzy przyczynili się do wydania tej książki.

*Kazimierz Jakubczyk*



## Rozdział 1.

### Algorytmy, ich analiza i metody tworzenia

Zbudowanie programu komputerowego jest zazwyczaj procesem złożonym, obejmującym sformułowanie problemu i jego specyfikacji oraz projektowanie rozwiązania i zaimplementowanie go w konkretnym języku programowania. Oczywiście, niezbędne jest przy tym opracowanie dokumentacji oraz przetestowanie programu i weryfikacja wyprodukowanych przez niego wyników. Program jest jedynie sformalizowanym zapisem algorytmu rozwiązywania problemu.

**Algorytmem** (ang. *algorithm*)<sup>1</sup> nazywamy przepis opisujący krok po kroku postępowanie prowadzące do rozwiązania postawionego problemu. Słowem *algorytm* określa się także skończony zestaw reguł opisujących sekwencję operacji, pozwalającą znaleźć rozwiązanie problemu w rozsądnym czasie. Ogólnie przyjmuje się, że najbardziej istotnymi własnościami algorytmów są [15, 30]:

- ♦ **skończoność** – wykonanie algorytmu zawsze zatrzymuje się po skończonej liczbie kroków (własność stopu),
- ♦ **dobrze zdefiniowanie** – każdy krok algorytmu jest opisany precyzyjnie,
- ♦ **dane wejściowe**, nazywane krócej **danymi** – algorytm posiada zero lub więcej danych wejściowych,
- ♦ **dane wyjściowe**, nazywane inaczej **wynikiem** – algorytm generuje jedną lub więcej danych wyjściowych,
- ♦ **efektywne zdefiniowanie** – wszystkie operacje algorytmu dają się wykonać dokładnie i w skończonym czasie, tzn. są zdefiniowane i można je wykonać np. za pomocą ołówka i kartki.

Algorytm nazywamy **poprawnym**, gdy dla każdego danych zgodnych ze specyfikacją problemu zatrzymuje się i daje poprawny wynik<sup>2</sup>. Przez **specyfikację** problemu rozumiemy jego ścisły opis precyzujący, jakie warunki muszą spełniać dane i wyniki oraz jakie związki mają zachodzić między nimi. Konkretnie dane spełniające wymagania określone w specyfikacji stanowią tzw. **egzemplarz** (ang.

---

<sup>1</sup> Słowo *algorytm* pochodzi od nazwiska Muhammad ibn Musa al-Chorezmi (łac. *Algorismus*), arabskiego matematyka, geografa i astronoma żyjącego ok. 780 – 850 r. Jego prace przyczyniły się do rozpowszechnienia cyfr arabskich i systemu dziesiętnego.

<sup>2</sup> Stwierdzenia typu „zatrzymuje się, daje wynik, oblicza, rozwiązuje” w odniesieniu do algorytmu czy programu są powszechnie stosowanymi skrótami myślowymi. W istocie odnoszą się one do procesu wykonywanego według algorytmu lub programu.

*instance*) problemu. Algorytm jest więc poprawny, gdy dla każdego egzemplarza problemu kończy się i daje żądany wynik, zgodny ze specyfikacją. Algorytm niepoprawny może dla pewnego egzemplarza problemu nie zatrzymać się albo po zatrzymaniu dać zły wynik.

Należy podkreślić, że poprawność algorytmu można jedynie udowodnić, podobnie jak dowodzi się twierdzenia matematyczne. W wielu dowodach korzysta się z indukcji matematycznej [7, 15] lub tzw. metody niezmienników [4, 23, 29]. Dowodzenie poprawności algorytmów, zwane też **weryfikacją**, jest często procesem długim i skomplikowanym, toteż w praktyce algorytmy zazwyczaj się tylko testuje, zwłaszcza ich realizacje komputerowe. Testowanie nie gwarantuje jednak poprawności algorytmu, gdyż nie uwzględnia wszystkich egzemplarzy problemu, a tylko ich część. Słynna zasada (E. Dijkstra) orzeka, że testowanie może jedynie wykazać obecność błędów, ale nie ich brak [13, 29].

## 1.1. Rozsądny czas wykonania algorytmu

Niezwykle ważne jest, aby rozwiązanie problemu było osiągalne w rozsądnym czasie. Nawet gdy algorytm ma własność stopu i jest dobrze i efektywnie zdefiniowany, jego czas realizacji może być nie do zaakceptowania pomimo zawrotnej, jak może się wydawać, szybkości działania współczesnych komputerów.

Aby się przekonać, jak długi może być okres oczekiwania na wynik, rozpatrzmy problem obliczania wyznacznika  $\det(A)$  macierzy kwadratowej  $A$  o  $n$  wierszach i  $n$  kolumnach:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}.$$

W wielu podręcznikach algebry można znaleźć, przyjmowany niekiedy za definicję wyznacznika, następujący wzór:

$$\det(A) = \sum_{k_1 k_2 \dots k_n} \text{sgn}(k_1 k_2 \dots k_n) a_{1k_1} a_{2k_2} \dots a_{nk_n}, \quad (1.1)$$

w którym  $k_1 k_2 \dots k_n$  oznacza permutację zbioru liczb  $\{1, 2, \dots, n\}$ , a  $\text{sgn}(k_1 k_2 \dots k_n)$  znak tej permutacji równy  $+1$  albo  $-1$  w zależności od tego, czy jest ona parzysta, czy nieparzysta [2, 18]. Sumowanie odbywa się po wszystkich permutacjach zbioru  $\{1, 2, \dots, n\}$ , których jest  $n!$ . Wszystkich wyrazów jest więc  $n!$ , a każdy z nich jest iloczynem  $n$  czynników, po jednym z każdego wiersza macierzy  $A$ .

Powszechnie znane i stosowane są dwa szczególne przypadki wzoru (1.1), mianowicie dla  $n = 2$ :

$$\det(A) = a_{11}a_{22} - a_{12}a_{21}$$

i dla  $n = 3$  (wzór Sarrusa):

$$\begin{aligned} \det(A) = & a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + \\ & + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} \end{aligned}$$

Oba wzory określają bardzo efektywny algorytm obliczania wyznacznika macierzy o małym rozmiarze  $n$ , przez co są często stosowane w praktyce obliczeniowej.

Sytuacja zmienia się jednak istotnie, gdy  $n$  rośnie. Analizując wzór (1.1), łatwo jest policzyć łączną liczbę wymaganych mnożeń oraz dodawań i odejmowań:

$$T(n) = n!(n-1) + (n!-1) = n \cdot n! - 1 \quad (1.2)$$

Założmy teraz, że dysponujemy superkomputerem, który w ciągu sekundy potrafi wykonać 1 bilion, czyli  $10^{12}$  operacji arytmetycznych. W celu zbadania, jak długo ten hipotetyczny komputer będzie obliczał dla różnych wartości  $n$  wyznacznik macierzy  $A$ , działając według algorytmu (1.1), posłużymy się prostą aplikacją konsolową w Delphi, wykorzystującą wzór (1.2):

```

program Wyznacznik;

{$APPTYPE CONSOLE}

uses
  SysUtils;

const
  s = 1.0E+12;      // Liczba operacji na sekundę
  m = 60*s;         // Liczba operacji na minutę
  h = 60*m;         // Liczba operacji na godzinę
  d = 24*h;         // Liczba operacji na dzień
  r = 365.25*d;     // Liczba operacji na rok

var
  n, k: integer;
  t   : real;       // T(n)

begin
  Write('n = ');
  ReadLn(n);
  t := n;
  for k:=2 to n do
    t := t*k;

```

```

t := t-1;
if t >= r then WriteLn(t/r, ' lat') else
  if t >= d then WriteLn(t/d, ' dni') else
    if t >= h then WriteLn(t/h, ' godziny') else
      if t >= m then WriteLn(t/m, ' minuty') else
        WriteLn(t/s, ' sekundy');
ReadLn;
end.

```

Wyniki tego programu dla wybranych wartości  $n$  przedstawia tabela 1.1. Być może wydają się one zaskakujące, pomimo że nie uwzględniają czasu wykonania operacji organizacyjnych, np. generowania permutacji, które nie należy do zadań banalnych [15, 18]. W każdym razie pokazują nader wyraziście, że rozpatrywany algorytm nie może przynieść żadnego pożytku w praktyce obliczeniowej, oczywiście poza wymienionymi wyżej przypadkami dla  $n = 2$  i  $3$ .

**Tabela 1.1.** Czas obliczeń hipotetycznego komputera

$n$	Czas obliczeń
14	1,2 sekundy
16	5,6 minuty
17	1,7 godziny
19	27 dni
21	34 lat
50	$5 \times 10^{46}$ lat

Często dany problem daje się rozwiązać za pomocą kilku algorytmów. Warto więc znać odpowiednie algorytmy nie tylko dlatego, by skorzystać z gotowego rozwiązania, ale także, by wybrać algorytm praktyczny i najbardziej efektywny. Na przykład wyznacznik macierzy o  $50 \times 50$  elementach można rozwiązać na typowym komputerze osobistym zaledwie w ciągu ułamka milisekundy za pomocą metody eliminacji Gaussa [9].

## 1.2. Dokładność algorytmów numerycznych

W algorytmach numerycznych działających na liczbach zmiennopozycyjnych, np. typu *real* w Turbo i Object Pascalu, pojawia się kwestia wiarygodności wyniku obliczeń uzyskanego przez komputer. Zarówno zapis zmiennopozycyjny liczb, jak i działania w arytmetyce zmiennopozycyjnej są z natury swojej obciążone błędami, które mogą się kumulować, wypaczając wynik obliczeń. Jeżeli kumulują się nadmiernie, może on całkowicie odbiegać od wyniku dokładnego.

Rozważmy zaczerpnięty z książki [12] przykład wyznaczania pierwiastków  $x_1, x_2$  równania kwadratowego:

$$x^2 - 2px + q = 0 \quad (p^2 - q > 0) \quad (1.3)$$

Oto dwa algorytmy dające rozwiązanie tego problemu:

#### Algorytm szkolny

$$\begin{aligned} x_1 &:= p + \sqrt{p^2 - q} \\ x_2 &:= p - \sqrt{p^2 - q} \end{aligned}$$

#### Algorytm zmodyfikowany

```
if p ≥ 0 then
    x1 := p + √(p² - q)
    x2 := q / x1
else
    x2 := p - √(p² - q)
    x1 := q / x2
```

Symbol  $:=$  oznacza operację przypisania wartości zmiennej, podobnie jak w języku Pascal. Algorytm szkolny rozwiązuje zadanie za pomocą powszechnie znanych wzorów z wyróżnikiem  $\Delta$ , które w przypadku równania (1.3) stają się nieco prostsze. Algorytm zmodyfikowany najpierw wyznacza w taki sposób większy co do wartości bezwzględnej pierwiastek, a następnie mniejszy na podstawie drugiego ze wzorów Viète'a wyrażających sumę i iloczyn pierwiastków:

$$x_1 + x_2 = 2p, \quad x_1 \cdot x_2 = q$$

Aby sprawdzić zachowanie obu algorytmów w arytmetyce zmiennopozycyjnej, posłużymy się znowu aplikacją konsolową w Delphi, która pozwala prześledzić ich wyniki dla różnych wartości współczynników  $p, q$  równania (1.3):

```
program Pierwiastki;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  p, q, x1, x2: real;

begin
  Write('p = '); ReadLn(p);
  Write('q = '); ReadLn(q);
```

```

x1 := p+sqrt(p*p-q);           // Algorytm szkolny
x2 := p-sqrt(p*p-q);
WriteLn('x1 =', x1, ' x2 =', x2);

if p >= 0 then                  // Algorytm zmodyfikowany
begin
  x1 := p+sqrt(p*p-q);
  x2 := q/x1;
end else
begin
  x2 := p-sqrt(p*p-q);
  x1 := q/x2;
end;
WriteLn('x1 =', x1, ' x2 =', x2);

ReadLn;
end.

```

Uruchamiając ten program dla szeregu danych, możemy zaobserwować, że algorytmy szkolny i zmodyfikowany nie zawsze dają takie same wyniki. Przykładowo, dla  $p = 100000$  i  $q = 1$  otrzymujemy w przypadku pierwszego wynik:

```
x1 = 1.99999999995000E+0005  x2 = 5.000000000158707E-0006
```

a w przypadku drugiego:

```
x1 = 1.99999999995000E+0005  x2 = 5.000000000012500E-0006
```

Oczywiście, różnić się mogą tylko wartości jednego z pierwiastków, a powodem tej różnicy jest wykonywanie obliczeń z ograniczoną liczbą cyfr znaczących, która dla typu *real*, począwszy od Delphi 4, wynosi 15 – 16 cyfr dziesiętnych.

Upraszczając znacznie rozważania, zdajmy się jedynie na intuicję i wyczucie. Zauważmy mianowicie, że gdy odejmujemy dwie bliskie wartości przybliżone, to ich cyfry bardziej znaczące się redukują, przez co następuje utrata dokładności uzyskanej różnicy, ponieważ o jej wartości decydują cyfry mniej znaczące, a więc mniej pewne<sup>3</sup>. Tak dzieje się w przypadku algorytmu szkolnego, gdy współczynnik  $p$  jest co do wartości bezwzględnej znacznie większy od współczynnika  $q$ . W przypadku algorytmu zmodyfikowanego unika się takiej sytuacji, wybierając najpierw ten pierwiastek, którego wartość daje się obliczyć na podstawie wzoru, w którym są dodawane dwie wartości o tych samych znakach.

---

<sup>3</sup> Innym znanym zjawiskiem w arytmetyce zmiennopozycyjnej, którego konsekwencje mogą być nieprzewidywalne, jest zaburzenie wyniku sumowania dwóch znacznie różniących się co do wartości bezwzględnej składników, który może zostać zredukowany do większego z nich. Z mnożeniem nie ma takich problemów.

Dokładna analiza problemu pokazuje, że algorytm szkolny może spowodować dowolnie duże zaburzenie jednego z pierwiastków, zaś algorytm zmodyfikowany zapewnia numeryczną poprawność obydwu pierwiastków [12]. Inaczej mówiąc, pierwszy algorytm jest **numerycznie niestabilny**, a drugi **stabilny**.

Algorytmy numeryczne i ich własności obliczeniowe są tematyką dziedziny naukowej zwanej **metodami numerycznymi** lub **analizą numeryczną**, stojącej na pograniczu matematyki i informatyki. W rozważaniach typowo informatycznych zazwyczaj przyjmuje się, że wszelkie obliczenia są wykonywane w arytmetyce nieskończonej precyzji, a zatem są dokładne. Przytoczony przykład uświadamia jednak, że nawet proste zadania obliczeniowe mogą stanowić problem, gdy są rozwiązywane za pomocą komputera. Utrata dokładności wyniku obliczeń prowadzonych w arytmetyce zmiennopozycyjnej może być konsekwencją nie tylko błędów zaokrągleń działań na liczbach przybliżonych, lecz także własności numerycznej zastosowanego algorytmu.

### 1.3. Reprezentacje algorytmów

Najbardziej ogólną i jednocześnie mało precyzyjną reprezentacją algorytmów jest ich **opis słowny** w języku naturalnym. Ta postać niesie duże ryzyko niejednoznaczności i trudno jest w niej opisać algorytmy złożone, toteż jeśli się ją stosuje, to głównie w celu zasugerowania metody rozwiązania problemu.

Popularnym i dokładnym sposobem zapisu algorytmów, użytecznym zwłaszcza wtedy, gdy są one kierowane do osób niekoniecznie obytych z programowaniem, jest **lista kroków**. Poszczególne kroki zawierają opisy operacji przewidzianych w algorytmie, stanowiące zazwyczaj mieszankę sformułowań w języku naturalnym i matematycznym. Domyślnie kolejność wykonywania kroków jest zgodna z ich kolejnością występowania w liście. Wyjątkiem są operacje przejścia do kroku o wskazanym numerze i operacja zakończenia algorytmu.

Aby zilustrować użycie listy kroków, posłużymy się problemem znalezienia **największego wspólnego dzielnika** dwóch liczb całkowitych, czyli największej liczby całkowitej, która dzieli obie liczby bez reszty. Oto specyfikacja problemu:

**Dane:**  $m, n$  – liczby całkowite dodatnie.

**Wynik:**  $\text{NWD}(n, m)$  – największy wspólny dzielnik liczb  $m$  i  $n$ .

Najbardziej znana na świecie metoda wyznaczania największego wspólnego dzielnika nosi nazwę **algorytmu Euklidesa**, od nazwiska greckiego matematyka, który opublikował ją około 300 roku p.n.e. Prawdopodobnie algorytm był znany 200 lat wcześniej. Jego zapis w postaci listy kroków jest następujący:

- K1.** Podziel  $n$  przez  $m$ . Niech  $r$  oznacza resztę z tego dzielenia.
- K2.** Jeżeli  $r = 0$ , szukany wynikiem jest  $m$ , więc zakończ algorytm.
- K3.** Wykonaj  $n := m$ ,  $m := r$  i wróć do kroku K1.

Oczywiście, Euklides opisał algorytm w inny sposób<sup>4</sup>. Warto wspomnieć, że algorytm Euklidesa jest najstarszym, który zawiera iterację.

Aby udowodnić poprawność algorytmu, rozważmy zależności:

$$n = q \cdot m + r, \quad 0 \leq r < m, \quad (1.4)$$

w których  $q$  jest ilorazem, a  $r$  resztą dzielenia  $n$  przez  $m$ . Jeżeli  $r = 0$ , to każdy dzielnik  $m$  jest dzielnikiem  $n$ , zatem i największy dzielnik liczby  $m$ , którym jest ona sama, jest dzielnikiem  $n$ . Jeżeli natomiast  $r > 0$ , to z zależności (1.4) wynika, że każdy wspólny dzielnik liczb  $m$  i  $n$  jest dzielnikiem liczby  $r$ , a także każdy wspólny dzielnik  $r$  i  $m$  jest dzielnikiem  $n$ . Zatem liczby  $n$  i  $m$  oraz  $m$  i  $r$  mają te same wspólne dzielniki, więc także największy wspólny dzielnik. Można więc parę liczb  $n$  i  $m$  zastąpić parą liczb  $m$  i  $r$ :

$$\text{NWD}(n, m) = \text{NWD}(m, r), \quad (1.5)$$

co ma miejsce w kroku K3. Ponadto z nierówności  $0 \leq r < m$  wynika, że ciąg reszt  $r$  otrzymywanych w kroku K1 jest malejący i ograniczony, a ponieważ jego wyrazy są całkowite, po skończonej liczbie kroków wystąpią równości:

$$n = q \cdot m + r, \quad r = 0.$$

Oznacza to, że w kroku K2 zostanie osiągnięty koniec algorytmu, a wtedy szukany największy wspólny dzielnik będzie równy  $m$ .

Obecnie najbardziej powszechną formą prezentacji algorytmów jest ich zapis w **pseudojęzyku** zbliżonym do „prawdziwego” języka programowania wysokiego poziomu, najczęściej do Pascala, C lub C++. W pseudojęzyku nie uwzględnia się technik programowania, np. stosuje abstrakcyjne typy danych zamiast typów standardowych, pomija obsługę błędów i deklaracje zmiennych lokalnych, dopuszcza nieformalne opisy zmiennych i parametrów, a nawet korzysta z języka naturalnego i symboliki matematycznej. Takie podejście ma na celu możliwie jak największe uproszczenie opisu algorytmów i ułatwienie zrozumienia ich działania, a jednocześnie zachowanie ogólnych cech „prawdziwego” języka, co pozwala na proste przejście do zapisu akceptowanego przez komputer. Jeżeli znajdzie potrzeba zaimplementowania algorytmu zapisanego w pseudojęzyku, wystarczy przetłumaczyć go z pseudojęzyka na konkretny język programowania.

<sup>4</sup> Sformułowanie Euklidesa i interesujące uwagi można znaleźć w książce [16].



Przykładami reprezentacji algorytmów w pseudojęzyku są zamieszczone na początku podrozdziału 1.2 zapisy algorytmów wyznaczania pierwiastków równania kwadratowego (1.3), a zwłaszcza zapis algorytmu zmodyfikowanego oparty na pascalowej instrukcji warunkowej **if ... else**, w którym korzysta się z symboliki matematycznej, a zamiast normalnych oznaczeń instrukcji złożonej **begin ... end** stosuje odpowiednio dobrane wcięcia, które redukują długość tekstu i podnoszą jego przejrzystość. Odpowiednik tego zapisu w języku Pascal stanowi fragment przytoczonej aplikacji konsolowej *Pierwiastki*.

Nietrudno jest opisać algorytm Euklidesa w pseudojęzyku programowania. Oto jedna z możliwych wersji takiej prezentacji:

```
function EUCLID(n, m)
  r := n mod m
  while r ≠ 0 do
    n := m
    m := r
    r := n mod m
  return m
```

Zapis przypomina definicję podprogramu funkcyjnego w Pascalu. Rozpoczyna się nagłówkiem określającym nazwę algorytmu i jego parametry (dane  $m$  i  $n$ ). Oba parametry są **przekazywane przez wartość** [26, 29], co oznacza, że otrzymują one wartości, których zmiany wewnątrz podprogramu będą poza nim niewidoczne. Znajdująca się poza zakresem instrukcji iteracyjnej **while** instrukcja **return**, zapożyczona od języków C i C++, kończy algorytm i określa jego wynik.

Czytelnik mający nawet niewielkie doświadczenie w Object Pascalu być może woli implementację algorytmu Euklidesa w tym języku:

```
function Euclid(n, m: integer): integer;
var
  r: integer;
begin
  r := n mod m;
  while r <> 0 do
    begin
      n := m;
      m := r;
      r := n mod m;
    end;
  Result := m;
end;
```

Użycie pseudojęzyka programowania przedstawimy jeszcze na przykładzie algorytmu **sortowania** ciągu liczbowego w porządku niemalejącym. Specyfikacja tego problemu jest następująca:






















**Dane:** Ciąg  $n$  liczb  $a_1, a_2, \dots, a_n$ .

**Wynik:** Uporządkowanie tego ciągu od liczby najmniejszej do największej.

Jeden z najprostszych algorytmów sortowania, znany pod nazwą **sortowania bąbelkowego** (ang. *bubble sort*), polega na porównywaniu dwóch sąsiadujących elementów ciągu i przestawianiu ich, gdy są ustawione w odwrotnej kolejności. Proces sortowania odbywa się w przebiegach, w których te operacje wykonywane są według wzrastających indeksów elementów, poczynając za każdym razem od indeksu 1. W rezultacie mniejsze elementy wędrują w kierunku początku ciągu, a większe w kierunku końca. W pierwszym przebiegu największy element zajmie w ciągu pozycję  $n$ . W drugim przebiegu przeglądany jest ciąg krótszy o ten jeden element, w rezultacie drugi co do wielkości element ciągu zajmie pozycję  $n - 1$ . W następnych przebiegach największe elementy z coraz to krótszych ciągów zajmą właściwe pozycje, aż w końcu wszystkie elementy zostaną posortowane.

Tabela 1.2 ilustruje działanie algorytmu dla ciągu 7-elementowego. Numery  $k = 1, 2, \dots, 6$  oznaczają kolejne przebiegi, a strzałki – przesuwanie się mniejszych elementów w kierunku początku ciągu. Ich wędrówka przypomina ruch bąbelków powietrza w naczyniu z wodą, stąd wywodzi się nazwa algorytmu.

**Tabela 1.2.** Przykład sortowania bąbelkowego ciągu 7-elementowego

Pocz.	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$		
27		13	13		6	6		4
13		27		6	13	13		6
28		6		27		15		13
6		28		15		4		15
35		15		28		27		27
15		4		35		28		28
4		35		35		35		35

Algorytm sortowania bąbelkowego można zapisać w pseudokodzie w postaci bardzo prostego podprogramu przypominającego procedurę w języku Pascal:

```

procedure BUBBLE-SORT(a[1..n])
  for k:=1 to n-1 do
    for p:=1 to n-k do
      if a[p] > a[p+1] then
        tmp := a[p]
        a[p] := a[p+1]
        a[p+1] := tmp

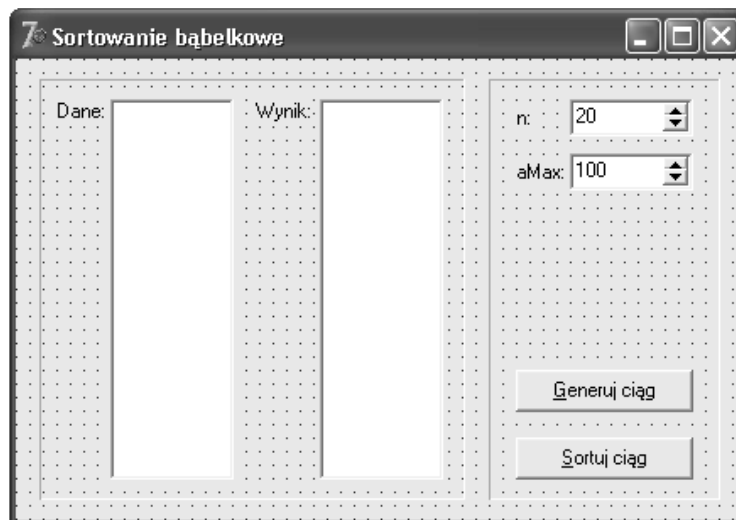
```

Parametrem podprogramu jest tablica  $a[1..n]$  zawierająca ciąg o długości  $n$ , który należy posortować<sup>5</sup>. W pseudojęzyku milcząco przyjmujemy, że tablice są **przekazywane przez zmienną** [26, 29], czyli że zmiany wartości ich elementów dokonywane wewnątrz podprogramu są widoczne na zewnątrz niego. Tablica  $a$  reprezentuje więc ciąg sortowany *in situ* (w miejscu), ponieważ jest ona zarówno parametrem wejściowym, jak i wyjściowym. Postać zagnieżdżonych instrukcji iteracyjnych **for** wynika ze spostrzeżenia, iż wszystkich przebiegów jest  $n - 1$  oraz że w przebiegu o numerze  $k = 1, 2, \dots, n - 1$  liczba porównań wynosi  $n - k$ .

Przypomnijmy, że w pseudojęzyku można korzystać z języka naturalnego. Operację zamiany dwóch elementów, złożoną z trzech przypisań i wymagającą użycia pomocniczej zmiennej  $tmp$ , moglibyśmy również zapisać krócej:

Zamień  $a[p]$  z  $a[p+1]$

Zajmiemy się teraz przetłumaczeniem algorytmu sortowania bąbelkowego na język Object Pascal i wykorzystaniem go w aplikacji okienkowej w Delphi, która pozwoli ocenić jakość algorytmu. Projekt formularza okna głównego tej aplikacji przedstawia rysunek 1.1. Znajdują się na nim dwie kontrolki *ListBox* (pola listy) i *SpinEdit* (pola edycji połączone z przyciskami) wraz z opisującymi je etykietami oraz dwa przyciski *Button* i komponenty *Bevel* (ozdobne wgłębienia).



**Rys. 1.1.** Projekt formularza aplikacji sortowania bąbelkowego

<sup>5</sup> Należy sobie zdawać sprawę, że taki opis parametru jest niezgodny ze składnią języka Pascal i w trakcie tworzenia implementacji trzeba się z nim jakoś uporać.

Aplikacja ma wygenerować losowo ciąg liczb całkowitych dodatnich, wyświetlić go w polu listy *ListBox1* opisanym etykietą *Dane*, posortować i wyświetlić w polu *ListBox2* opisanym etykietą *Wynik*. Długość ciągu i górne ograniczenie jego elementów będzie można określać za pomocą pól edycji *SpinEdit1* i *SpinEdit2* opisanych etykietami *n* i *aMax*. Właściwości *Value*, *MinValue* i *MaxValue* tych pól ustawiamy np. na 20, 1 i 32000 oraz na 100, 1 i 1000000. Oznacza to, że wstępnie długość ciągu wynosi 20 i można ją zmieniać od 1 do 32 tys., a górne ograniczenie jego elementów jest równe 100 i można je zmieniać od 1 do 1 mln. Zakładamy, że ich dolne ograniczenie wynosi 1. Zauważmy jeszcze, że gdy w polu *ListBox1* nie ma danych, przycisk *Button2* powinien być zablokowany, ponieważ nie ma czego sortować. Zatem ustawiamy jego właściwość *Enabled* na *false*.

Zajmijmy się teraz obsługą zdarzeń tworzonej aplikacji. Na początku, gdy zostanie uruchomiona, powinna uaktywnić generator liczb losowych. Aby tak się działo, definiujemy procedurę obsługi zdarzenia *OnCreate* formularza:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Randomize;
end;
```

Za każdym razem, gdy użytkownik zmieni ustawienia jednego z pól edycji, informacje wyświetlone w polach listy staną się nieaktualne. Należy wtedy obie listy wyczyścić, a ponadto zablokować przycisk sortowania *Button2*. Operacje te umieszczamy w procedurze obsługi zdarzenia *OnChange* definiowanej dla pola *SpinEdit1*, którą przypisujemy również polu *SpinEdit2*. Oto postać tej metody:

```
procedure TForm1.SpinEdit1Change(Sender: TObject);
begin
    ListBox1.Clear;
    ListBox2.Clear;
    Button2.Enabled := false;
end;
```

Pozostały jeszcze do zdefiniowania procedury obsługi zdarzeń *OnClick* obu przycisków. Pierwsza powinna wyczyścić obie listy, wygenerować i wyświetlić ciąg oraz uaktywnić przycisk sortowania, zaś druga posortować i wyświetlić ciąg oraz zablokować przycisk sortowania. Oto robocza wersja tych metod:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ListBox1.Clear;
    ListBox2.Clear;
    // Wygeneruj ciąg i wyświetl go w polu ListBox1
    Button2.Enabled := true;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    // Posortuj ciąg i wyświetl go w polu ListBox2
    Button2.Enabled := false;
end;
```

Znaleźliśmy się w sytuacji, w której musimy zdefiniować tablicę reprezentującą ciąg liczb całkowitych, który ma być generowany i sortowany. Sztuczne reguły deklarowania tablic w języku Pascal uniemożliwiają użycie zmiennej do określania zakresu indeksów, co w przypadku ciągu zmiennej długości prowadzi to nieefektywnego wykorzystania pamięci, ponieważ rozmiar tablicy można jedynie określić „na wyrost”, by mogła pomieścić ciąg potencjalnie najdłuższy. O wiele lepszym rozwiązaniem, dostępnym począwszy od Delphi 4, jest wykorzystanie tzw. **tablic dynamicznych**, deklarowanych bez precyzowania zakresu indeksów [5, 14].

Tablice dynamiczne są zawsze indeksowane od zera, podobnie jak w językach C i C++. Rozmiar (liczbę elementów) tablicy dynamicznej udostępnia funkcja *Length*, a wartości najniższego i najwyższego indeksu funkcje *Low* i *High*. Oczywiście wartością *Low* jest zero, a *High* wartość *Length* pomniejszona o jeden. Gdy zadeklarujemy tablicę dynamiczną, nie zajmuje ona pamięci. Aby przydzielić jej pamięć i zarazem określić rozmiar, korzystamy z procedury *SetLength*, która umożliwia również zmianę rozmiaru tablicy, i to bez utraty zawartości pozostających w niej elementów. Możemy też zwolnić całą pamięć zajmowaną przez tablicę dynamiczną, przypisując jej wartość **nil** (wskaźnik pusty).

Rozważania te prowadzą do zadeklarowania w części prywatnej klasy *TForm1* pola reprezentującego ciąg liczb całkowitych:

```
a: array of integer;
```

a w części publicznej metod generowania tego ciągu i wyświetlania go:

```
procedure GenerujCiąg;
procedure WyświetlCiąg(Lista: TListBox);
```

Puste szkielety obu metod, których wygenerowanie przez Delphi wymuszamy za pomocą klawiszy *Ctrl+Shift+C* (dokańczanie klas), rozbudowujemy do postaci:

```
procedure GenerujCiąg;
var
    k: integer;
begin
    SetLength(a, SpinEdit1.Value);
    for k:=0 to High(a) do
        a[k] := Random(SpinEdit2.Value)+1;
end;
```

```

procedure WyswietlCiag(Lista: TListBox);
var
    k: integer;
begin
    for k:=0 to High(a) do
        Lista.Items.Add(IntToStr(a[k]));
    end;

```

Elementy tablicy  $a$  są numerowane od 0 do  $n - 1$ . Aby uniknąć tej niedogodności, moglibyśmy w metodzie *GenerujCiag* przydzielić jej o jeden element więcej i po prostu nie korzystać z elementu  $a[0]$ , ale takie rozwiązanie byłoby niezbyt eleganckie. Właściwość *Items* komponentu *ListBox* jest listą łańcuchów typu *TStrings*, dlatego dodanie kolejnej pozycji do listy określonej w parametrze metody *WyswietlCiag* wymagało zamiany wartości całkowitej na łańcuch.

Wykorzystując metody *GenerujCiag* i *WyswietlCiag*, możemy uściślić kod metod *Button1Click* i *Button2Click* obsługujących zdarzenie *OnClick* obydwu przycisków aplikacji. I tak, komentarz występujący w pierwszej z nich zastępujemy wywołaniami metod *GenerujCiag* i *WyswietlCiag*:

```

GenerujCiag;
WyswietlCiag(ListBox1);

```

a w drugiej – nowym komentarzem i wywołaniem metody *WyswietlCiag*:

```

// Posortuj ciag
WyswietlCiag(ListBox2);

```

Metoda *Button1Click* została sprecyzowana, a metoda *Button2Click* wymaga dalszego uściślenia. Aplikację w tej postaci możemy uruchomić, zapisując ją uprzednio na dysku, np. moduł formularza w pliku *MainUnit.pas*, a projekt w pliku *Sortowanie.dpr*. Jest oczywiste, że na razie nie będzie ona sortowała ciągu, dlatego zawartości obydwu pól listy będą takie same.

Przejdźmy wreszcie do sformułowania algorytmu *BUBBLE-SORT* w języku Object Pascal. Definicję algorytmu zamieścimy w odrębnym module, co pozwoli na wykorzystanie go w innych aplikacjach, gdy zajdzie taka potrzeba. Dodajemy zatem do aplikacji nowy moduł i przechodzimy do jego rozbudowy.

Zaczynamy od przetłumaczenia na Pascal nieformalnego opisu tablicy  $a[1..n]$  będącej parametrem podprogramu *BUBBLE-SORT*. Skorzystamy z dostępnej, począwszy od Turbo Pascala 7, sposobności umieszczania tzw. **tablic otwartych** (ang. *open array*) w parametrach podprogramów [26]. Podobnie jak w przypadku tablic dynamicznych, nie określa się dla nich zakresu indeksów, które zmieniają się zawsze od zera wzwyż. Zatem elementy tablicy  $a$ , które należy posortować, mają indeksy od 0 do  $n - 1$ . Taka indeksacja pociąga za sobą zmianę numeracji kolej-

nych porównań w pętli wewnętrznej **for**, mianowicie  $p = 0, 1, \dots, n - k - 1$  zamiast  $p = 1, 2, \dots, n - k$ . Uwzględniając funkcję *High*, której wartością jest  $n - 1$ , możemy już łatwo sprecyzować algorytm sortowania bąbelkowego w postaci modułu:

```
unit BubbleUnit;

interface

procedure BubbleSort(var a: array of integer);

implementation

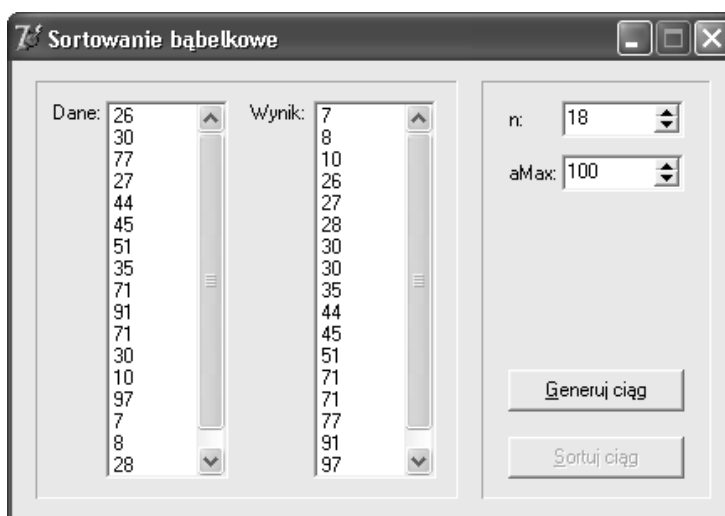
procedure BubbleSort(var a: array of integer);
var
  k, p, tmp: integer;
begin
  for k:=1 to High(a) do
    for p:=0 to High(a)-k do
      if a[p] > a[p+1] then
        begin
          tmp := a[p];
          a[p] := a[p+1];
          a[p+1] := tmp;
        end;
      end;
    end;
  end;
end.
```

Zauważmy, że procedura *BUBBLE-SORT* jest uniwersalna – jej parametrem może być tablica o elementach dowolnego typu, w którym jest określona relacja porządku, np. tablica liczb całkowitych, rzeczywistych, a nawet łańcuchów. Jej implementacja *BubbleSort* nie jest tak elastyczna, ponieważ należało w niej uwzględnić konkretny typ elementów tablicy, tj. *integer*. Jeżeli chcemy sortować tablice różnych typów, możemy utworzyć oddzielną wersję procedury dla każdego z nich. Małym udogodnieniem jest możliwość skorzystania z tzw. **przeciążania** lub **przeładowywania** (ang. *overload*), które zezwala na nadanie tym wszystkim procedurom wspólnej nazwy [5, 14]. Innym rozwiązaniem jest utworzenie jednej wersji procedury wykorzystującej tablicę o elementach typu **wariantowego** (ang. *variant*), w których można przechowywać wartości różnych typów [5, 22]. Sprawność takiej procedury jest jednak dużo mniejsza, gdyż operacje na danych typu wariantowego są bardzo powolne.

Po tej dygresji powróćmy do naszej aplikacji. Aby wykorzystać procedurę *BubbleSort* w module *MainUnit*, rozszerzamy jego listę **uses** o nazwę *BubbleUnit*, a zamiast komentarza w metodzie *Button2Click* umieszczamy jej wywołanie:

```
BubbleSort(a);
```

Rysunek 1.2 przedstawia przykładowy wynik wykonania aplikacji. Uruchamiając aplikację dla większych wartości  $n$  łatwo się przekonać, że czas jej wykonania nie jest jednakowy – wyraźnie się wydłuża, gdy  $n$  rośnie. Użytkownik oczekujący na wynik w przypadku dużych  $n$  może nawet poczuć się zdezorientowany, nie wiedząc, czy program się zawiesił, czy działa nadal.



Rys. 1.2. Efekt wykonania aplikacji sortowania bąbelkowego

Tę niedogodność można naprawić, wyświetlając kursor klepsydry, gdy wykonywane są czasochłonne operacje, i przywracając kursor domyślny, gdy zostaną one zakończone lub przerwane z powodu wyjątku. Właściwym rozwiązaniem jest wykorzystanie bloku **try ... finally** w metodach *Button1Click* i *Button2Click*. Oto poprawiona wersja modułu *MainUnit*, w której przyjęto takie rozwiązanie:

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls, Spin, BubbleUnit;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    ListBox2: TListBox;
    SpinEdit1: TSpinEdit;
    SpinEdit2: TSpinEdit;
```



```
Label1: TLabel;  
Label2: TLabel;  
Label3: TLabel;  
Label4: TLabel;  
Button1: TButton;  
Button2: TButton;  
Bevel1: TBevel;  
Bevel2: TBevel;  
procedure FormCreate(Sender: TObject);  
procedure SpinEdit1Change(Sender: TObject);  
procedure Button1Click(Sender: TObject);  
procedure Button2Click(Sender: TObject);  
private  
  a: array of integer;  
public  
  procedure GenerujCiag;  
  procedure WyswietlCiag(Lista: TListBox);  
end;  
  
var  
  Form1: TForm1;  
  
implementation  
  
{ $R *.dfm }  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  Randomize;  
end;  
  
procedure TForm1.SpinEdit1Change(Sender: TObject);  
begin  
  ListBox1.Clear;  
  ListBox2.Clear;  
  Button2.Enabled := false;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  ListBox1.Clear;  
  ListBox2.Clear;  
  Screen.Cursor := crHourGlass;  
  try  
    GenerujCiag;  
    WyswietlCiag(ListBox1);  
  finally  
    Screen.Cursor := crDefault;  
  end;  
  Button2.Enabled := true;  
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Screen.Cursor := crHourGlass;
    try
        BubbleSort(a);
        WyswietlCiag(ListBox2);
    finally
        Screen.Cursor := crDefault;
    end;
    Button2.Enabled := false;
end;

procedure GenerujCiag;
var
    k: integer;
begin
    SetLength(a, SpinEdit1.Value+1);
    for k:=0 to High(a) do
        a[k] := Random(SpinEdit2.Value)+1;
    end;

procedure WyswietlCiag(Lista: TListBox);
var
    k: integer;
begin
    for k:=0 to High(a) do
        Lista.Items.Add(IntToStr(a[k]));
    end;

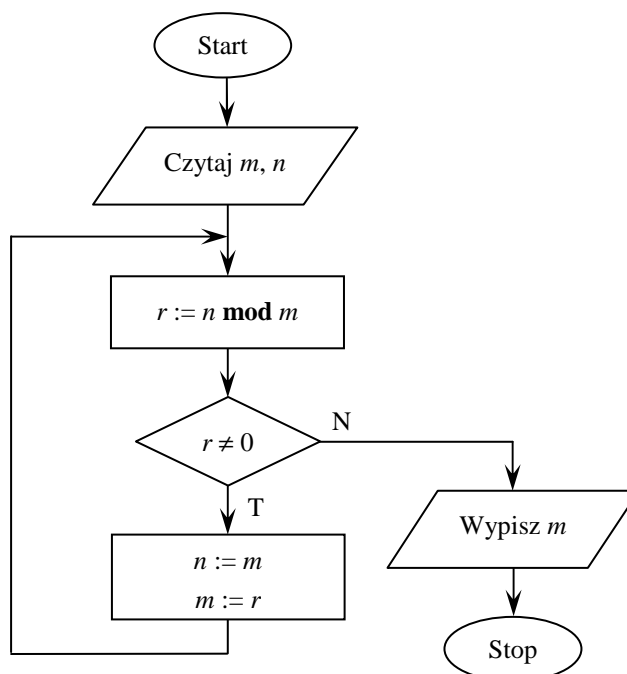
end.
```

Oczywistą zaletą implementacji algorytmów w postaci **programów komputerowych** jest możliwość ich kompilacji i wykonania na komputerze. Taka reprezentacja algorytmów często ułatwia również sprawdzanie ich poprawności. Instrukcje programu są bardziej precyzyjne niż jakiekolwiek inne opisy działań, toteż niektóre błędy znikają automatycznie dzięki właściwościom samego języka programowania, a ponadto część błędów daje się usunąć za pomocą wyszukanych środowisk programistycznych, część zaś na podstawie analizy tekstu programu. Wadą zapisu algorytmów w konkretnym języku programowania jest zależność od swoistych cech tego języka i szczegółów implementacyjnych zaciemniających ich zrozumienie, a także nieczytelność dla osób nieprzygotowanych.

Istnieje jeszcze kilka innych sposobów reprezentowania algorytmów, m.in. graficzne, do których należą **schematy blokowe**, nazywane też **sieciami działań**. Schematy blokowe były kiedyś najbardziej popularną formą zapisu algorytmów, zwłaszcza w okresie dominacji języków programowania Fortran i Cobol. Obecnie ich znaczenie jest dużo mniejsze, głównie za sprawą języków Pascal, C i C++, na bazie których tworzy się intuicyjnie zrozumiałe pseudojęzyki programowania.

Nadal jednak nadają się doskonale do pełnienia funkcji języka pośredniego między zleceniodawcą a programistą.

Schemat blokowy składa się z symboli graficznych, w których opisuje się operacje wykonywane w algorytmie, oraz łączących je pionowych i poziomych linii oznaczanych strzałkami, które określają następstwo tych operacji. Stosuje się kilka kształtów symboli dla rozróżniania rodzajów operacji. Najważniejszymi z nich są: prostokąt (instrukcje przypisania), romb (podjęcie decyzji) i równoległobok (instrukcje wejścia-wyjścia). Przykładowy zapis algorytmu Euklidesa w postaci schematu blokowego, uwzględniającego wczytywanie danych i wypisywanie wyniku, jest pokazany na rysunku 1.3.



**Rys. 1.3.** Schemat blokowy algorytmu Euklidesa

W dalszej części tej książki algorytmy są prezentowane w pseudojęzyku ze względu na przejrzystość takiego zapisu i możliwość łatwego przekształcenia go do programu w pełni zgodnego ze składnią konkretnego języka programowania. Część algorytmów jest również zaprezentowana w postaci kompletnego projektu nadającego się do uruchomienia w środowisku Delphi, co pozwala na obserwację ich funkcjonowania i sprawdzenie efektywności w praktyce. Niektóre algorytmy są opisane w postaci listy kroków.

## 1.4. Pomiar czasu wykonania programu

Czas wykonania algorytmu zakodowanego w programie można zmierzyć za pomocą poleceń systemowych służących do pomiaru czasu. Uzyskany wynik zależy oczywiście od szybkości komputera, systemu operacyjnego, języka programowania, kompilatora i wielu innych czynników. Nie jest więc miarodajny, ale może dać przybliżone wyobrażenie o sprawności algorytmu i dostarczyć danych eksperymentalnych potwierdzających jego szczegółową analizę teoretyczną.

Poniżej jest przedstawiona przykładowa aplikacja konsolowa w Delphi ilustrująca to praktyczne podejście w przypadku sortowania bąbelkowego ciągu liczb całkowitych za pomocą procedury *BubbleSort* omówionej w poprzednim podrozdziale. Czas rozpoczęcia i zakończenia sortowania jest pobierany z systemu za pomocą funkcji *Time*, a liczba milisekund pomiędzy tymi momentami czasowymi jest wyznaczana za pomocą funkcji *MillisecondsBetween* zdefiniowanej w module *DateUtils* (inne rozwiązanie polega na użyciu procedury *DecodeTime* – moduł *SysUtils*). Pomiar czasu jest powtarzany dla ciągów losowych o podwajanej długości  $n = 1000, 2000, 4000, \dots, 64000$ .

```
program CzasSort;  
  
{$APPTYPE CONSOLE}  
  
uses  
    SysUtils, DateUtils, BubbleUnit;  
  
var  
    n, k: integer;  
    a: array of integer;  
    Start: TDateTime;  
  
begin  
    Randomize;  
    WriteLn('  n    Czas ms');  
    WriteLn('-----');  
    n := 1000;  
    while n <= 64000 do  
        begin  
            SetLength(a, n);  
            for k:=0 to High(a) do a[k] := Random(MaxInt);  
            Start := Time;  
            BubbleSort(a);  
            WriteLn(n:5, MillisecondsBetween(Time, Start):8);  
            n := 2*n;  
        end;  
    WriteLn('-----');  
    ReadLn;  
end.
```

Tabela 1.3 przedstawia przykładowe wyniki wykonania tej aplikacji otrzymane dla procesora 667 MHz. Łatwo sprawdzić, że dobrym przybliżeniem ukazującym zależność uzyskanego czasu sortowania od  $n$  jest  $8,4 \times 10^{-6} n^2$  milisekund.

**Tabela 1.3.** Sortowanie bąbelkowe (procesor 667 MHz)

$n$	Czas sortowania
1000	10 milisekund
2000	30 milisekund
4000	130 milisekund
8000	0,5 sekundy
16000	2,1 sekundy
32000	8,6 sekundy
64000	34,5 sekundy

Problemem przy mierzeniu czasu wykonania algorytmu może być zbyt duża szybkość procesora, która może sprawić, że algorytm wykonuje się tylko przez znikomy ułamek sekundy. Narzucającym się wówczas rozwiązaniem jest wykonanie algorytmu w pętli i obliczenie średniego czasu jej wykonania. Z kolei mogą się wtedy pojawić pewne trudności organizacyjne, np. takie jak zapewnienie powtarzalności danych. Przykładowo, wielokrotne sortowanie ciągu wymaga za każdym razem odtworzenia jego pierwotnej postaci (przekopiowania)<sup>6</sup>. Oczywiście, w ostatecznym rozrachunku należy uwzględnić czas tych dodatkowych operacji.

Inny problem wynika z wielozadaniowości systemu operacyjnego [20]. System wielozadaniowy może, w sposób niezauważalny dla użytkownika, przerwać na moment algorytm i zająć się innym zadaniem realizowanym w tle. Ta niezwykle pożyteczna i przez lata oczekiwana funkcja systemu staje się w tym przypadku przeszkodą zaburzającą wynik pomiaru czasu wykonania algorytmu.

## 1.5. Złożoność obliczeniowa algorytmów

Algorytmy ocenia się i porównuje na podstawie rozmaitych, zależnych głównie od okoliczności ich użycia, a niekiedy nawet subiektywnych kryteriów. Zazwyczaj zwraca się uwagę na prostotę i elegancję algorytmów, ich czas działania i zajętość pamięci, a w przypadku algorytmów numerycznych również na dokładność obli-

---

<sup>6</sup> Tablice dynamiczne w Delphi najwygodniej jest kopiować, podobnie jak łańcuchy, za pomocą funkcji *Copy* [5, 22].

czeń. Często wybór określonego algorytmu jest podyktowany kompromisem pomiędzy sprzecznymi ze sobą kryteriami, np. prostotą a efektywnością. Algorytmy proste są łatwiejsze do zrozumienia, prezentacji i weryfikacji, ale czas ich realizacji bywa dłuższy niż algorytmów efektywnych, które z kolei charakteryzują się zwykle wyższym stopniem skomplikowania.

Podstawowymi zasobami niezbędnymi do wykonania każdego algorytmu na komputerze są **czas wykonania** i **wielkość zajmowanej pamięci**, dlatego stojąc przed wyborem jednego spośród wielu algorytmów rozwiązujących postawiony problem, najczęściej bierze się pod uwagę te czynniki. Analiza algorytmu polega na określeniu zasobów komputerowych, jakie są potrzebne do jego wykonania. Dokonując analizy algorytmu należy także uwzględnić jego poprawność, prostotę i użyteczność praktyczną. Analiza kilku algorytmów dla tego samego problemu prowadzi zwykle do wyboru najlepszego pod względem czasu lub pamięci.

Wielkość zasobów komputerowych potrzebnych do wykonania algorytmu określa się mianem **złożoności obliczeniowej algorytmu**. Dla wielu algorytmów czas ich wykonania i wielkość zajmowanej pamięci zwiększa się, gdy wzrasta wielkość zestawu danych wejściowych (wielkość egzemplarza problemu). Dlatego uzasadnione jest traktowanie złożoności obliczeniowej jako funkcji zależnej od **rozmiaru danych** wejściowych, nazywanego również **rozmiarem problemu** lub **zadania**, który jest liczbą całkowitą wyrażającą wielkość zestawu danych (wielkość problemu). Na przykład w problemie sortowania za rozmiar danych przyjmuje się liczbę elementów ciągu wejściowego, a w problemie obliczania wyznacznika macierzy kwadratowej – liczbę jej wierszy lub kolumn.

Pozostaje jeszcze kwestia określenia jednostki miary złożoności obliczeniowej. W przypadku **złożoności pamięciowej** za jednostkę tę przyjmuje się **komórkę pamięci**, którą w zależności od kontekstu może być słowo maszynowe, bajt, bit lub inna jednostka pamięci zajmowanej przez wartość typu prostego. Na przykład dla algorytmów działających na danych typu *real* komórka, począwszy od Delphi 4, zajmuje 8 bajtów. W przypadku **złożoności czasowej** w algorytmie wyróżnia się charakterystyczne dla niego operacje o tej własności, że całość pracy wykonywanej przez niego jest w przybliżeniu proporcjonalna do liczby wykonań tych operacji. Nazywamy je **operacjami dominującymi** algorytmu. Na przykład w algorytmie sortowania bąbelkowego za operację dominującą przyjmuje się porównanie dwóch elementów ciągu wejściowego i ewentualnie ich przestawienie, a w algorytmach obliczania wyznacznika macierzy – operacje arytmetyczne  $+$ ,  $-$ ,  $*$ ,  $/$ . Jednostką miary złożoności czasowej jest **wykonanie jednej operacji dominującej**.

Podstawową zaletą definiowania złożoności czasowej algorytmu jako liczby wykonywanych w nim operacji dominujących jest jej niezależność od szybkości procesora, cech języka programowania użytego do zaimplementowania algorytmu i umiejętności programisty przekładających się na jakość zakodowania go. Złożo-

ność czasowa algorytmu staje się tym samym uniwersalną miarą jego efektywności czasowej, własnością algorytmu zależną tylko od niego i rozmiaru danych.

W rzeczywistości nie jest to do końca prawdą, ponieważ czas wykonania algorytmu może się różnić dla danych o tym samym rozmiarze, jak np. w przypadku sortowania bąbelkowego. Gdy ciąg wejściowy jest posortowany, nie występuje żadne przestawienie jego elementów, a gdy jest odwrotnie uporządkowany, liczba przestawień jest największa. W niezbyt dogłębnych analizach algorytmów sortowania zazwyczaj bierze się pod uwagę tylko porównania elementów.

Dokonyamy teraz analizy złożoności obliczeniowej algorytmu *BUBBLE-SORT*. Ciąg wejściowy zajmuje  $n$  komórek pamięci, a do przestawiania jego elementów potrzebna jest jedna komórka dodatkowa (zmienna *tmp*). Zatem złożoność pamięciowa algorytmu wynosi:

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

Z kolei o złożoności czasowej algorytmu decydują pętle **for**. W każdym wykonaniu pętli wewnętrznej ma miejsce dokładnie jedno porównanie dwóch elementów ciągu, więc w wykonaniu pętli zewnętrznej, opisującym przebieg  $k$ , zachodzi  $n - k$  porównań. Zatem ogólna liczba porównań wykonywanych przez algorytm jest sumą ciągu arytmetycznego wyrazów określających liczbę porównań w kolejnych przebiegach sortowania:

$$T(n) = \sum_{k=1}^{n-1} (n - k) = (n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n.$$

Z powyższego wzoru wynika, że przybliżenie  $8,4 \times 10^{-6} n^2$  ms czasów sortowania zestawionych w tabeli 1.3 dla procesora 667 MHz nie było dziełem przypadku. Z dokładnością do współczynnika proporcjonalności, zależnego m.in. od procesora i kompilatora, czynnikiem decydującym o efektywności czasowej algorytmu sortowania bąbelkowego jest  $n^2$ .

Powyższa analiza złożoności obliczeniowej sortowania bąbelkowego została przeprowadzona dla dowolnych danych rozmiaru  $n$ , toteż  $T(n)$  określa **złożoność pesymistyczną** tego algorytmu. W bardziej dogłębnych analizach wyróżnia się:

- ♦  $W(n)$  – **złożoność pesymistyczną**, definiowaną jako wielkość zasobów komputerowych potrzebnych przy najgorszych danych (ang. *worst case*),
- ♦  $A(n)$  – **złożoność oczekiwaną**, definiowaną jako wielkość zasobów komputerowych potrzebnych przy typowych, przeciętnych statystycznie danych (ang. *avarage case*),
- ♦  $B(n)$  – **złożoność najlepszą**, definiowaną jako wielkość zasobów komputerowych potrzebnych przy najlepszych danych (ang. *best case*).

Jest oczywiste, że dla algorytmu *BUBBLE-SORT* zachodzi równość:

$$W(n) = A(n) = B(n) = T(n).$$

Zanalizujemy ulepszoną wersję algorytmu sortowania bąbelkowego polegającą na zapamiętaniu, czy w trakcie kolejnego przebiegu dokonano jakichś przestawień elementów ciągu. Przebieg, w którym nie nastąpiły żadne przestawienia wskazuje, że następne przebiegi są zbędne i sortowanie można zakończyć. Oto zapis tego algorytmu w pseudokodzie programowania:

```

procedure BUBBLE-SORT2(a[1..n])
  for k:=1 to n-1 do
    stop := true
    for p:=1 to n-k do
      if a[p] > a[p+1] then
        tmp := a[p]
        a[p] := a[p+1]
        a[p+1] := tmp
        stop := false
    if stop then
      return

```

W przypadku najlepszych danych, gdy ciąg jest posortowany, wykonywany jest tylko jeden przebieg sortowania, w którym występuje  $n - 1$  porównań i nie ma żadnych przestawień. W przypadku najgorszych danych, gdy ciąg jest odwrotnie posortowany, wykonywane są wszystkie przebiegi, dokładnie jak w algorytmie niezmodyfikowanym. Wynika stąd, że złożoność czasowa algorytmu w przypadku pesymistycznym i najlepszym wynosi:

$$W(n) = T(n) = \frac{1}{2}n^2 - \frac{1}{2}n$$

$$B(n) = n - 1$$

Analiza średniego przypadku złożoności czasowej algorytmu *BUBBLE-SORT2* prowadzi do skomplikowanych obliczeń i rozczarowujących wyników [17].

## 1.6. Przeszukiwanie sekwencyjne i binarne

Zajmiemy się teraz problemem **przeszukiwania**, nazywanego też problemem **wyszukiwania**. Jego specyfikacja jest następująca:

**Dane:** Ciąg  $n$  liczb  $a_1, a_2, \dots, a_n$  i element  $x$ .

**Wynik:** Indeks  $k$  taki, że  $x = a_k$  lub  $-1$ , gdy  $x$  nie występuje w tym ciągu.



Z uwagi na przyjętą w Delphi numerację elementów tablic otwartych i dynamicznych od zera wzwyż przyjęliśmy, że gdy szukany element  $x$  nie występuje w ciągu, wynikiem przeszukiwania jest  $-1$  zamiast  $0$ .

Najprostszy algorytm przeszukiwania, nazywany **przeszukiwaniem sekwencyjnym** (ang. *sequential search*), polega na przeglądaniu kolejnych elementów ciągu i kończy się, gdy poszukiwany element zostanie znaleziony lub cały ciąg zostanie przeszukany. Oto jedna z prostszych wersji jego zapisu w pseudojęzyku programowania:

```
function SEQUENTIAL-SEARCH(a[1..n], x)
  for k:=1 to n do
    if x = a[k] then
      return k
  return -1
```

Operacjami dominującymi w tym algorytmie są porównania wartości  $x$  z elementami  $a[k]$ . Ich liczba jest równa liczbie wykonania pętli **for**. Najlepszy przypadek danych zachodzi wtedy, gdy  $x$  jest pierwszym elementem ciągu, a najgorszy, gdy jest ostatnim elementem ciągu lub w nim nie występuje. Zatem:

$$W(n) = T(n) = n$$

$$B(n) = 1$$

Spróbujmy zanalizować przypadek średni statystycznie. W tym celu założmy, że prawdopodobieństwo zdarzenia, że  $x$  występuje w ciągu, wynosi  $p$ , oraz że jeśli w nim występuje, to prawdopodobieństwo znalezienia  $x$  na którejkolwiek pozycji od  $1$  do  $n$  jest jednakowe. Wynika stąd, że prawdopodobieństwo zdarzenia, że  $x$  znajduje się na pozycji  $k$ , wynosi  $p/n$ , a prawdopodobieństwo zdarzenia, że nie występuje w ciągu, wynosi  $1 - p$ . Zatem prawdopodobieństwo  $p_k$  zdarzenia  $\omega_k$ , że algorytm wykonuje  $k$  porównań, wynosi:

$$p_k = \begin{cases} p/n & (k = 1, 2, \dots, n-1) \\ p/n + (1-p) & (k = n) \end{cases}$$

Niech  $X_n$  oznacza zmienną losową [7, 10, 21], której wartościami są liczby porównań wykonywanych przez algorytm dla danych rozmiaru  $n$ :

$$X_n(\omega_k) = k \quad (k = 1, 2, \dots, n).$$

Jej wartość oczekiwana (średnia statystycznie) wyraża się wzorem:

$$\begin{aligned}
 E(X_n) &= \sum_{k=1}^n X_n(\omega_k) p_k = \sum_{k=1}^n k \cdot \frac{p}{n} + n \cdot (1-p) = \frac{p}{n} \sum_{k=1}^n k + n \cdot (1-p) = \\
 &= \frac{p}{n} \cdot \frac{n \cdot (n+1)}{2} + n \cdot (1-p) = n \cdot \left(1 - \frac{p}{2}\right) + \frac{p}{2}
 \end{aligned}$$

Ostatecznie więc średnia liczba porównań wykonywanych przez algorytm przeszukiwania sekwencyjnego przy założeniu, że prawdopodobieństwo wystąpienia poszukiwanego elementu w ciągu wynosi  $p$ , określa zależność:

$$A(n) = E(X_n) = n \cdot \left(1 - \frac{p}{2}\right) + \frac{p}{2}.$$

Zwróćmy jeszcze uwagę na trzy szczególne przypadki uzyskanego wyniku i ich interpretację:

- ♦ Jeżeli  $p = 1$ , czyli gdy wiadomo, że poszukiwany element znajduje się w ciągu, to  $A(n) = (n+1)/2$ . Oznacza to, że średnio przeszukiwana jest połowa ciągu.
- ♦ Jeżeli  $p = 1/2$ , czyli gdy fakt, że szukany element znajduje się w ciągu, można przewidywać, rzucając monetą, to  $A(n) = (3n+1)/4$ . Oznacza to, że średnio przeszukiwane jest około  $3/4$  ciągu.
- ♦ Jeżeli  $p$  jest bliskie zero, czyli gdy jest mało prawdopodobne, by poszukiwany element znajdował się w ciągu, to  $A(n)$  jest bliskie  $n$ . Oznacza to, że średnio trzeba przeglądać cały ciąg.

Zazwyczaj problemem przy analizowaniu oczekiwanej złożoności czasowej algorytmu jest określenie średniego statystycznie przypadku danych. Najczęściej zakłada się, że wszystkie dane są jednakowo prawdopodobne, ale w praktyce może to wyglądać inaczej. Z tego względu w dalszej części niniejszej książki będziemy głównie koncentrować się na przypadku pesymistycznym.

Powróćmy do problemu przeszukiwania i załóżmy, że elementy ciągu są uporządkowane niemalejąco, co w praktyce jest częstym przypadkiem. Algorytm przeszukiwania sekwencyjnego można wtedy również stosować, ale o wiele bardziej efektywny jest algorytm **przeszukiwania binarnego** (ang. *binary search*), który wykorzystuje uporządkowanie ciągu. Jego charakterystyczną cechą jest porównywanie poszukiwanego elementu  $x$  ze środkowym elementem ciągu. W wyniku tego porównania wiadomo, w której połowie ciągu kontynuować poszukiwanie, powtarzając porównanie  $x$  ze środkowym elementem wybranej połowy itd.

Aby uściślić rozumowanie, rozpatrzmy ciąg elementów o indeksach od  $i$  do  $j$ . Na początku rozpatrujemy cały ciąg, przyjmując  $i = 1, j = n$ . Dla jego środkowego elementu, o indeksie  $k = (i + j)/2$ , wystąpi jeden z trzech przypadków:

- $x = a[k]$  – element  $x$  został znaleziony na pozycji  $k$ , więc należy zakończyć algorytm;
- $x < a[k]$  – element  $x$  może wystąpić tylko w lewej połowie ciągu, więc należy rozpatrzeć ją, przyjmując  $j = k - 1$ ;
- $x > a[k]$  – element  $x$  może wystąpić tylko w prawej połowie ciągu, więc należy rozpatrzeć ją, przyjmując  $i = k + 1$ .

Jeżeli po wykonaniu kolejnego kroku zachodzi nierówność  $i > j$ , poszukiwanie kończy się niepomyślnie, a jego wynikiem jest  $-1$ . Rozważania te prowadzą do następującego zapisu algorytmu w pseudojęzyku programowania:

```
function BINARY-SEARCH(a[1..n], x)
  i := 1
  j := n
  while i ≤ j do
    k := (i+j) div 2
    if x = a[k] then
      return k
    else if x < a[k] then
      j := k-1
    else
      i := k+1
  return -1
```

Zanalizujmy złożoność czasową tego algorytmu, wyrażając ją jako liczbę wykonania pętli **while** zależną od  $n$ . Zauważmy, że z wyjątkiem przypadku, gdy element  $x$  zostanie znaleziony, w każdym wykonaniu pętli mają miejsce dwa porównania. Zatem wydaje się oczywiste, że przy dokładnym przyrównywaniu efektywności czasowej algorytmu przeszukiwania binarnego do sekwencyjnego należy pomnożyć uzyskany wynik przez 2. Czytelnik znający język assemblera być może będzie miał zastrzeżenie. Wszak można w implementacji algorytmu w tym języku umieścić tylko jedno porównanie, które ustawia kod warunku, umożliwiając wykonanie odpowiedniego skoku do jednego z trzech fragmentów kodu odpowiadających opisanym wyżej przypadkom. Tak czy owak, nie ma znaczenia, czy za operację dominującą przyjmiemy pojedyncze czy podwójne porównanie, bowiem złożoność czasowa algorytmu wyraża czas jego wykonania w kategoriach proporcjonalności. Możemy więc przyjąć, że w każdym wykonaniu pętli **while** wykonywana jest tylko jedna operacja dominująca.

Zatem dla  $n = 1$  wykonywana jest jedna operacja dominująca, zaś dla  $n > 1$  problem zostaje zredukowany do rozmiaru dwa razy mniejszego, gdy wykonanie operacji dominującej nie zakończy się sukcesem, czyli gdy poszukiwany element nie jest środkowym elementem rozpatrywanego fragmentu ciągu. Złożoność czasową algorytmu można więc wyrazić w postaci następującej rekurencji:

$$T(n) = \begin{cases} 1 & (n = 1) \\ T\left(\frac{n}{2}\right) + 1 & (n > 1) \end{cases} \quad (1.6)$$

Analiza ta opiera się na intuicji i prowadzi do niezręcznego dzielenia długości ciągu  $n$  przez 2, gdy jest ona liczbą nieparzystą. W celu uściślenia rozważań i wyjaśnienia wątpliwości rozpatrzmy oddzielnie  $n$  parzyste i nieparzyste.

W przypadku  $n$  parzystego istnieje  $m$  całkowite takie, że  $n = 2m$ . Element środkowy ciągu ma wtedy indeks  $k = \lfloor (1 + 2m)/2 \rfloor = m^7$ . Ciąg jest więc dzielony na dwa podciągi, których elementy są numerowane od 1 do  $m - 1$  i od  $m + 1$  do  $2m$ . Pierwszy z nich ma długość  $m - 1$ , a drugi  $m$ , zatem:

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

Z kolei w przypadku  $n$  nieparzystego istnieje  $m$  całkowite takie, że  $n = 2m + 1$ . Element środkowy ciągu ma wówczas indeks  $k = (1 + 2m + 1)/2 = m + 1$ , więc ciąg jest dzielony na podciągi, których elementy są numerowane od 1 do  $m$  i od  $m + 2$  do  $2m + 1$ . Obydwa mają po  $m$  elementów, zatem:

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

Tak więc, równanie (1.6), opisujące pesymistyczny czas działania algorytmu przeszukiwania binarnego, ma właściwie postać:

$$T(n) = \begin{cases} 1 & (n = 1) \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 & (n > 1) \end{cases} \quad (1.7)$$

W praktyce często pomija się szczegóły techniczne przy formułowaniu równań rekurencyjnych, czego przykładem jest przeprowadzona wyżej nieformalna analiza prowadząca do równania (1.6). Często też rozwiązuje się rekurencję, podstawiając zamiast  $n$  inną zmienną, a następnie wnioskuje się, że uzyskane rozwiązanie jest prawdziwe dla dowolnych  $n$ . Metodę podstawienia zmiennej zastosujemy w przy-

---

<sup>7</sup> Wartością  $\lfloor x \rfloor$  jest największa liczba całkowita nie większa niż  $x$ , a wartością  $\lceil x \rceil$  najmniejsza liczba całkowita nie mniejsza niż  $x$ . Funkcje te są nazywane *podłogą* i *sufitem*. Ich własności są szczegółowo omówione w książce [10].

padku równania (1.7). Przyjmijmy mianowicie, że  $n$  jest potęgą dwójki:  $n = 2^m$ . Mamy wtedy<sup>8</sup>:

$$\begin{aligned} T(n) &= T(2^m) = T(2^{m-1}) + 1 = T(2^{m-2}) + 2 = \dots = T(2^0) + m = \\ &= T(1) + m = m + 1 = \log n + 1 \end{aligned}$$

Zatem dla  $n$  będących potęgami dwójki otrzymujemy:

$$T(n) = \log n + 1. \quad (1.8)$$

W celu wykazania, że uzyskany wynik jest prawdziwy dla dowolnego  $n$ , można skorzystać z tzw. metody rekurencji uniwersalnej [7]. Posłużymy się jednak indukcją matematyczną [19, 21], traktując powyższe podstawienie zmiennej tylko jako wygodny sposób odgadnięcia oszacowania dla  $T(n)$ , którego prawdziwość należy udowodnić. Pokażemy, że dla dowolnych  $n$  zachodzi wzór:

$$T(n) = \lfloor \log n \rfloor + 1, \quad (1.9)$$

który dla potęg dwójki pokrywa się ze wzorem (1.8).

Sprawdzenie, że warunek początkowy indukcji matematycznej jest prawdziwy, czyli że wzór (1.9) zachodzi dla  $n = 1$ , sprowadza się do wykorzystania pierwszej części równości (1.7):

$$T(1) = 1 = \lfloor 0 \rfloor + 1 = \lfloor \log 1 \rfloor + 1.$$

Aby wykazać krok indukcyjny zakładamy, że wzór (1.9) jest spełniony dla wszystkich dodatnich liczb całkowitych mniejszych lub równych  $n$ . Mamy udowodnić, że wzór ten jest również spełniony dla  $n + 1$ . Korzystając z drugiej części równości (1.7) i łatwej do sprawdzenia nierówności:

$$\left\lfloor \frac{n+1}{2} \right\rfloor \leq n,$$

na podstawie której możemy skorzystać z założenia indukcyjnego, otrzymujemy:

$$\begin{aligned} T(n+1) &= T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + 1 = \left\lfloor \log \left\lfloor \frac{n+1}{2} \right\rfloor \right\rfloor + 1 + 1 \leq \left\lfloor \log \frac{n+1}{2} \right\rfloor + 2 = \\ &= \lfloor \log(n+1) - 1 \rfloor + 2 = \lfloor \log(n+1) \rfloor - 1 + 2 = \lfloor \log(n+1) \rfloor + 1 \end{aligned}$$

---

<sup>8</sup> Podstawą logarytmu jest 2.

Zamiast równości uzyskaliśmy słabą nierówność, ale ponieważ  $T(n)$  ma być górnym oszacowaniem złożoności czasowej algorytmu, krok indukcyjny został wykazany. Zatem z zasady indukcji wnioskujemy, że wzór (1.9) jest prawdziwy dla każdej dodatniej liczby całkowitej  $n$ .

Dokonajmy teraz krótkiego porównania efektywności czasowej obydwu algorytmów przeszukiwania. Zazwyczaj uważa się, że jeden algorytm jest lepszy od drugiego, jeżeli jego złożoność obliczeniowa jest funkcją niższego rzędu względem rozmiaru danych  $n$ . W przypadku przeszukiwania sekwencyjnego jest nią funkcja liniowa, a w przypadku binarnego funkcja logarytmiczna. Zatem lepszym powinien być algorytm przeszukiwania binarnego, co w pełni potwierdzają wyniki przedstawione w tabeli 1.4.

**Tabela 1.4.** Złożoność czasowa algorytmów przeszukiwania

$n$	Sekwencyjne	Binarne
10	10	4
100	100	7
1 000	1 000	10
10 000	10 000	14
100 000	100 000	17
1 000 000	1 000 000	20

Algorytm przeszukiwania sekwencyjnego jest prosty, ale czas jego wykonania jest proporcjonalny do rozmiaru danych. Oznacza to, że dwukrotne zwiększenie liczby elementów ciągu spowoduje dwukrotne wydłużenie czasu przeszukiwania. Algorytm przeszukiwania binarnego wymaga posortowanych danych, ale w każdej jego iteracji liczba rozpatrywanych elementów ciągu jest redukowana o połowę. Dwukrotne zwiększenie liczby elementów spowoduje jedynie zwiększenie liczby iteracji o jeden, a więc nieznaczne wydłużenie czasu przeszukiwania. Pomijając małe rozmiary danych, dla których przeszukiwanie sekwencyjne działa wystarczająco szybko, przeszukiwanie binarne jest znacznie wydajniejsze.

## 1.7. Notacja asymptotyczna

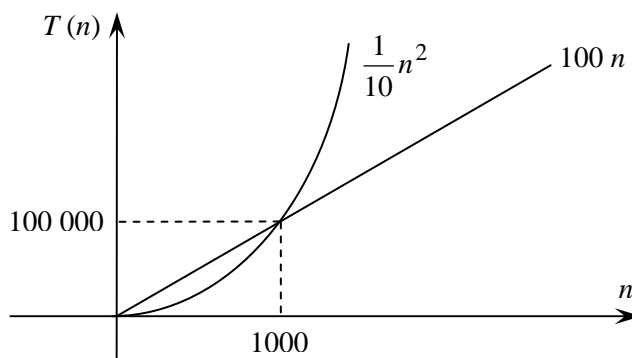
Złożoność czasowa algorytmu, określona w wyniku analizy teoretycznej jako funkcja zależna od rozmiaru danych  $n$ , zawiera szczególnie ważną informację zwaną jej **rzędem wielkości**. Mówimy na przykład, że złożoność czasowa sortowania bąbelkowego jest rzędu  $n^2$ , przeszukiwania sekwencyjnego rzędu  $n$ , a przeszukiwania binarnego rzędu  $\log n$ . Rząd wielkości złożoności czasowej algorytmu

charakteryzuje jej zachowanie asymptotyczne, gdy  $n$  rośnie do nieskończoności. Faktyczny czas wykonania implementacji komputerowej algorytmu (programu) różni się od złożoności wyliczonej teoretycznie, a właściwie od jej rzędu wielkości, współczynnikiem proporcjonalności zależnym od architektury komputera, szybkości procesora, języka programowania, kompilatora, zdolności programisty i wielu innych czynników. Potwierdzeniem tego są przytoczone w tabeli 1.3 (podrozdział 1.4) wyniki pomiaru czasu sortowania bąbelkowego.

Rząd wielkości złożoności czasowej wpływa najbardziej na czas wykonania algorytmu, toteż na jego podstawie porównuje się złożoność różnych algorytmów. Z pewnością można uważać, że dla dostatecznie dużych  $n$  bardziej wydajny jest algorytm, którego rząd wielkości jest niższy. Nie oznacza to jednak, że algorytmy o wyższym rzędzie nie są przydatne. Rozważmy dla przykładu dwa hipotetyczne algorytmy o złożoności  $100 \cdot n$  i  $\frac{1}{10} \cdot n^2$ , w których operacje dominujące zajmują tyle samo czasu (zob. rys. 1.4). Z nierówności:

$$100n > \frac{1}{10}n^2$$

wynika, że dla  $n$  mniejszych od 1000 bardziej wydajny jest algorytm o złożoności kwadratowej, a więc o wyższym rzędzie. Gdy jednak  $n$  wzrasta do nieskończoności, czynnikiem decydującym o efektywności algorytmu staje się rząd wielkości złożoności czasowej, przez co lepszy jest algorytm o złożoności liniowej.



**Rys. 1.4.** Złożoność czasowa dwóch algorytmów

W analizach teoretycznych złożoności obliczeniowej algorytmu zazwyczaj ignoruje się dla dużych rozmiarów danych współczynnik proporcjonalności przy składniku o najwyższym rzędzie i składniki niższych rzędów, pozostawiając sam rząd wielkości czasu działania algorytmu lub jego zapotrzebowania na pamięć. Do

precyzyjnego sformułowania stwierdzenia, że dla dostatecznie dużych rozmiarów danych istotny jest jedynie rząd wielkości złożoności obliczeniowej algorytmu, używa się tzw. **notacji asymptotycznej**, zwanej **notacją  $O$**  (ang. *O-notation*). Mówimy np., że złożoność czasowa algorytmu przeszukiwania binarnego jest  $O(\log n)$ , co oznacza, że w miarę zwiększania się rozmiaru danych  $n$  czas wykonania algorytmu jest proporcjonalny do wartości nie większej niż  $\log n$ .

Aby wyrazić ściśle, czym jest notacja  $O$ , podamy definicję tego pojęcia. Zakładamy, że występujące w niej funkcje są określone w dziedzinie liczb naturalnych, a ich wartości należą do zbioru nieujemnych liczb rzeczywistych.

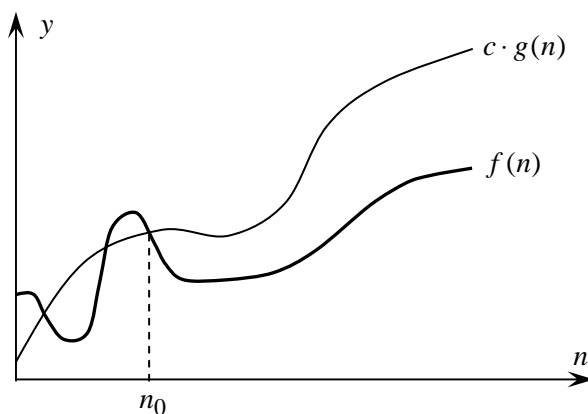
**Definicja.** Mówimy, że funkcja  $f$  jest **co najwyżej rzędu  $g$** , co zapisujemy jako:

$$f(n) = O(g(n)),$$

jeżeli istnieje stała rzeczywista  $c > 0$  i stała naturalna  $n_0$  takie, że dla każdej liczby naturalnej  $n \geq n_0$  zachodzi nierówność:

$$f(n) \leq c \cdot g(n).$$

Notacja  $O$  daje **górne ograniczenie asymptotyczne**  $g(n)$  ciągu  $f(n)$  z dokładnością do stałego współczynnika  $c$  (por. rys. 1.5). W praktyce najczęściej  $f(n)$  oznacza złożoność czasową algorytmu, natomiast  $g(n)$  prosty ciąg o znanej szybkości wzrostu, taki jak  $\log n$ ,  $n$ ,  $n \log n$ ,  $n^2$  itd.



**Rys. 1.5.** Interpretacja geometryczna notacji  $f(n) = O(g(n))$



Rozpatrzmy dla przykładu funkcję:

$$T(n) = 5n^3 + 2n^2 + 7.$$

Jej najbardziej znaczącym czynnikiem jest  $5n^3$ , co pozwala przypuszczać, że:

$$T(n) = O(n^3).$$

Istotnie, dla  $n \geq 3$  zachodzi nierówność  $2n^2 + 7 < n^3$ . Jeżeli więc przyjmiemy  $n_0 = 3$  i  $c = 6$ , otrzymamy:

$$T(n) < 5n^3 + n^3 = 6n^3 = O(n^3).$$

Najważniejsze rodzaje złożoności czasowej większości użytecznych algorytmów są przedstawione w tabeli 1.5. Jako przykłady uwzględnione są w niej omówione w tym rozdziale algorytmy przeszukiwania binarnego i sekwencyjnego oraz sortowania bąbelkowego. Rodzaj ich złożoności wynika natychmiast z równań określających postać funkcji  $T(n)$ . W następnym podrozdziale udowodnimy też, że algorytm Euklidesa znajdowania największego wspólnego dzielnika  $NWD(n, m)$  dwóch dodatnich liczb naturalnych  $n$  i  $m$  ma złożoność  $O(\log m)$ .

**Tabela 1.5.** Najważniejsze rodzaje złożoności czasowej algorytmów

$n$	Rodzaj złożoności	Przykład
$O(1)$	Stała	Dostęp do elementu tablicy
$O(\log n)$	Logarytmiczna	Przeszukiwanie binarne
$O(n)$	Liniowa	Przeszukiwanie sekwencyjne
$O(n \log n)$	Liniowo-logarytmiczna	Sortowanie kopcowe
$O(n^2)$	Kwadratowa	Sortowanie bąbelkowe
$O(n^3)$	Sześcienna	Mnożenie macierzy
$O(2^n)$	Wykładnicza	Algorytm plecakowy
$O(n!)$	Wykładnicza	Generowanie permutacji

Złożoność  $O(1)$  oznacza, że czas wykonania algorytmu jest stały, czyli nie zależy od rozmiaru danych. Złożoność kwadratowa i sześcienna są szczególnymi przypadkami tzw. **złożoności wielomianowej** postaci  $O(p(n))$ , gdzie  $p(n)$  jest wielomianem. Algorytm o wielomianowej złożoności czasowej nazywamy krótko

**algorytmem wielomianowym.** Zauważmy, że chociaż funkcja  $n \log n$  nie jest wielomianem zależnym od  $n$ , to algorytm o złożoności  $O(n \log n)$  można traktować jak algorytm wielomianowy, ponieważ

$$n \log n < n^2.$$

Ogólnie przyjmuje się, że problem informatyczny nie jest trudny, gdy można go rozwiązać za pomocą algorytmu wielomianowego. Algorytmy o złożoności wykładniczej nie są algorytmami wielomianowymi. Ich złożoność rośnie tak szybko, że mogą być one wykonywane jedynie dla małych rozmiarów danych. Przykład takiego algorytmu, obliczającego wyznacznik macierzy kwadratowej, został podany w podrozdziale 1.1. Z równości (1.2) wynika, że złożoność czasowa algorytmu wynosi  $O(n \cdot n!)$ , przez co jest on nierealizowalny nawet dla niewielkich  $n$ .

Notacja asymptotyczna  $O$  nie jest jedynym środkiem wyrażania funkcyjnej złożoności obliczeniowej algorytmów. Oprócz niej używa się notacji  $\Omega$  i  $\Theta$ , które dają dolne i dokładne ograniczenie asymptotyczne badanej funkcji [3, 7, 8, 19].

## 1.8. Rekurencja a iteracja

Algorytm, który wywołuje sam siebie bezpośrednio lub pośrednio, nazywamy **rekurencyjnym**. Zazwyczaj opis algorytmu za pomocą rekurencji jest bardziej przejrzysty i zwięzły niż bez niej. Aby algorytm rekurencyjny działał poprawnie, ciąg zawartych w nim wywołań rekurencyjnych powinien być zawsze skończony. Oznacza to, że wywołania rekurencyjne powinny być uzależnione od warunku, który w pewnym momencie przestaje być spełniony.

Rozpatrzmy ponownie algorytm Euklidesa znajdowania największego wspólnego dzielnika  $NWD(n, m)$  dwóch liczb całkowitych dodatnich  $m$  i  $n$ . Równość (1.5) możemy zapisać w postaci:

$$NWD(n, m) = NWD(m, n \bmod m), \quad (1.10)$$

która określa, że liczby  $n$  i  $m$  oraz  $m$  i  $n \bmod m$  mają ten sam największy wspólny dzielnik. Korzystając z niej, możemy przekształcić algorytm iteracyjny *EUCLID* do następującej postaci:

```
function EUCLID2(n, m)
  if n mod m = 0 then
    return m
  else
    return EUCLID2(m, n mod m)
```

Algorytm *EUCLID2* jest rekurencyjny, bo wywołuje sam siebie, gdy reszta z dzielenia  $n$  przez  $m$  jest różna od zera. Jego działanie polega na rekurencyjnym stosowaniu równania (1.10). Każde wywołanie rekurencyjne powoduje wykonanie algorytmu dla nowej pary wartości  $m$  i  $n \bmod m$  przekazywanych jako parametry  $n$  i  $m$ . Ciąg wywołań rekurencyjnych jest skończony, gdyż wartości obydwu parametrów systematycznie maleją, aż w końcu drugi z nich stanie się dzielnikiem pierwszego (w skrajnym przypadku równym 1), a zarazem największym wspólnym dzielnikiem początkowych wartości  $n$  i  $m$ .

Implementacja tej wersji algorytmu Euklidesa w języku Object Pascal ma postać następującej funkcji rekurencyjnej:

```
function Euclid2(n, m: integer): integer;  
begin  
  if n mod m = 0 then  
    Result := m  
  else  
    Result := Euclid2(m, n mod m);  
end;
```

Jeżeli przyjrzeć się algorytmom *EUCLID* i *EUCLID2*, a zwłaszcza implementacjom komputerowym *Euclid* i *Euclid2*, wersje rekurencyjne wydają się jaśniejsze i prostsze. Należy jednak pamiętać, że chociaż rekurencja często upraszcza strukturę algorytmów, jej realizacja na komputerze powoduje zwiększenie złożoności pamięciowej, ponieważ z każdym wywołaniem podprogramu wiąże się zarezerwowanie obszaru pamięci, zwanego **ramką stosu** (ang. *stack frame*), w której są pamiętane wartości parametrów, zmiennych lokalnych i adres miejsca, do którego ma nastąpić powrót po zakończeniu wywołania. To dodatkowe obciążenie pamięci może być w przypadku podprogramów rekurencyjnych wyraźnie odczuwalne, ponieważ dla każdego nowego wywołania, przy niezakończonych wywołaniach poprzednich, do ramek zarezerwowanych poprzednio dołączana jest nowa ramka. Ważne jest więc, by głębokość wywołań rekurencyjnych była względnie mała.

Zauważmy, że liczba wywołań rekurencyjnych w algorytmie *EUCLID2* jest równa liczbie wykonania pętli **while** w algorytmie *EUCLID*. Wynika stąd wniosek, że złożoności czasowe obu algorytmów są takie same, oczywiście z dokładnością do pewnego współczynnika proporcjonalności. Zatem złożoność czasową algorytmu Euklidesa możemy wyznaczyć, obliczając liczbę wywołań rekurencyjnych w jego wersji rekurencyjnej. Wyrazimy ją jako funkcję  $T(m)$  wywołań rekurencyjnych zależną od parametru  $m$ .

Jeżeli  $m = 1$ , nie ma wywołań rekurencyjnych, czyli  $T(1) = 0$ . Jeżeli  $m > 1$ , drugim parametrem w pierwszym wywołaniu rekurencyjnym jest

$$m_1 = n \bmod m.$$

Może zdarzyć się przypadek, że zachodzi nierówność  $m_1 \leq m/2$ . W przypadku przeciwnym, gdy zachodzi nierówność  $m_1 > m/2$ , drugim parametrem w drugim wywołaniu rekurencyjnym jest

$$m_2 = m \bmod m_1,$$

a wtedy

$$m_2 = m \bmod m_1 = m - (m \operatorname{div} m_1) \cdot m_1 = m - 1 \cdot m_1 = m - m_1 < m/2.$$

Wynika stąd, że w dwóch kolejnych wywołaniach rekurencyjnych drugi parametr zmniejsza się co najmniej dwukrotnie. Rozumowanie to prowadzi do następującej rekurencji:

$$T(m) = \begin{cases} 0 & (m = 1) \\ T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + 2 & (m > 1) \end{cases} \quad (1.11)$$

W celu rozwiązania tego równania zastosujemy metodę podstawienia zmiennej. Przyjmując, że  $m$  jest potęgą dwójki, tj.  $m = 2^k$ , otrzymujemy:

$$\begin{aligned} T(m) &= T(2^k) = T(2^{k-1}) + 2 = T(2^{k-2}) + 2 \cdot 2 = T(2^{k-3}) + 2 \cdot 3 = \dots \\ &= T(2^0) + 2k = 2 \log m \end{aligned}$$

Stąd można łatwo udowodnić za pomocą indukcji matematycznej (zob. ćw. 1.13), podobnie jak w przypadku przeszukiwania binarnego, że dla dowolnych  $m$

$$T(m) = 2 \lfloor \log m \rfloor. \quad (1.12)$$

Równość (1.12) oznacza, że algorytm Euklidesa jest efektywny, ma bowiem logarytmiczną złożoność czasową  $O(\log m)$ .

Zwróćmy jeszcze uwagę na dwie kwestie dotyczące równoważności obu wersji algorytmu Euklidesa. Po pierwsze, algorytm rekurencyjny jest nieco wolniejszy, ponieważ w każdym wywołaniu rekurencyjnym tworzy nową ramkę stosu, a w momencie zakończenia wywołania likwiduje ją. Różnica przejawia się nie w wyższym rzędzie jego złożoności czasowej, lecz w jej większym współczynniku proporcjonalności. Po drugie, zupełnie inna jest złożoność pamięciowa obu algorytmów. W przypadku wersji iteracyjnej wynosi ona  $O(1)$ , a w przypadku wersji rekurencyjnej  $O(\log m)$ , podobnie jak złożoność czasowa. Dysproporcja pomiędzy  $m$  a  $\log m$  jest jednak zbyt ogromna, by stała się przyczyną niechęci do korzystania z algorytmu rekurencyjnego lub uznania go za nieodpowiedni.

Dociekliwemu Czytelnikowi być może nie podoba się, że w algorytmie rekurencyjnym dwukrotnie oblicza się resztę z dzielenia wartości  $n$  przez  $m$ . Tę usterkę można usunąć, zwiększając o jeden głębokość wywołań rekurencyjnych:

```
function EUCLID3(n, m)
  if m = 0 then
    return n
  else
    return EUCLID3(m, n mod m)
```

Zauważmy, że po tej modyfikacji algorytm działa dla dowolnych nieujemnych liczb całkowitych  $m$  i  $n$ , z których przynajmniej jedna jest dodatnia. Istotnie, jeżeli  $m = 0$ , to wynikiem jest od razu wartość  $n$ , a jeżeli  $n = 0$ , czyli gdy  $m > 0$ , to ma miejsce jedno wywołanie rekurencyjne dla pary wartości  $m$  i  $0$ , a wówczas wynikiem jest wartość  $m$  przekazana jako parametr  $n$ .

Rekurencję można zawsze wyeliminować, zastępując ją iteracją. Należy to czynić w przypadkach uzasadnionych, np. gdy nastąpi znaczny wzrost szybkości wykonania algorytmu. Na pewno eliminacja rekurencji jest niewskazana, gdy nie wnosi nic nowego oprócz pogorszenia czytelności, ponieważ zazwyczaj wymaga jawnej obsługi ramek stosu, przez co utrudnia zarówno sam proces przekształcania algorytmu do postaci iteracyjnej, jak i zrozumienie go. Szczególnie prosta jest eliminacja tzw. **rekurencji końcowej** lub **ogonowej** (ang. *tail recursion*), w której wywołanie rekurencyjne jest ostatnią instrukcją algorytmu [1, 22]. Istnieją nawet kompilatory, które to robią automatycznie.

Algorytm *EUCLID3* zawiera rekurencję końcową. Jego odpowiednik iteracyjny ma postać następującą:

```
function EUCLID4(n, m)
  while m ≠ 0 do
    r := n mod m
    n := m
    m := r
  return n
```

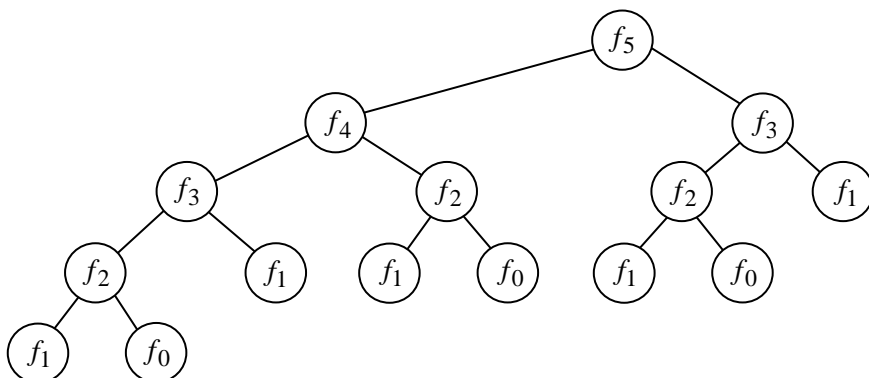
Naturalną tendencją przy rozwiązywaniu problemu zdefiniowanego rekurencyjnie jest zastosowanie rekurencji. Aby przekonać się, że takie podejście niekiedy jest niewłaściwe, zajmijmy się problemem wyznaczania kolejnych wyrazów ciągu Fibonacciego. Oto jego definicja:

$$f_n = \begin{cases} 0 & (n = 0) \\ 1 & (n = 1) \\ f_{n-1} + f_{n-2} & (n \geq 2) \end{cases}$$

Warto wspomnieć, że liczby Fibonacciego mają liczne zastosowania w informatyce i matematyce, a nawet występują w przyrodzie [10]. Zważywszy na prostotę ich rekurencyjnej definicji, oczywiste wydaje się zaproponowanie następującego algorytmu obliczania wartości  $f_n$ :

```
function FIB(n)
  if n ≤ 1 then
    return n
  else
    return FIB(n-1) + FIB(n-2)
```

Algorytm jest rekurencyjny. Każde jego wywołanie dla  $n \geq 2$  uaktywnia dwa dalsze wywołania rekurencyjne. W rezultacie dla większych  $n$  liczba wywołań rośnie lawinowo, przy czym większość z nich powtarza się wielokrotnie. Rysunek 1.6 przedstawia drzewo ukazujące 15 wywołań przy obliczaniu liczby  $f_5$ .



**Rys. 1.6.** Drzewo wywołań rekurencyjnych  $FIB(5)$

Spróbujmy sprawdzić, jak bardzo niepraktyczny jest to algorytm. Niech  $T(n)$  oznacza liczbę wierzchołków (węzłów) w drzewie wywołań rekurencyjnych przy obliczaniu wartości  $f_n$ . Zauważmy, że dla  $n \geq 2$  drzewo o korzeniu (najwyższym wierzchołku)  $f_n$  składa się z dwóch poddrzew o korzeniach  $f_{n-1}$  i  $f_{n-2}$  oraz jego własnego korzenia  $f_n$ . Funkcję  $T(n)$  można więc opisać za pomocą zależności postaci:

$$T(n) = \begin{cases} 1 & (n = 0) \\ 1 & (n = 1) \\ T(n-1) + T(n-2) + 1 & (n \geq 2) \end{cases}$$

Rekurencja ta przypomina bardzo definicję ciągu Fibonacciego. Nietrudno zauważyć, że zachodzi nierówność:

$$T(n) \geq f_{n+1} \quad (n = 0, 1, \dots), \quad (1.13)$$

którą można łatwo sprawdzić za pomocą indukcji matematycznej. Wynika z niej, że złożoność czasowa algorytmu jest rzędu nie niższego niż obliczana wartość. Aby uwidocznić charakter tej złożoności, skorzystajmy ze wzoru ujawniającego zwartą postać liczb Fibonacciego [10, 21]:

$$f_n = \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1-\sqrt{5}}{2} \right)^n.$$

Ponieważ wartość bezwzględna drugiego składnika jest mniejsza od 1/2 dla każdej nieujemnej liczby całkowitej  $n$ , wartość  $f_n$  jest liczbą całkowitą najbliższą liczby:

$$\frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n \approx \frac{1}{\sqrt{5}} 1,61803^n.$$

Stąd i nierówności (1.13) wynika, że złożoność czasowa algorytmu *FIB* jest co najmniej wykładnicza:

$$T(n) \geq O(1,61803^n),$$

co w praktyce jest czynnikiem dyskwalifikującym (zob. ćw. 1.15).

Liczby Fibonacciego można również obliczać iteracyjnie, posługując się algorytmem nie tak eleganckim, ale o liniowej złożoności czasowej:

```
function FIB2(n)
  if n ≤ 1 then
    return n
  fp1 := 0
  fp2 := 1
  for k:=2 to n do
    tmp := fp2
    fp2 := fp2 + fp1
    fp1 := tmp
  return fp2
```

Użycie trzech zmiennych pomocniczych pozwoliło na uniknięcie wielokrotnego wyliczania tych samych wartości. Zmienne *fp1* i *fp2* służą do przechowywania wartości  $f_{k-2}$  i  $f_{k-1}$  podczas obliczania wartości  $f_k$ .

Należy podkreślić, że rekurencji nie powinno się unikać za wszelką cenę, gdyż wiele algorytmów rekurencyjnych jest równie efektywnych, jak ich odpowiedniki iteracyjne. Istnieją problemy, dla których rekurencja jest w pełni uzasadniona, zaś iteracja jedynie komplikuje zrozumienie sposobu ich rozwiązania. Niektóre z nich zostaną omówione w rozdziale 5. niniejszej części książki.

## 1.9. Metody tworzenia algorytmów

Niemal każdy problem daje się zakwalifikować do jakiejś klasy problemów, dla której są znane algorytmy rozwiązania. Jeżeli nie ma gotowego rozwiązania dla postawionego problemu, przydatna może okazać się znajomość technik tworzenia efektywnych algorytmów. Do najbardziej popularnych należą:

- ♦ metoda „dziel i zwyciężaj”,
- ♦ programowanie dynamiczne,
- ♦ metoda zachłanna,
- ♦ metoda prób i błędów.

Istotą metody „dziel i zwyciężaj” jest podział problemu na dwa lub większą liczbę podproblemów mniejszych rozmiarów, które dają się rozwiązać mniejszym kosztem, rozwiązanie każdego z nich, a następnie złożenie rozwiązania zasadniczego problemu z uzyskanych rozwiązań cząstkowych. Jeżeli podproblemy są zbyt duże, by dały się rozwiązać w prosty sposób, dzieli się je na jeszcze mniejsze podproblemy. Proces podziału kontynuowany jest aż do momentu, gdy podproblemy staną się na tyle małe, że ich rozwiązanie nie będzie stanowić trudności. Naturalną postacią wstępnego zapisu algorytmu typu „dziel i zwyciężaj” jest podprogram rekurencyjny. Później, gdy zajdzie potrzeba, można opracować bardziej wydajną wersję iteracyjną algorytmu.

Przykładem zastosowania metody „dziel i zwyciężaj” jest algorytm przeszukiwania binarnego ciągu uporządkowanego. W wyniku porównania poszukiwanego elementu  $x$  ze środkowym elementem tablicy  $a[i..j]$  dalsze przeszukiwanie zostaje ograniczone do jej lewej lub prawej połowy:

```
function BINARY-SEARCH2(a[1..n], x, i, j)
  if i > j then
    return -1
  k := (i+j) div 2
  if x = a[k] then
    return k
  else if x < a[k] then
    return BINARY-SEARCH2(a, x, i, k-1)
  else
    return BINARY-SEARCH2(a, x, k+1, j)
```



Sprawdzenie, czy element  $x$  znajduje się w tablicy  $a[1..n]$ , polega na skorzystaniu z wywołania

```
BINARY-SEARCH2(a, x, 1, n)
```

Algorytm *BINARY-SEARCH2* jest rekurencyjnym odpowiednikiem omówionego w podrozdziale 1.6 algorytmu iteracyjnego *BINARY-SEARCH*. Aby bardziej upodobnić jego nagłówek do wersji iteracyjnej, można pozbyć się dwóch ostatnich parametrów, ukrywając rekurencję poprzez wyodrębnienie wewnętrznej funkcji przeszukującej:

```
function BINARY-SEARCH3(a[1..n], x)
  function SEARCH(i, j)
    if i > j then
      return -1
    k := (i+j) div 2
    if x = a[k] then
      return k
    else if x < a[k] then
      return SEARCH(i, k-1)
    else
      return SEARCH(k+1, j)
  return SEARCH(1, n)
```

Całą pracę w algorytmie *BINARY-SEARCH3* wykonuje funkcja *SEARCH*, a rola samego algorytmu sprowadza się do pojedynczego wywołania tej funkcji z parametrami 1 i  $n$ , które powoduje przeszukanie całej tablicy.

Metoda „dziel i zwyciężaj” nie powinna być stosowana, gdy podział problemu prowadzi do wielokrotnego rozwiązywania tych samych podproblemów, jak np. w przypadku obliczania wyrazów ciągu Fibonacciego. Omówiona w poprzednim podrozdziale funkcja rekurencyjna *FIB* liczy wartość  $f_n$  dla  $n \geq 2$ , sumując wartości  $f_{n-1}$  i  $f_{n-2}$ , czyli dzieląc problem rozmiaru  $n$  na dwa podproblemy rozmiaru  $n-1$  i  $n-2$ . W rezultacie prowadzi to do wielokrotnego obliczania tych samych wartości i wykładniczej złożoności algorytmu.

**Programowanie dynamiczne** opiera się również na podziale problemu na mniejsze podproblemy. Jednak tym razem najpierw rozwiązuje się podproblemy, poczynając od najmniejszych, a ich wyniki przechowuje w tablicy pomocniczej, nawet gdy nie wszystkie okażą się potrzebne. Później, gdy zajdzie potrzeba ich wykorzystania, algorytm może się do nich bezpośrednio odwoływać, zamiast obliczać je na nowo. W ten sposób programowanie dynamiczne poprawia metodę „dziel i zwyciężaj” w sytuacji, gdy wymagane jest wielokrotne rozwiązywanie tych samych podproblemów. Technika programowania dynamicznego zazwyczaj redukuje złożoność czasową algorytmu z wykładniczej do wielomianowej.

W celu przekształcenia algorytmu *FIB* do postaci zgodnej z filozofią programowania dynamicznego użyjemy tablicy pomocniczej  $f[0..n]$ , w której będziemy zapisywać obliczane kolejne wartości  $f_0, f_1, \dots, f_n$ :

```
function FIB3(n)
  f[0] := 0
  f[1] := 1
  for k:=2 to n do
    f[k] := f[k-1] + f[k-2]
  return f[n]
```

Otrzymany algorytm ma złożoność czasową  $O(n)$ . Zauważmy, że zaprezentowany w poprzednim podrozdziale algorytm *FIB2* stanowi modyfikację algorytmu *FIB3* polegającą na zastąpieniu tablicy  $f[0..n]$  trzema pomocniczymi zmiennymi.

**Metoda zachłanna** polega na konsekwentnym wyborze najkorzystniejszego w danym momencie rozwiązania w nadziei otrzymania rozwiązania optymalnego globalnie. Mówiąc inaczej, algorytm zachłanny rozwiązuje dany problem w wyniku podejmowania w każdym kroku decyzji, z których każda wydaje się w danej chwili najlepsza. W wielu przypadkach podejście zachłanne prowadzi do znalezienia rozwiązania optymalnego globalnie, ale często jest nieskuteczne, dając jedynie rozwiązanie przybliżone.

Przykładem skutecznego zastosowania metody zachłannej jest zaprezentowany w podrozdziale D.7 problem konwersji liczby z notacji rzymskiej na dziesiętną i dziesiętnej na rzymską. Zachłanność algorytmu przejawia się w wyborze symboli rzymskich w kolejności od M do I, czyli od największej do najmniejszej wagi.

Innym przykładem jest często spotykany problem wydawania reszty w sklepie lub wypłacania kwoty w banku. Zazwyczaj klienci nie lubią otrzymywać zbyt wielu banknotów i monet, dlatego kasjer powinien wyrażać kwotę przy użyciu jak najmniejszej liczby nominałów. Algorytm opisujący metodę postępowania może wyglądać następująco:

```
procedure RESZTA(Kwota, Ile[1..14])
  Nom[1..14] = (20000, 10000, 5000, 2000, 1000, 500, 200, 100,
               50, 20, 10, 5, 2, 1)
  for k:=1 to 14 do
    Ile[k] := Kwota div Nom[k]
    Kwota := Kwota mod Nom[k]
```

Tablica *Nom* zawiera wszystkie nominały wyrażone w groszach, w kolejności od 200 zł do 1 gr. Algorytm wylicza dla każdego z nich, ile razy mieści się w kwocie wyrażonej w groszach, zapamiętuje wynik w tablicy *Ile* i pomniejsza tę kwotę o wartość wyliczonej liczby nominału. Zachłanność algorytmu polega na pobieraniu kolejnego najwyższego nieuwzględnionego nominału.

Optymalność rozwiązania generowanego przez algorytm gwarantuje przyjęty system wartości nominałów. Gdyby jednak uwzględnić hipotetyczną monetę o wartości 12 groszy, algorytm nie zawsze dawałby rozwiązanie optymalne. Na przykład kwotę 15 groszy wyraziłby za pomocą trzech monet (12 gr + 2 gr + 1 gr), podczas gdy właściwe rozwiązanie zawierałoby dwie monety (10 gr + 5 gr).

Istnieją problemy, dla których najwygodniej jest szukać rozwiązania nie za pomocą wyraźnie określonej reguły obliczeniowej, lecz **metodą prób i błędów**. Cechą charakterystyczną takich algorytmów jest to, że kolejne kroki, które mogą doprowadzić do rozwiązania, są zapisywane, a gdy później okaże się, że jakiś krok jest niewłaściwy, następuje usunięcie jego zapisu i wycofanie się do stanu poprzedzającego błędną decyzję. Algorytmy takie są nazywane **algorytmami z powrotami** lub **z nawrotami**. Najbardziej naturalnym ich opisem jest rekurencja.

Przykładem zastosowania algorytmu z powrotami jest problem plecakowy. Jedną z jego odmian, omówioną w rozdziale 5. niniejszej książki, polega na wyniesieniu przez Alibabę skarbu znajdującego w Sezamie. Alibaba może unieść tylko pewien ciężar, więc musi zdecydować, które przedmioty wybrać, aby w drodze powrotnej wynieść część skarbu o jak najwyższej wartości. Gdyby Alibaba kierował się strategią działania zachłannego, mógłby nie dokonać najlepszego wyboru. Na przykład, gdyby był w stanie unieść ciężar 50 kg, to w przypadku skarbu złożonego z trzech przedmiotów o wadze 25 kg, 42 kg, 20 kg i wartości 400 zł, 680 zł, 560 zł wybrałby jeden przedmiot o ciężarze 42 kg i wartości 680 zł zamiast dwóch przedmiotów o łącznym ciężarze 45 kg i wartości 960 zł. Właściwe rozwiązanie polega na systematycznym podejmowaniu kolejnych prób dobierania przedmiotów aż do wyczerpania wszystkich dopuszczalnych kombinacji.

## Ćwiczenia

1.1. Dla danych wartości rzeczywistych  $p$ ,  $q$  i  $r$  funkcja:

$$f(x) = x^3 + p \cdot x^2 + q \cdot x + r$$

jest ciągła i spełnia warunki:

$$\lim_{x \rightarrow -\infty} f(x) = -\infty, \quad \lim_{x \rightarrow +\infty} f(x) = +\infty,$$

ma więc co najmniej jedno miejsce zerowe. Opracuj algorytm wyznaczania go z wymaganą dokładnością. Algorytm ma najpierw znajdować taki przedział  $[a, b]$ , w którego końcach funkcja przyjmuje przeciwne znaki:

$$f(a)f(b) < 0,$$

a potem wyznaczyć istniejące w nim rozwiązanie **metodą bisekcji**, zwanej też **metodą połowienia**. Polega ona na podziale przedziału na dwa równe poprzedziały i wyborze tego z nich, w którego końcach funkcja zachowuje przeciwne znaki. Postępowanie takie jest powtarzane aż do chwili, gdy uzyskany podprzedział ma wystarczająco małą długość. Zapisz ten algorytm w postaci schematu blokowego i implementacji komputerowej.

**1.2.** Problem szukania największego wspólnego dzielnika dwóch dodatnich liczb całkowitych można rozwiązać bez użycia operacji dzielenia, wykonując jedynie odejmowania. Opis słowny tej wersji algorytmu Euklidesa jest następujący:

*Od wartości większej odejmuj mniejszą dopóty, dopóki są one różne. Gdy się zrównają, będą równe największemu wspólnemu dzielnikowi obydwu wartości początkowych.*

Zapisz ten algorytm w postaci listy kroków, schematu blokowego i w pseudokodzie programowania oraz udowodnij jego poprawność.

**1.3.** Zaproponuj algorytmy obliczania sumy i iloczynu oraz wyszukiwania elementu maksymalnego i minimalnego ciągu liczbowego. Zapisz je w pseudokodzie programowania, udowodnij ich poprawność i oszacuj złożoność obliczeniową.

**1.4.** Dowolną liczbę rzeczywistą  $x \neq 0$  można zapisać w tzw. **znormalizowanej postaci zmiennopozycyjnej**:

$$x = s \cdot 2^c \cdot m,$$

gdzie  $s = +1$  lub  $-1$  jest jej **znakiem**,  $c$  – liczbą całkowitą zwaną **cechą**, a  $m$  – liczbą rzeczywistą z przedziału  $[1/2, 1)$  zwaną **mantysą**. Dla  $x = 0$  przyjmujemy  $s = c = m = 0$ . Opracuj algorytm wyznaczania trójki liczb  $s$ ,  $c$ ,  $m$  dla danej liczby rzeczywistej  $x$ . Zapisz ten algorytm w postaci schematu blokowego i w pseudokodzie programowania oraz udowodnij jego poprawność.

**1.5.** W udoskonalonym algorytmie sortowania bąbelkowego **BUBBLE-SORT2** zapamiętuje się, czy podczas przebiegu nastąpiły jakieś przestawienia elementów ciągu, a gdy nie miały one miejsca, sortowanie zostaje zakończone. Inne ulepszenie polega na zapamiętaniu indeksu ostatniego przestawienia. Elementy o indeksach wyższych od tego indeksu są ustawione w prawidłowej kolejności, można więc nie rozpatrywać ich w następnych przebiegach. Można też kolejne przebiegi wykonywać w naprzemiennych kierunkach, a wtedy dwa indeksy ostatnich przestawień mogą się przesuwac ku środkowi ciągu, powodując zmniejszenie liczby porównań.

Zapisz tak zmodyfikowane algorytmy w pseudojęzyku programowania i w postaci implementacji komputerowej.

**1.6.** Opracuj w Delphi uniwersalną implementację algorytmu sortowania bąbelkowego, która umożliwia nie tylko sortowanie ciągów liczb całkowitych, lecz także rzeczywistych i łańcuchów. Porównaj jej efektywność czasową w przypadku sortowania liczb całkowitych z efektywnością procedury *BubbleSort*.

**1.7.** Zapisz algorytmy przeszukiwania sekwencyjnego i binarnego w pseudojęzyku programowania bez wykorzystania instrukcji **return**. Opracuj implementacje komputerowe tych algorytmów i program, który mierzy i porównuje ich czasy wykonania.

**1.8.** Korzystając z definicji notacji asymptotycznej  $O$ , udowodnij następujące twierdzenia:

$$f(n) = O(g(n)) \wedge c > 0 \Rightarrow c \cdot f(n) = O(g(n))$$

$$f(n) = O(a(n)) \wedge g(n) = O(b(n)) \Rightarrow f(n) + g(n) = O(\max(a(n), b(n)))$$

$$f(n) = O(a(n)) \wedge g(n) = O(b(n)) \Rightarrow f(n) \cdot g(n) = O(a(n) \cdot b(n))$$

$$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

**1.9.** Uporządkuj następujące funkcje według ich rzędów wielkości i wyraż je w notacji asymptotycznej  $O$ :

$$\begin{array}{lll} n^3 \log n + 5 & n\sqrt{n^4 - 3n + 2} & n^{5/2} / \log n \\ n^{7/2}(\log n + 1) & n^2 \sqrt{n^3 + \log n + 2} & n^3 / \sqrt{n - \log n} \end{array}$$

**1.10.** Znajdź rząd wielkości poniższych funkcji i wyraż go w notacji asymptotycznej  $O$ :

$$f(n) = 1 + 3 + 5 + \dots + (2n - 1)$$

$$f(n) = 1 + 2 + 4 + \dots + 2^{n-1}$$

$$f(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

$$f(n) = \log 1 + \log 2 + \log 3 + \dots + \log n$$

$$f(n) = a_0 n^k + a_1 n^{k-1} + \dots + a_{k-1} n + a_k, \text{ gdzie } a_0 > 0$$

**1.11.** Niech  $A$  i  $B$  będą macierzami odpowiednio o  $m \times k$  i  $k \times m$  elementach. Zapisz w pseudojęzyku programowania algorytm obliczania iloczynu tych macierzy metodą tradycyjną i wyraż jego złożoność czasową w notacji asymptotycznej  $O$ .

**1.12.** Zapisz w pseudojęzyku algorytm obliczania wartości wielomianu według wzoru:

$$a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n$$

i algorytm Hornera

$$(\dots((a_0 x + a_1)x + a_2)x + \dots + a_{n-1})x + a_n$$

Znajdź złożoności obliczeniowe obu algorytmów i porównaj czasy działania ich implementacji komputerowych.

**1.13.** Korzystając z zależności rekurencyjnej (1.11) i indukcji matematycznej udowodnij, że wzór (1.12) określa złożoność czasową algorytmu Euklidesa.

**1.14.** Opracuj implementacje komputerowe wersji *EUCLID3* i *EUCLID4* algorytmu Euklidesa oraz program, który mierzy i porównuje czasy ich wykonania.

**1.15.** Opracuj implementacje komputerowe wersji *FIB*, *FIB2* i *FIB3* algorytmu wyznaczania liczb Fibonacciego oraz program, który mierzy i porównuje czasy ich wykonania.

**1.16.** Napisz podprogram rekurencyjny wypisujący słownie cyfry dziesiętne nieujemnej liczby całkowitej. Na przykład dla liczby 7034 prawidłowym wynikiem jest tekst

siedem zero trzy cztery

**1.17.** Opracuj w pseudojęzyku programowania rekurencyjne algorytmy obliczania sumy i iloczynu oraz wyszukiwania elementu maksymalnego i minimalnego ciągu liczbowego. Porównaj złożoności obliczeniowe tych algorytmów ze złożonościami ich odpowiedników iteracyjnych (zob. ćw. 1.3).

**1.18. Wielomiany Czebyszewa** można zdefiniować następująco:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \quad (k = 2, 3, \dots)$$

Opracuj dwa algorytmy obliczania wartości wielomianu Czebyszewa stopnia  $n$ , korzystając z powyższej definicji rekurencyjnej i programowania dynamicznego. Znajdź złożoności obliczeniowe obu algorytmów.

**1.19. Współczynniki Newtona**, zwane również **dwumiennymi**, są określone dla dowolnej nieujemnej liczby całkowitej  $n$  i  $k = 0, 1, \dots, n$  następująco:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Wprost z definicji wynika, że mają one własność symetrii:

$$\binom{n}{k} = \binom{n}{n-k}$$

Ponadto spełniają znany związek rekurencyjny:

$$\binom{n}{k} = \begin{cases} 1 & (k = 0, n) \\ \binom{n-1}{k-1} + \binom{n-1}{k} & (k = 1, 2, \dots, n-1) \end{cases}$$

na podstawie którego można je ustawić w tzw. **trójkąt Pascala** [10, 11]. Każdy wiersz trójkąta powstaje w ten sposób, że współczynniki skrajne są jedynkami, zaś wewnętrzne sumą dwóch sąsiednich współczynników wiersza poprzedniego – tego na lewo skos i tego na prawo skos:

$$\begin{array}{ccccccc} n=0 & & & & & & 1 \\ n=1 & & & & 1 & & 1 \\ n=2 & & & 1 & & 2 & & 1 \\ n=3 & & 1 & & 3 & & 3 & & 1 \\ n=4 & 1 & & 4 & & 6 & & 4 & & 1 \\ & \dots & & \dots & & \dots & & \dots & & \dots \end{array}$$

Opracuj trzy algorytmy obliczania wartości współczynników Newtona, korzystając z definicji, metody „dziel i zwyciężaj” i programowania dynamicznego oraz znajdź ich złożoności czasowe. Opracuj implementacje komputerowe tych algorytmów i program, który mierzy i porównuje czasy ich wykonania.

**1.20. Liczby Stirlinga drugiego rodzaju**  $S(n, k)$ , określone dla dowolnej nieujemnej liczby całkowitej  $n$  i  $k = 0, 1, \dots, n$ , spełniają następujące związki rekurencyjne [18, 21]:

$$S(n, n) = 1 \quad (n \geq 0)$$

$$S(n, 0) = 0 \quad (n > 0)$$

$$S(n, k) = S(n-1, k-1) + kS(n-1, k) \quad (0 < k < n)$$

Podobnie jak w przypadku współczynników Newtona, wzory te umożliwiają łatwe wyznaczanie liczb  $S(n, k)$  i ustawianie ich w tablicę, którą można traktować jako **trójkąt Stirlinga**. Opracuj dwa algorytmy obliczania liczb Stirlinga, korzystając z powyższych wzorów i programowania dynamicznego, oraz program drukujący trójkąt Stirlinga.



## Rozdział 2. Sortowanie

Jednym z najważniejszych i najczęściej spotykanych problemów informatycznych jest **sortowanie**, które polega na ustawieniu elementów zbioru w określonym porządku w celu ułatwienia ich późniejszego wyszukiwania. Rozróżnia się sortowanie **wewnętrzne**, gdy wszystkie elementy sortowanego zbioru znajdują się w pamięci wewnętrznej komputera, i **zewnętrzne**, gdy znajdują się w pamięci masowej. W pierwszym przypadku elementy są dostępne bezpośrednio, w drugim jest wymagane przesyłanie ich pomiędzy pamięcią wewnętrzną a urządzeniami pamięci zewnętrznej (dyski, taśmy magnetyczne).

W praktyce elementami sortowanego zbioru są zazwyczaj rekordy złożone z szeregu pól danych, z których jedno lub kilka tworzy tzw. **klucz** (ang. *key*), według którego są ustawiane rekordy. Pozostałe pola nie mają wpływu na przebieg sortowania. Wartościami klucza są często liczby (całkowite lub rzeczywiste) i łańcuchy. Na przykład w kartotece studentów, w której rekordy zawierają takie pola, jak *imię*, *nazwisko*, *data* i *miejsce urodzenia*, *pleć*, *PESEL*, *adres zamieszkania*, *numer indeksu*, *nazwa wydziału* i *kierunku*, *rok studiów* itp., kluczem może być pole *numer indeksu*, a gdy interesuje nas alfabetyczne uporządkowanie kartoteki według nazwiska i imienia studenta, klucz tworzą dwa pola – *nazwisko* i *imię*.

W najprostszym przypadku, gdy elementami sortowanego zbioru są liczby lub łańcuchy, kluczem jest cały element, a w najbardziej ogólnym na elementach jest zdefiniowana **funkcja porządkująca**, zwana też **funkcją klucza**, o wartościach należących do pewnego zbioru, w którym jest określona relacja liniowego porządku, oznaczana zwyczajowo symbolem  $\leq$  lub  $\geq$ . Najczęściej nie oblicza się wartości funkcji klucza, lecz przechowuje je w jawnej postaci w polu rekordu.

Formalna specyfikacja **problemu sortowania** jest następująca:

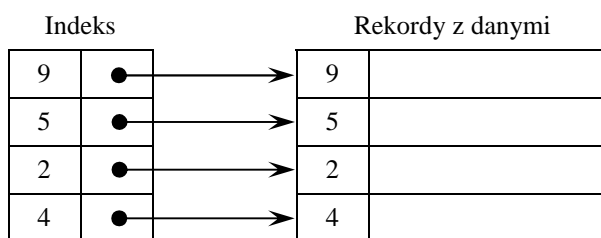
**Dane:** Ciąg  $n$  elementów  $a_1, a_2, \dots, a_n$  i określona na nich funkcja klucza  $key$ .

**Wynik:** Uporządkowanie elementów  $a_{i_1}, a_{i_2}, \dots, a_{i_n}$  takie, że

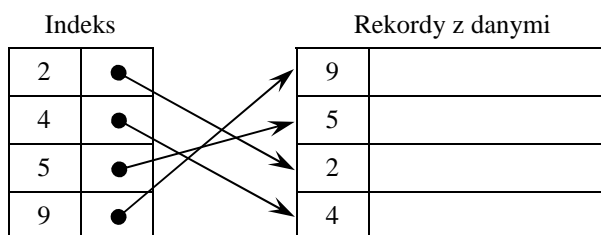
$$key(a_{i_1}) \leq key(a_{i_2}) \leq \dots \leq key(a_{i_n}).$$

W praktyce rekordy mogą zawierać dużą liczbę danych, dlatego ich fizyczne przemieszczanie w pamięci komputera może być operacją czasochłonną. Aby uniknąć przemieszczania dużych rekordów, tworzy się niekiedy dodatkowy zbiór,

zwany **indeksem**, którego rekordy są parami pól złożonymi z klucza i dowiązania (wskaźnika) do rekordu z danymi, a następnie sortuje się go (por. rys. 2.1). Korzystając z posortowanego indeksu, można łatwo przestawić właściwe rekordy tak, by uzyskać ich pożądane uporządkowanie, ale najczęściej się tego nie robi, ponieważ dowiązania wystarczają do określenia położenia rekordów.



a) przed sortowaniem



b) po sortowaniu

**Rys. 2.1.** Sortowanie z wykorzystaniem indeksu

W praktycznych zastosowaniach klucz jest jedyną istotną składową służącą do identyfikacji rekordów i sterującą przebiegiem procesu sortowania. Pozostałe dane są z nim jedynie przemieszczane jako część każdego rekordu. To, czy sortujemy duże rekordy, czy tylko ich indeksy, czy nawet pojedyncze liczby, a także to, czy klucze składają się z wielu pól, czy są pojedynczymi polami, jest nieistotne pod względem wyboru algorytmu i wyjaśnienia sposobu jego działania. Dlatego bez szkody dla ogólności rozważań możemy przyjąć, że elementy sortowanego zbioru są pewnymi rekordami o kluczu przechowywanym w polu *key*. Oczywiście, przetłumaczenie algorytmu sortowania hipotetycznych rekordów z pseudojęzyka na konkretny język programowania wymaga uwzględnienia rzeczywistej struktury rekordów i innych szczegółów implementacyjnych.

Zajmiemy się najpierw podstawowymi algorytmami sortowania wewnętrznego. Najbardziej naturalną reprezentacją ciągu elementów jest tablica. Zakładamy więc, że dane do sortowania znajdują się w tablicy, a z uwagi na oszczędną gospodarkę

pamięcią, że ich porządkowanie jest wykonywane *in situ* (w miejscu), czyli bez wykorzystania tablicy pomocniczej.

Złożoność czasową algorytmu sortowania wewnętrznego będziemy określać, przyjmując za operację dominującą porównanie dwóch elementów ciągu, podobnie jak w przypadku sortowania bąbelkowego omówionego w rozdziale 1. Taka miara jest reprezentatywna, gdy operacja porównania jest czasochłonna, np. gdy klucze są długimi łańcuchami. W rzeczywistości przemieszczanie rekordów często zajmuje o wiele więcej czasu niż porównanie kluczy i w bardziej szczegółowej analizie powinno być uwzględniane. Złożonością pamięciową rozpatrywanych algorytmów jest liczba komórek pamięci zajmowanych przez sortowany ciąg powiększona o jedną komórkę potrzebną do przestawienia jego dwóch elementów.

## 2.1. Sortowanie przez selekcję

**Sortowanie przez selekcję** (ang. *selection sort*) polega na wybraniu najmniejszego elementu w ciągu  $a_1, a_2, \dots, a_n$  i zamienieniu go miejscami z pierwszym elementem –  $a_1$ , wybraniu najmniejszego elementu w  $a_2, a_3, \dots, a_n$  i zamienieniu go z drugim –  $a_2$  itd., aż w końcu pozostanie jeden największy element –  $a_n$ . Tabela 2.1 obrazuje działanie algorytmu na przykładzie ciągu liczbowego złożonego z 7 elementów. Numery  $i = 1, 2, \dots, 6$  określają docelowe miejsca kolejnych najmniejszych elementów, a strzałki pokazują ich przesunięcia.

**Tabela 2.1.** Przykład sortowania przez selekcję ciągu 7-elementowego

Pocz.	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
15	4	4	4	4	4	4
28	28	6	6	6	6	6
13	13	13	13	13	13	13
6	6	28	28	15	15	15
35	35	35	35	35	28	28
4	15	15	15	28	35	32
32	32	32	32	32	32	35

Wstępne sformułowanie algorytmu może wyglądać następująco:

```
for i:=1 to n-1 do
  k := indeks najmniejszego elementu spośród a[i]...a[n]
  zamień a[i] z a[k]
```

Uściślenie tego zapisu wymaga dodania nagłówka procedury i sprecyzowania operacji zapisanych w języku naturalnym. Wyznaczenie indeksu  $k$  najmniejszego elementu spośród elementów  $a_i, a_{i+1}, \dots, a_n$  odbywa się według prostego schematu przeszukiwania sekwencyjnego tablicy z wykorzystaniem pętli **for**. Pełna wersja algorytmu sortowania przez selekcję może w pseudojęzyku programowania mieć postać następującego podprogramu:

```

procedure SELECTION-SORT( $a[1..n]$ )
  for  $i:=1$  to  $n-1$  do
     $k := i$ 
    for  $j:=i+1$  to  $n$  do
      if  $a[j].key < a[k].key$  then  $k := j$ 
     $tmp := a[i]$ 
     $a[i] := a[k]$ 
     $a[k] := tmp$ 

```

W implementacji algorytmu *SELECTION-SORT* w języku Object Pascal należy określić typ tablicy reprezentującej sortowany ciąg. Dla ustalenia uwagi założymy, podobnie jak w przypadku algorytmu *BUBBLE-SORT*, że jest ona typu *integer*, zadeklarujemy ją też w parametrze podprogramu jako tablicę otwartą. Ponadto wprowadzimy małe udoskonalenie algorytmu polegające na pomijaniu przestawień elementów, które zajmują właściwą pozycję. W wyniku otrzymujemy następującą procedurę sortowania przez selekcję:

```

procedure SelectionSort(var  $a$ : array of integer);
var
   $i, j, k, tmp$ : integer;
begin
  for  $i:=0$  to  $High(a)-1$  do
    begin
       $k := i$ ;
      for  $j:=i+1$  to  $High(a)$  do
        if  $a[j] < a[k]$  then  $k := j$ ;
      if  $k \neq i$  then
        begin
           $tmp := a[i]$ ;
           $a[i] := a[k]$ ;
           $a[k] := tmp$ ;
        end;
      end;
    end;
end;

```

Przypomnijmy, że konsekwencją zadeklarowania tablicy  $a$  jako otwartej jest numeracja jej elementów od 0 do  $n-1$  zamiast od 1 do  $n$ . Dlatego inny jest zakres wartości zmiennej sterującej w zewnętrznej pętli **for** w podprogramie *SelectionSort* niż w jego pierwowzorze *SELECTION-SORT*.

W podprogramie występują dwie zagnieżdżone instrukcje **for**, a porównanie elementów ciągu znajduje się wewnątrz zakresu pętli wewnętrznej. Złożoność czasowa algorytmu jest więc równa liczbie wszystkich wykonań pętli wewnętrznej:

$$T(n) = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2).$$

Sortowanie przez selekcję ma, podobnie jak sortowanie bąbelkowe, kwadratowy rodzaj złożoności czasowej. Dlatego też w zastosowaniach, w których  $n$  jest duże, powinno się korzystać z algorytmów bardziej wydajnych.

## 2.2. Sortowanie przez wstawianie

**Sortowanie przez wstawianie** (ang. *insertion sort*) jest powszechnie stosowane przez grających w karty. Zazwyczaj podczas rozdawania kart gracz pobiera ze stołu kolejne karty jedną ręką i wstawia je w odpowiednie miejsce kompletowanej talii kart trzymanej w drugiej ręce. Ogólnie, ciąg elementów jest podzielony na dwie części – uporządkowaną  $a_1, a_2, \dots, a_{i-1}$  i nieuporządkowaną  $a_i, a_{i+1}, \dots, a_n$ . Dla każdego kroku o numerze  $i = 2, 3, \dots, n$  element  $a_i$  jest przenoszony z części nieuporządkowanej ciągu do jego części uporządkowanej, bez zaburzania jej uporządkowania. Tabela 2.2 ilustruje działanie algorytmu na przykładzie ciągu liczb całkowitych składającego się z 7 elementów.

**Tabela 2.2.** Przykład sortowania przez wstawianie ciągu 7-elementowego

Pocz.	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
25	13	13	6	6	6	6
13	25	18	13	13	13	13
18	18	25	18	18	18	15
6	6	6	25	25	24	18
35	35	35	35	35	25	24
24	24	24	24	24	35	25
15	15	15	15	15	15	35

Algorytm możemy wstępnie sformułować następująco:

```

for  $i := 2$  to  $n$  do
    tmp :=  $a[i]$ 
    wstaw tmp w odpowiednie miejsce części  $a[1] \dots a[i-1]$ 

```

Jednym z narzucających się sposobów znalezienia odpowiedniego miejsca, w które należy wstawić element *tmp*, jest porównywanie go z kolejnymi elementami  $a_j$  ( $j = i-1, i-2, \dots$ ) i przesuwanie ich o jedną pozycję w kierunku prawego końca ciągu (w tab. 2.2 ku dołowi) dopóty, dopóki jest od nich mniejszy<sup>1</sup>:

```
j := i-1
while tmp.key < a[j].key do
  a[j+1] := a[j]
  j := j-1
a[j+1] := tmp
```

Niestety, program nie zadziała prawidłowo, gdy element *tmp* ma klucz mniejszy od kluczy wszystkich elementów uporządkowanej części ciągu. Błąd możemy łatwo naprawić, przerywając pętlę, gdy osiągnięty został lewy koniec (początek) ciągu. Pełny algorytm możemy zapisać w pseudojęzyku w postaci podprogramu:

```
procedure INSERTION-SORT(a[1..n])
  for i:=2 to n do
    tmp := a[i]
    j := i-1
    while (j > 0) and (tmp.key < a[j].key) do
      a[j+1] := a[j]
      j := j-1
    a[j+1] := tmp
```

Implementacja algorytmu w języku Object Pascal w przypadku ciągu liczb całkowitych reprezentowanego przez tablicę otwartą ma postać:

```
procedure InsertionSort(var a: array of integer);
var
  i, j, tmp: integer;
begin
  for i:=1 to High(a) do
    begin
      tmp := a[i];
      j := i-1;
      while (j >= 0) and (tmp < a[j]) do
        begin
          a[j+1] := a[j];
          j := j-1;
        end;
      a[j+1] := tmp;
    end;
  end;
```

---

<sup>1</sup> Takie przemieszczanie elementu ciągu, polegające na naprzemiennym porównywaniu go z pozostałymi elementami i przesuwaniu ich, nazywane jest przesiewaniem [17, 28].

Analiza złożoności czasowej algorytmu przez wstawianie sprowadza się, podobnie jak w przypadku algorytmu przez selekcję, do wyliczenia łącznej liczby wykonań pętli wewnętrznej:

$$T(n) = 1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2).$$

Algorytm ma więc kwadratową złożoność czasową, z tego względu jest użyteczny jedynie dla niewielkich wartości  $n$ .

Powróćmy jeszcze do kodu procedury *INSERTION-SORT*. Warunek kontynuacji pętli **while** możemy uprościć, tworząc dodatkowy element  $a_0$  o kluczu  $-\infty$  (minus nieskończoność), zwany **wartownikiem**<sup>2</sup>. Sprawdzanie, czy osiągnięty został lewy koniec ciągu, stanie się wówczas zbędne, ponieważ każda wartość  $tmp$  będzie większa od wartości wartownika. Algorytm ma wówczas postać:

```

procedure INSERTION-SORT2( $a[0..n]$ )
   $a[0].key := -\infty$ 
  for  $i := 2$  to  $n$  do
     $tmp := a[i]$ 
     $j := i-1$ 
    while  $tmp.key < a[j].key$  do
       $a[j+1] := a[j]$ 
       $j := j-1$ 
     $a[j+1] := tmp$ 

```

Zauważmy, że zastosowanie metody z wartownikiem wymagało rozszerzenia zakresu indeksu tablicy  $a$  do  $0..n$ . Użycie wartownika daje znikomy zysk czasowy i nie poprawia asymptotycznej złożoności czasowej algorytmu, a jedyną korzyścią jest poprawienie jego czytelności.

## 2.3. Sortowanie przez wstawianie binarne

W algorytmie sortowania przez wstawianie ciąg wynikowy  $a_1, a_2, \dots, a_{i-1}$ , do którego wstawiamy nowy element  $tmp$ , jest uporządkowany. W związku z tym dla ustalenia miejsca wstawienia możemy zastosować przeszukiwanie **binarne**, które zredukuje liczbę porównań elementów ciągu. W wyniku porównania klucza nowego elementu z kluczem środkowego elementu uporządkowanej części ciągu wybie-

<sup>2</sup> W praktyce w roli  $-\infty$  może wystąpić dowolna wartość, która jest mniejsza od wartości każdego klucza. W Delphi najwygodniej jest przyjąć najmniejszą wartość typu klucza. Dla liczb całkowitych można ją określić szesnastkowo lub za pomocą funkcji *Low*, np. dla typu *integer* jest nią *Low(integer)*. Dla liczb rzeczywistych można ją zdefiniować jako  $-1/0$ .

ramy tę jej połowę, w której znajduje się miejsce wstawienia nowego elementu, dzielimy ją znowu na połowę i wybieramy właściwą itd., aż w końcu uzyskany przedział przeszukiwania ma długość 1. Tak ulepszony algorytm ma postać:

```

procedure BIN-INSERTION-SORT(a[1..n])
  for i:=2 to n do
    tmp := a[i]
    L := 1
    P := i-1
    while L ≤ P do
      k := (L+P) div 2
      if tmp.key < a[k].key then P := k-1
      else L := k+1
    for k:=i-1 downto L do
      a[k+1] := a[k]
    a[L] := tmp

```

Zmienne  $L$  i  $P$  określają indeksy lewego i prawego końca (pierwszego i ostatniego elementu) rozpatrywanej części ciągu. Jej długość zmniejsza się w każdym wykonaniu pętli **while** o połowę. Po zakończeniu pętli znalezione miejsce wstawienia nowego elementu wskazuje zmienna  $L$ , dlatego wszystkie elementy uporządkowanej części ciągu są od tego miejsca przesuwane w prawo o jedną pozycję, po czym nowy element zostaje wstawiony na pozycję  $L$ .

Nietrudno przetłumaczyć powyższą wersję algorytmu na język Object Pascal. W przypadku sortowania ciągu liczb całkowitych reprezentowanego przez tablicę otwartą implementacja ta ma postać:

```

procedure BinInsertionSort(var a: array of integer);
var
  i, k, L, P, tmp: integer;
begin
  for i:=1 to High(a) do
    begin
      tmp := a[i];
      L := 0;
      P := i-1;
      while L <= P do
        begin
          k := (L+P) div 2;
          if tmp < a[k] then P := k-1
          else L := k+1;
        end;
      for k:=i-1 downto L do
        a[k+1] := a[k];
      a[L] := tmp;
    end;
  end;

```



Przeprowadźmy teraz analizę złożoności czasowej algorytmu sortowania przez wstawianie binarne. Dla każdego  $i = 2, 3, \dots, n$  wykonywane jest przeszukiwanie binarne ciągu  $a_1, a_2, \dots, a_{i-1}$  liczącego  $i - 1$  elementów. Na podstawie wzoru (1.9) ogólna liczba porównań wynosi:

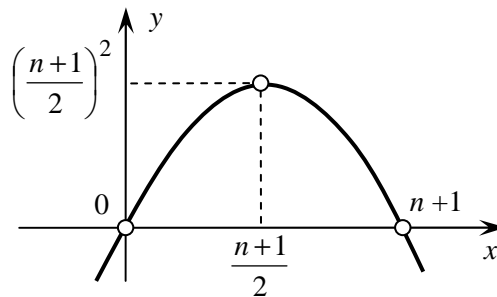
$$\begin{aligned} T(n) &= \lfloor \log 1 \rfloor + 1 + \lfloor \log 2 \rfloor + 1 + \dots + \lfloor \log(n-1) \rfloor + 1 \leq \\ &\leq \log 1 + \log 2 + \dots + \log(n-1) + (n-1) = \\ &= \log(n-1)! + (n-1) \end{aligned} \quad (2.1)$$

Dalszą analizę moglibyśmy przeprowadzić w oparciu o nierówność  $n! \leq n^n$ , która jest słabym ograniczeniem górnym silni wynikającym z oczywistego faktu, iż każdy z  $n$  czynników występujących w iloczynie określającym silnię jest nie większy niż  $n$ . Moglibyśmy też wykorzystać słynny wzór Stirlinga:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n,$$

który daje asymptotyczne przybliżenie silni, zwłaszcza dobre dla dużych  $n$  [7, 10]. Skorzystamy jednak z nierówności:

$$n! \leq \left(\frac{n+1}{2}\right)^n. \quad (2.2)$$



**Rys. 2.2.** Wykres wielomianu  $P(x) = x(n+1-x)$

W celu wykazania prawdziwości nierówności (2.2) zauważmy, że:

$$(n!)^2 = (1 \cdot 2 \cdot \dots \cdot n) \cdot (n \cdot \dots \cdot 2 \cdot 1) = (1 \cdot n) \cdot \dots \cdot (n \cdot 1) = \prod_{k=1}^n k(n+1-k) = \prod_{k=1}^n P(k),$$

gdzie

$$P(x) = x(n+1-x) = -x^2 + (n+1)x$$

jest wielomianem drugiego stopnia. Łatwo sprawdzić (por. rys. 2.2), że w punkcie  $x = (n+1)/2$  przyjmuje on wartość maksymalną:

$$\max_x P(x) = \left(\frac{n+1}{2}\right)^2.$$

Mamy zatem:

$$(n!)^2 \leq \prod_{k=1}^n \left(\frac{n+1}{2}\right)^2 = \left(\frac{n+1}{2}\right)^{2n},$$

skąd po spierwiastkowaniu dostajemy nierówność (2.2).

Możemy wreszcie powrócić do analizy złożoności czasowej sortowania przez wstawianie binarne. Na podstawie nierówności (2.1) i (2.2) otrzymujemy:

$$\begin{aligned} T(n) &\leq \log\left(\frac{n}{2}\right)^{n-1} + (n-1) = (n-1)\log\left(\frac{n}{2}\right) + (n-1) = \\ &= (n-1)(\log n - \log 2) + (n-1) = (n-1)(\log n - 1) + (n-1) = \\ &= (n-1)\log n = n \log n - \log n = O(n \log n) \end{aligned}$$

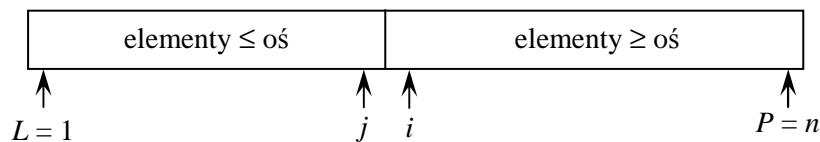
Tak więc, dzięki zastosowaniu przeszukiwania binarnego algorytm działa w czasie liniowo-logarytmicznym. Niestety, liczba porównań zmniejszyła się, ale nie liczba przesunięć elementów ciągu, które w rzeczywistości zajmują często więcej czasu niż porównania. Co więcej, ulepszony algorytm może w niektórych przypadkach działać dłużej, niż jego poprzednik, czego oczywistym przykładem jest sortowanie ciągu uporządkowanego.

## 2.4. Sortowanie szybkie

Jednym z najbardziej znanych i najszybszych algorytmów sortowania jest tzw. **sortowanie szybkie** (ang. *quick sort*), wynalezione i opublikowane w roku 1962 przez C.A.R. Hoare'a. Algorytm działa na zasadzie podziału ciągu na dwie części:

- 1) wybieramy dowolnie jeden element ciągu, tzw. **oś** (ang. *pivot*),
- 2) dzielimy ciąg na grupę elementów mniejszych od osi lub jej równych i grupę elementów większych od osi lub jej równych,
- 3) sortujemy w ten sam sposób obie grupy osobno.

Algorytm jest zbudowany w oparciu o metodę „dziel i zwyciężaj”, a naturalną jego konstrukcją programistyczną jest rekurencja. Sortowanie odbywa się w nim poprzez rozdzielanie elementów ciągu względem wybranej osi podziału, którą może być dowolny element ciągu. Kolejność elementów w obydwu grupach nie jest określona. Istotne jest jedynie to, że pierwsza zawiera elementy o mniejszych wartościach kluczy, a druga o większych (por. rys. 2.3). Dalszy podział obu grup, aż do uzyskania grup o długości 1, prowadzi do uporządkowania całego ciągu, ponieważ wszystkie grupy są uporządkowane względem siebie.



Rys. 2.3. Podział ciągu w sortowaniu szybkim

Najlepiej byłoby, gdyby w charakterze osi wybierać tzw. **medianę**, która jest środkowym co do wartości klucza elementem ciągu, czyli środkowym elementem ciągu uporządkowanego. Wybór mediany prowadzi do równego podziału ciągu na elementy mniejsze i większe. Dla wygody wybiera się często pierwszy lub środkowy element ciągu. Podział ciągu  $a_1, a_2, \dots, a_n$  względem środkowego elementu można opisać w pseudojęzyku programowania następująco:

```
i := 1
j := n
oś := a[(1+n) div 2].key
repeat
  while a[i].key < oś do i := i+1
  while oś < a[j].key do j := j-1
  if i ≤ j then
    Zamień a[i] z a[j]
    i := i+1
    j := j-1
until i > j
```

W pierwszej pętli **while** przebiegamy ciąg od strony lewej do prawej, zaś w drugiej od strony prawej do lewej. W obydwu pętlach *oś* pełni rolę wartownika, a elementy  $a_i$  i  $a_j$ , które spowodowały ich zakończenie, spełniają nierówność  $a_i.key \geq oś \geq a_j.key$ . W przypadku ostrej nierówności zaburzają podział ciągu na elementy mniejsze i większe, dlatego je zamieniamy. Proces przeglądania i zamiany kontynuujemy w pętli **repeat ... until**, aż wszystkie elementy zostaną uwzględnione i w rezultacie rozdzielone na elementy mniejsze i większe.

Podział ciągu na elementy mniejsze i większe względem środkowego elementu ilustruje tabela 2.3. Postępowanie rozpoczyna się od obu końców ciągu i polega na przesuwaniu się ku środkowi. Strzałki wskazują miejsca zatrzymań na elementach, które są zamieniane. Gdy droga z lewej strony zetknie się z drogą z prawej strony, postępowanie kończy się.

**Tabela 2.3.** Przykład podziału ciągu 8-elementowego (oś = 20)

1	2	3	4	5	6	7	8
15	14	24	20	18	25	12	22
		↑				↑	
15	14	12	20	18	25	24	22
			↑	↑			
15	14	12	18	20	25	24	22
			↑				

Przejdźmy teraz do sformułowania algorytmu sortowania szybkiego. Po podzieleniu ciągu na grupy elementów mniejszych i większych dzielimy dalej obie grupy dopóty, dopóki nie otrzymamy grup złożonych tylko z jednego elementu. Postępowanie to możemy opisać w postaci procedury:

```

procedure QUICK-SORT(a[1..n])
  procedure SORTUJ(L, P)
    i := L
    j := P
    oś := a[(L+P) div 2].key
    repeat
      while a[i].key < oś do i := i+1
      while oś < a[j].key do j := j-1
      if i ≤ j then
        tmp := a[i]
        a[i] := a[j]
        a[j] := tmp
        i := i+1
        j := j-1
    until i > j
    if L < j then SORTUJ(L, j)
    if i < P then SORTUJ(i, P)
if n > 1 then SORTUJ(1, n)

```

Zasadniczą pracę wykonuje w niej procedura *SORTUJ*, która dzieli ciąg elementów o indeksach od *L* do *P* (lewy i prawy koniec), a następnie wywołuje siebie rekurencyjnie, gdy uzyskane grupy elementów mniejszych i większych zawierają

więcej niż jeden element. Rola procedury *QUICK-SORT* sprowadza się jedynie do wywołania procedury *SORTUJ* dla parametrów 1 i  $n$ . Implementacja algorytmu sortowania szybkiego w Object Pascalu ma w przypadku ciągu liczb całkowitych reprezentowanego przez tablicę otwartą postać następującą:

```
procedure QuickSort(var a: array of integer);

  procedure Sortuj(L, P: integer);
  var
    i, j, os, tmp: integer;
  begin
    i := L;
    j := P;
    os := a[(L+P) div 2];
    repeat
      while a[i] < os do Inc(i);
      while os < a[j] do Dec(j);
      if i <= j then
        begin
          tmp := a[i];
          a[i] := a[j];
          a[j] := tmp;
          Inc(i);
          Dec(j);
        end;
    until i > j;
    if L < j then Sortuj(L, j);
    if i < P then Sortuj(i, P);
  end;

begin
  Sortuj(0, High(a));
end;
```

Zajmijmy się analizą złożoności czasowej sortowania szybkiego. Zauważmy najpierw, że liczba porównań osi z elementami tablicy  $a[1..n]$  w podziale jej na elementy mniejsze i większe wynosi  $n$  lub  $n + 1$ . Dodatkowe porównanie ma miejsce wtedy, gdy dwie drogi przeglądania elementów, od strony lewej do prawej i od strony prawej do lewej, kończą się na tym samym elemencie, czego konsekwencją jest dwukrotne porównanie go z osią. Ponieważ czas wykonania algorytmu zależy nie tylko od rozmiaru danych  $n$ , ale w dużej mierze od nich samych, a właściwie od stopnia zrównoważenia podziałów danych na elementy mniejsze i większe, rozpatrzmy dwa skrajne przypadki.

Przypadek pesymistyczny zachodzi wtedy, gdy podział prowadzi do jednej grupy zawierającej jeden element i drugiej złożonej z  $n - 1$  elementów. Jeżeli takie podziały powstają w każdym kroku, to złożoność czasową algorytmu można opisać za pomocą następującego równania rekurencyjnego:

$$T(n) = \begin{cases} 0 & (n = 1) \\ T(n-1) + n + 1 & (n > 1) \end{cases} \quad (2.3)$$

Składnik  $n + 1$  wyraża czas podziału ciągu  $n$ -elementowego na elementy mniejsze i większe. Równanie (2.3) rozwiązujemy następująco:

$$\begin{aligned} T(n) &= T(n-1) + n + 1 = T(n-2) + n + (n+1) = \\ &= T(n-3) + (n-1) + n + (n+1) = \dots \\ &= T(1) + 3 + 4 + \dots + (n-1) + n + (n+1) = \\ &= \frac{(n-1)(n+4)}{2} = \frac{1}{2}n^2 + \frac{3}{2}n - 2 = O(n^2) \end{aligned}$$

Rozpatrzmy teraz przypadek optymistyczny, w którym zawsze za oś podziału udaje się wybrać medianę, czyli gdy za każdym razem podział ciągu na dwie grupy elementów jest zrównoważony. Równanie rekurencyjne przyjmuje wtedy postać:

$$T(n) = \begin{cases} 0 & (n = 1) \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n + 1 & (n > 1) \end{cases} \quad (2.4)$$

W celu rozwiązania tej rekurencji posłużymy się metodą podstawienia zmiennej. Przyjmujemy mianowicie, że  $n = 2^k$ . Otrzymujemy wtedy:

$$\begin{aligned} T(n) &= T(2^k) = 2T(2^{k-1}) + 2^k + 1 = \\ &= 2[2T(2^{k-2}) + 2^{k-1} + 1] + 2^k + 1 = \\ &= 2^2 T(2^{k-2}) + 2 \cdot 2^k + 2 + 1 = \\ &= 2^2 [2T(2^{k-3}) + 2^{k-2} + 1] + 2 \cdot 2^k + 2 + 1 = \\ &= 2^3 T(2^{k-3}) + 3 \cdot 2^k + 2^2 + 2 + 1 = \dots = \\ &= 2^k T(2^0) + k \cdot 2^k + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2 + 1 = \\ &= k \cdot 2^k + \frac{2^k - 1}{2 - 1} = k \cdot 2^k + 2^k - 1 = \\ &= n \log n + n - 1 = n \log n + O(n) \end{aligned}$$

Korzystając z równania (2.4), możemy przez indukcję matematyczną udowodnić (zob. ćw. 2.7), że oszacowanie to jest prawdziwe dla dowolnego  $n$ .

Z przeprowadzonych rozważań wynika, że w przypadku pesymistycznym, w którym podziały są maksymalnie nie zrównoważone, algorytm działa w czasie  $O(n^2)$ , a w przypadku optymistycznym, w którym podziały są zrównoważone, działa o wiele szybciej, bo w czasie  $O(n \log n)$ . Interesujące jest, że dla przeciętnych danych złożoność czasowa algorytmu jest bliższa jego złożoności czasowej w przypadku optymistycznym niż w pesymistycznym. Dowodzi się, że nawet gdy podziały są dokonywane w proporcji 9:1, która sprawia wrażenie mocno niezrównoważonej, algorytm działa w czasie  $O(n \log n)$  [7], a dla losowych danych jego efektywność jest niewiele gorsza od optymalnej [3, 19]:

$$A(n) \approx 2 \ln 2 \cdot (n+1) \log n \approx 1,38 \cdot n \log n + O(n).$$

Sortowanie szybkie zostało gruntownie przebadane. Algorytm najwygodniej jest zapisać rekurencyjnie, nieco trudniej iteracyjnie ze względu na jawną organizację stosu służącego do zapamiętania lewego i prawego indeksu wszystkich grup elementów, które mają być dalej dzielone [4, 22, 28]. Szczególnie ważnym problemem jest wybór osi i sposób podziału ciągu na elementy mniejsze i większe. Wiele różnych pomysłów rozwiązania tego problemu spowodowało powstanie niezliczonych wersji algorytmu sortowania szybkiego [1, 2, 3, 13, 17].

## 2.5. Sortowanie kopcowe

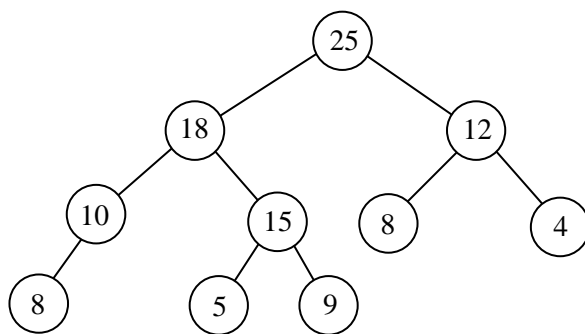
Chociaż algorytm sortowania szybkiego działa w przypadku ciągów losowych nieporównywalnie szybciej niż proste algorytmy sortowania, takie jak sortowanie bąbelkowe, sortowanie przez selekcję czy przez wstawianie, dając wynik w czasie liniowo-logarytmicznym, jego efektywność może w przypadkach niekorzystnych spaść do kwadratowej. Mówiąc inaczej, sortowanie szybkie jest pewnego rodzaju loterią, od której zależy szybkość uzyskania rozwiązania.

Omawiane w niniejszym podrozdziale **sortowanie przez kopcowanie** (ang. *heap sort*), zwane krócej **sortowaniem kopcowym** lub **stogowym**, ma złożoność czasową  $O(n \log n)$  w każdym przypadku. Algorytm działa na reprezentowanej przez tablicę strukturze zwanej **kopcem** lub **stogiem**.

Kopiec (stóg) jest **drzewem binarnym** składającym się z **wierzchołków**, zwanych również **węzłami**, połączonych **krawędziami**. Dokładnie jeden wierzchołek drzewa, zwany **korzeniem**, jest wyróżniony. Do każdego wierzchołka, z wyjątkiem korzenia, dochodzi jedna krawędź, a z każdego wierzchołka mogą wychodzić co najwyżej dwie krawędzie. Stąd każdy wierzchołek, który nie jest korzeniem, ma dokładnie jednego poprzednika, każdy może też mieć co najwyżej dwóch następników (rys. 2.4). Wierzchołek, który nie ma następnika, nazywamy **liściem**.

Kopiec jest drzewem binarnym, w którego wierzchołkach znajdują się elementy pewnego, dającego się uporządkować zbioru takie, że jeżeli wierzchołek  $v$  jest następnikiem wierzchołka  $u$ , to element znajdujący się w wierzchołku  $v$  jest nie większy od elementu znajdującego się w wierzchołku  $u$ . Wynika stąd następujące **uporządkowanie kopcowe** (por. rys. 2.4):

- w korzeniu kopca znajduje się największy element lub jeden z największych elementów, gdy jest ich kilka,
- na ścieżkach kopca, prowadzących wzdłuż krawędzi od korzenia do liścia, elementy są uporządkowane nierosnąco.



**Rys. 2.4.** Przykład drzewa binarnego będącego kopcem

Kopiec nazywamy **zupełnym**, gdy ma wszystkie poziomy całkowicie wypełnione, z wyjątkiem co najwyżej ostatniego, spójnie wypełnionego od strony lewej do pewnego miejsca (por. rys. 2.5). Kopiec zupełny ma więc na poziomie 0 jeden wierzchołek (korzeń), na poziomie 1 dwa wierzchołki, na poziomie 2 cztery wierzchołki itd., na ostatnim poziomie, o numerze  $h$ , co najmniej jeden, a co najwyżej  $2^h$  wierzchołków. Numer ostatniego poziomu jest równy **wysokości** kopca, którą jest długość najdłuższej ścieżki prowadzącej od korzenia do liścia, wyrażona jako liczba krawędzi tej ścieżki. Wysokość kopca zupełnego można łatwo wyznaczyć na podstawie liczby jego wierzchołków  $n$ . Mamy mianowicie:

$$2^0 + 2^1 + \dots + 2^{h-1} + 1 \leq n \leq 2^0 + 2^1 + \dots + 2^{h-1} + 2^h,$$

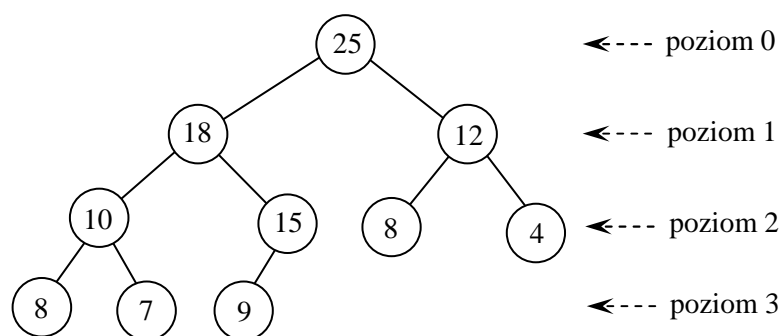
czyli

$$2^h \leq n < 2^{h+1}.$$

Stąd otrzymujemy:

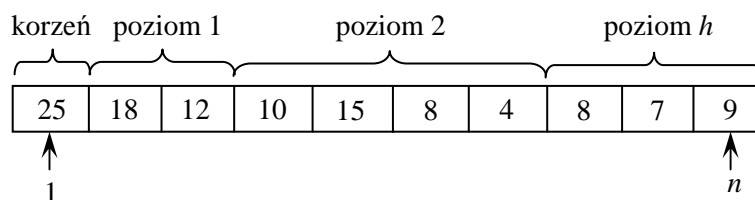
$$h = \lfloor \log n \rfloor. \quad (2.5)$$



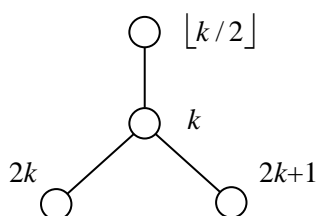


Rys. 2.5. Przykład kopca zupełnego

Algorytm sortowania kopcowego wynalazł w roku 1964 J. Williams, a ulepszył w tym samym roku R.W. Floyd, wprowadzając zręczny sposób budowy kopca zupełnego *in situ*. W tablicy reprezentującej kopiec zupełny najpierw jest zapisany korzeń kopca, a następnie, w kolejności od strony lewej do prawej, wierzchołki poziomu 1, potem wierzchołki poziomu 2 itd. (zob. rys. 2.6). Zapis ten prowadzi do specyficznej indeksacji sąsiadujących wierzchołków. I tak, poprzednik wierzchołka  $k$  różnego od korzenia ma indeks  $\lfloor k/2 \rfloor$ , a następniki wierzchołka  $k$ , o ile istnieją, mają indeksy: lewy  $2k$ , prawy  $2k + 1$  (por. rys. 2.7).



Rys. 2.6. Reprezentacja tablicowa kopca przedstawionego na rys. 2.5

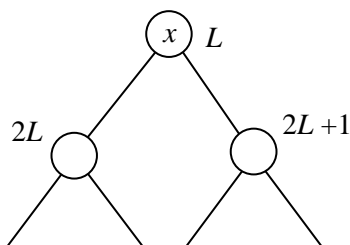


Rys. 2.7. Indeksacja sąsiednich wierzchołków kopca

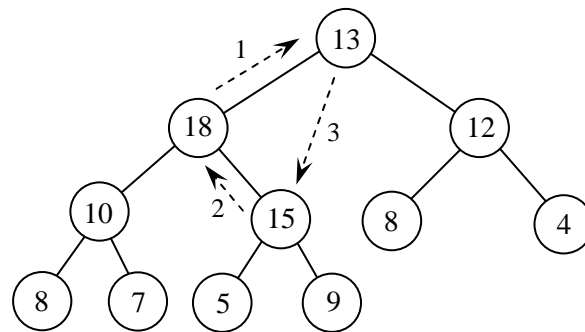
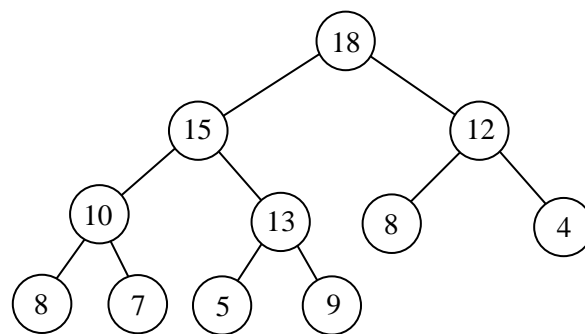
**Idea sortowania kopcowego.** Załóżmy, że elementy  $a_1, a_2, \dots, a_n$  są ustawione w kopiec. Wówczas element  $a_1$  stoi na szczycie kopca, jako korzeń jest więc największym elementem ciągu. Wynika stąd, że jeżeli potrafilibyśmy w jakiś sposób ustawić wszystkie elementy ciągu w kopiec, to po zamianie  $a_1$  z  $a_n$ , czyli po przestawieniu korzenia kopca z jego ostatnim liściem, umieścilibyśmy największy element ciągu w miejscu  $a_n$ . Następnie moglibyśmy powtórzyć całe postępowanie dla elementów  $a_1, a_2, \dots, a_{n-1}$ , tj. ustawić je zgodnie z porządkiem kopcowym, po czym zamienić  $a_1$  z  $a_{n-1}$ . Tym razem drugi co do wielkości element ciągu znalazłby się na właściwym miejscu. Postępując tak dalej, za każdym razem z mniejszą o jeden liczbą elementów, uporządkowalibyśmy cały ciąg.

Okazuje się, że wystarczy tylko raz zbudować kopiec z wszystkich elementów ciągu (operacja *construct*), a potem po każdej zamianie korzenia z ostatnim liściem jedynie przywrócić uporządkowanie kopcowe pomniejszonemu o ten liść drzewu (operacja *sort*). W obydwu operacjach wykorzystuje się specyficzną metodę przesiewania, która polega na przesuwaniu zaburzającego uporządkowanie kopcowe korzenia w głąb drzewa, aż przywrócony zostanie porządek kopcowy.

**Algorytm przesiewania.** Załóżmy, że dla pewnych wartości indeksów  $L$  i  $P$  (lewy i prawy) ciąg  $a_{L+1}, a_{L+2}, \dots, a_P$  zawiera dwa kopce o korzeniach  $a_{2L}$  i  $a_{2L+1}$ . Strukturę tę rozszerzamy w lewo o nowy element  $x = a_L$  tak, by ciąg  $a_L, a_{L+1}, \dots, a_P$  zawierał kopiec o korzeniu  $a_L$  łączącym oba kopce (zob. rys. 2.8). Jednak po tej operacji uporządkowanie kopcowe rozszerzonego drzewa może dla korzenia  $x$  zostać zaburzone, należy je więc przywrócić. I tak, jeżeli nie zachodzi nierówność  $x \geq \max(a_{2L}, a_{2L+1})$ , wierzchołek  $x$  zamieniamy z większym z jego następników  $a_{2L}$  i  $a_{2L+1}$  (por. rys. 2.9). W ten sposób zostanie on przesunięty o jeden poziom w dół, zaś jego większy następnik o poziom w górę. Postępowanie to kontynuujemy dla kolejnych następników przesuniętego wierzchołka  $x$  dopóty, dopóki nie zostanie przywrócona struktura kopcowa.



**Rys. 2.8.** Dołączanie nowego wierzchołka do kopca

a) przesiewanie wierzchołka  $x = 13$ 

b) wynik operacji przesiewania

**Rys. 2.9.** Przywracanie uporządkowania kopcowego

Operację przesiewania wierzchołka w głąb kopca możemy opisać w pseudokodzie programowania następująco:

```

procedure Przesiewaj(L, P)
    i := L;
    j := 2*i;
    x := a[i];
    while j ≤ P do
        if j < P then
            if a[j].key < a[j+1].key then j := j+1
            if x.key ≥ a[j].key then break
            a[i] := a[j]
            i := j
            j := 2*i
    a[i] := x

```

Indeksy  $i, j$  wskazują wierzchołki, które mają być zamienione podczas każdego kroku przesiewania. Pętla **while** zostaje przerwana za pomocą instrukcji **break** albo kończy się naturalnie w wyniku niespełnienia warunku  $j \leq P$ . W pierwszym przypadku wierzchołek  $x$  stanie się korzeniem stosownego podkopca, ponieważ jest większy od obydwu następników (jednego, gdy drugiego nie ma) lub jest im równy. W drugim przypadku  $x$  zajmie pozycję liścia.

**Operacja construct** (konstrukcja kopca). Zauważmy, że elementy podciągu  $a_{n/2+1}, a_{n/2+2}, \dots, a_n$  są liśćmi drzewa binarnego o wierzchołkach  $a_1, a_2, \dots, a_n$ . Zatem nie mają następników, czyli są kopcami jednoelementowymi. Możemy więc w kolejnych krokach złożyć z nich strukturę rozszerzać w lewo o nowe wierzchołki  $a_L$  ( $L = n/2, n/2-1, \dots, 1$ ) poprzez ustawianie ich we właściwych miejscach za pomocą procedury przesiewania. W rezultacie operacja konstrukcji kopca sprowadza się do następującej pętli:

```
for L:=n div 2 downto 1 do
  Przesiewaj(L, n)
```

Proces budowania kopca z nieuporządkowanej tablicy złożonej z 8 elementów ilustruje tabela 2.4. Strzałki pokazują przemieszczanie się większych wierzchołków o poziom wyżej podczas przesiewania mniejszych wierzchołków w głąb kopca.

**Tabela 2.4.** Przykład budowy kopca 8-elementowego

Pocz.	$L = 4$	$L = 3$	$L = 2$	$L = 1$
14	14	14	14	32
25	25	25	32	27
7	7	8	8	8
10	27	27	27	14
32	32	32	25	25
8	8	7	7	7
2	2	2	2	2
27	10	10	10	10

**Operacja sort** (sortowanie kopca). Aby posortować elementy kopca, należy wykonać  $n - 1$  kroków, z których każdy obejmuje zamianę korzenia kopca z ostatnim liściem, a następnie skrócenie kopca o ten liść i przywrócenie uporządkowania kopcowego, które mogło zostać zaburzone przez element zamieszczony w korzeniu. Wykorzystując procedurę przesiewania, operację sortowania kopca możemy zapisać następująco:

















































```

for P:=n downto 2 do
  tmp := a[1]
  a[1] := a[P]
  a[P] := tmp
  Przesiewaj(1, P-1)

```

Przebieg procesu sortowania kopca, którego budowę przedstawiono w tabeli 2.4, ilustruje tabela 2.5. Strzałki pokazują przemieszczanie się większych wierzchołków kopca podczas przesiewania wykonywanego po każdej zamianie korzenia z ostatnim liściem i pomniejszeniu kopca.

**Tabela 2.5.** Przykład procesu sortowania kopca 8-elementowego

Pocz.	$P = 8$	$P = 7$	$P = 6$	$P = 5$	$P = 4$	$P = 3$	$P = 2$						
32		27		25		14		10		8		7	2
27		25		14		10		7		2		8	7
8		8		2		7		25		14		10	10
14		10		7		14		10		7		2	7
25		14		10		7		25		14		10	10
7		2		7		25		14		10		7	2
2		7		25		14		10		7		2	7
10		7		2		7		25		14		10	10

Przejdźmy wreszcie do sprecyzowania algorytmu sortowania kopcowego, łącząc przytoczone wyżej trzy fragmenty kodu w jeden podprogram. Kierując się względami efektywnościowymi, wprowadzimy proste modyfikacje. Mianowicie, występujące w procedurze *Przesiewaj* parametry  $L$ ,  $P$  i zmienną lokalną  $x$  zadeklarujemy na zewnątrz niej, a instrukcje **for** w operacjach *construct* i *sort* zastąpimy bardziej elastycznymi instrukcjami **while**. Ponadto w fazie sortowania zamiast pomocniczej zmiennej *tmp* użyjemy wspomnianej zmiennej  $x$ . W rezultacie otrzymamy następujący kod w pseudojęzyku programowania:

```

procedure HEAP-SORT(a[1..n])

  var L, P, x

  procedure Przesiewaj
    i := L;
    j := 2*i;
    x := a[i];
    while j ≤ P do
      if j < P then
        if a[j].key < a[j+1].key then j := j+1

```

```

        if x.key ≥ a[j].key then break
        a[i] := a[j]
        i := j
        j := 2*i
        a[i] := x

P := n                      // Construct
L := (n div 2)+1
while L>1 do
    L := L-1
    Przesiewaj
while P>1 do                // Sort
    x := a[1]
    a[1] := a[P]
    a[P] := x
    P := P-1
    Przesiewaj

```

Zanim przedstawimy implementację algorytmu *HEAP-SORT* w języku Object Pascal zauważmy, że numeracja elementów tablicy otwartej od 0 do  $n - 1$  zamiast od 1 do  $n$  pociąga za sobą zmianę indeksacji sąsiadujących wierzchołków kopca. I tak, jeżeli wierzchołek  $k$  nie jest korzeniem kopca, jego poprzednik ma indeks  $\lfloor (k-1)/2 \rfloor$ , a jeżeli nie jest liściem, jego lewy następnik ma indeks  $2k + 1$ , a prawy  $2k + 2$  (gdy istnieje). Indeksacja ta prowadzi do następującego sformułowania algorytm sortowania kopcowego tablicy otwartej o elementach całkowitych:

```

procedure HeapSort(var a: array of integer);

var
    L, P, x: integer;

procedure Przesiewaj;
var
    i, j: integer;
begin
    i := L;
    j := 2*i+1;
    x := a[i];
    while j<=P do
    begin
        if j<P then
            if a[j] < a[j+1] then Inc(j);
            if x >= a[j] then Break;
            a[i] := a[j];
            i := j;
            j := 2*i+1;
        end;
        a[i] := x;
    end;

```

```

begin
  P := High(a);           // Construct
  L := Length(a) div 2;
  while L > 0 do
    begin
      Dec(L);
      Przesiewaj;
    end;
  while P > 0 do           // Sort
    begin
      x := a[0];
      a[0] := a[P];
      a[P] := x;
      Dec(P);
      Przesiewaj;
    end;
  end;
end;

```

Analiza złożoności czasowej sortowania kopcowego wiąże się z wyznaczeniem liczby przemieszczeń opuszczanego z wierzchołka w głąb kopca elementu w procedurze *Przesiewaj*. Jest oczywiste, że liczba ta nie może przekroczyć wysokości kopca określonej wzorem (2.5), tj. wartości  $\log n$ . Z kolei operacje *construct* i *sort*, wykorzystujące procedurę *Przesiewaj*, wykonują się w czasie liniowym. Wynika stąd, że algorytm działa w czasie  $O(n \log n)$ .

W celu znalezienia lepszego oszacowania, uwzględniającego współczynnik proporcjonalności przy  $n \log n$ , wyznaczmy liczbę wykonań pętli **while** w procedurze *Przesiewaj*. Na początku zmienna  $i$  przyjmuje wartość  $L$ , a po kolejnych krokach iteracji co najmniej dwa razy większą niż poprzednio:

dla 1 iteracji	$i = L$
dla 2 iteracji	$i \geq 2L$
dla 3 iteracji	$i \geq 2^2 L$
.....	
dla $k$ iteracji	$i \geq 2^{k-1} L$

Warunkiem koniecznym wykonania  $k$  iteracji jest spełnienie nierówności:

$$P \geq j = 2i \geq 2^k L,$$

czyli

$$k \leq \log \frac{P}{L}.$$

Zatem liczba wykonań pętli w operacji przesiewania wynosi co najwyżej  $\log \frac{P}{L}$ .

Operacja budowy kopca (*construct*) wykonywana jest przy  $P = n$  w pętli dla  $L = n/2, n/2 - 1, \dots, 1$ , więc wymagana w tej fazie liczba operacji przesiewających wynosi co najwyżej:

$$C(n) = \sum_{L=1}^{n/2} \log \frac{n}{L} = \frac{n}{2} \log n - \sum_{L=1}^{n/2} \log L$$

Z kolei w operacji sortowania (*sort*) przesiewania są wykonywane przy  $L = 1$  dla  $P = n-1, n-2, \dots, 1$ . Wymagana liczba iteracji wynosi więc wtedy co najwyżej:

$$S(n) = \sum_{P=1}^{n-1} \log P$$

W każdej iteracji przesiewania mogą wystąpić dwa porównania elementów tablicy. Zatem pesymistyczna złożoność czasowa sortowania kopcowego, wyrażona jako liczba porównań elementów, daje się oszacować następująco:

$$T(n) = 2[C(n) + S(n)] = n \log n + 2 \sum_{k=\frac{n}{2}+1}^{n-1} \log k \leq n \log n + \frac{2n}{2} \log(n-1)$$

Ostatecznie otrzymujemy:

$$T(n) \leq 2n \log n = O(n \log n)$$

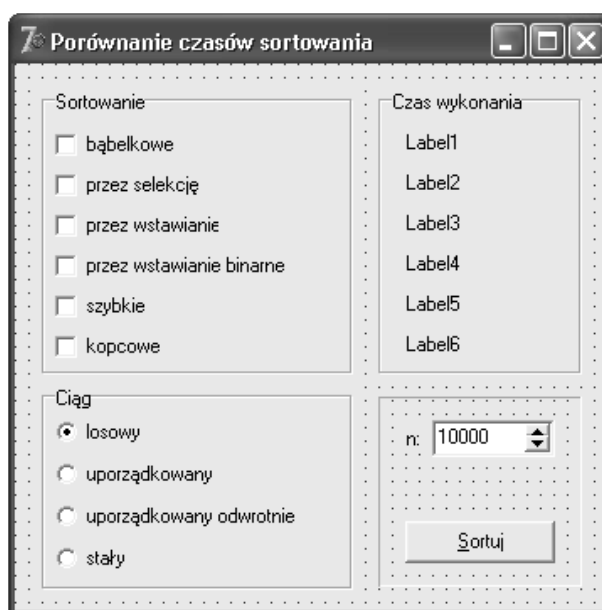
Algorytm sortowania kopcowego charakteryzuje się więc doskonałą efektywnością nawet w przypadku pesymistycznym, co jest jego najważniejszą zaletą. Takiego czasu wykonania nie gwarantował żaden z omawianych uprzednio algorytmów, oprócz binarnego *insertion sort*, którego negatywną cechą jest duża liczba przesunięć elementów. Wyniki przeprowadzonych badań wykazują, że w średnim przypadku współczynnik przy  $n \log n$  jest bliski 2 [17, 19]. Algorytm *quick sort* ma przewagę dla danych losowych, bowiem jego efektywność wynosi wtedy około  $1,38 \cdot n \log n$ , ale jest pewnego rodzaju loterią.

## 2.6. Porównanie algorytmów sortowania wewnętrznego

Zbudujemy teraz aplikację okienkową w Delphi, która pozwoli na praktyczne porównanie czasu wykonania omówionych algorytmów sortowania wewnętrznego tablic, których elementami są liczby całkowite. Czytelnik może ją łatwo rozbudować o inne algorytmy i dostosowywać do innych rodzajów danych spotykanych



w rzeczywistych warunkach, zastępując np. liczby dużymi rekordami, w których kluczami są łańcuchy lub wartości dowolnego typu, w odniesieniu do którego można zdefiniować relację porządkującą  $\leq$  lub  $\geq$ . Projekt formularza okna aplikacji jest pokazany na rysunku 2.10.



**Rys. 2.10.** Projekt formularza aplikacji porównującej czasy wykonania wybranych algorytmów sortowania wewnętrznego

Na formularzu umieszczone są obok siebie dwa komponenty *GroupBox* (pole grupy) o właściwościach *Caption* ustawionych na *Sortowanie* i *Czas wykonania*. Na pierwszym polu rozmieszczonych jest pionowo sześć komponentów *CheckBox* (pole wyboru), a na drugim sześć etykiet. Pola wyboru umożliwią wybór dowolnych z sześciu algorytmów sortowania, których czas wykonania chcemy zmierzyć. Ich właściwości *Caption* określają nazwy algorytmów. Etykiety na razie pozostają niezmienione. Będziemy je zmieniać programowo, umieszczając na nich napisy wyrażające zmierzony czas wykonania wybranych algorytmów.

W dolnej części formularza znajduje się komponent *RadioGroup* (grupa przycisków opcji), którego właściwość *Caption* ma wartość *Ciąg*, właściwość *Items* jest listą łańcuchów: *losowy*, *uporządkowany*, *uporządkowany odwrotnie* i *stały*, zaś właściwość *ItemIndex* ma wartość zero. Przyciski opcji pozwolą nam zmieniać charakter uporządkowania ciągu, a tym samym sprawdzać, jak wpływa ono na szybkość sortowania. Wstępnie wybrany jest ciąg losowy. Obok przycisków opcji

znajduje się grupka trzech komponentów otoczona ozdobną ramką *Bevel*: *SpinEdit* (pole edycji) wraz z etykietą i przycisk z napisem *Sortuj*. Pole edycji posłuży nam do określania długości ciągu. Jego właściwości *MinValue*, *MaxValue* i *Value* mają wartości: 1, 500000 i 10000. Oznacza to, że wstępnie długość ciągu wynosi 10 tys. i będzie ją można zmieniać w zakresie od 1 do 500 tys.

Do projektu aplikacji dodajemy sześć modułów zawierających kod algorytmów sortowania. Zakładamy, że Czytelnik sam zbuduje moduły *SelectUnit*, *InsertUnit*, *BinInsertUnit*, *QuickUnit* i *HeapUnit*, w których umieści zaprezentowane w tym rozdziale podprogramy *SelectionSort*, *InsertionSort*, *BinInsertionSort*, *QuickSort* i *HeapSort*, podobnie jak w rozdziale 1. zbudowano moduł *BubbleUnit* z podprogramem *BubbleSort*. Najwygodniej jest umieścić wszystkie moduły w katalogu z plikami aplikacji. Oczywiście, listę **uses** modułu formularza należy rozszerzyć o nazwy dołączonych modułów. Przyjmijmy, że moduł formularza zapisujemy w pliku *MainUnit.pas*, a projekt główny aplikacji w pliku *Porownanie.dpr*.

Zadeklarujmy teraz w sekcji publicznej klasy *TForm1* bezparametrową metodę *UstawEtykiety*. Jej pusty szkielec, którego wygenerowanie wymuszamy na Delphi za pomocą klawiszy *Ctrl+Shift+C*, uzupełniamy do postaci:

```
procedure TForm1.UstawEtykiety;
begin
    Label1.Caption := 'x';
    Label2.Caption := 'x';
    Label3.Caption := 'x';
    Label4.Caption := 'x';
    Label5.Caption := 'x';
    Label6.Caption := 'x';
end;
```

Definiujemy też procedurę obsługi zdarzenia *OnCreate* formularza, która powinna zmienić napisy na etykietach i uaktywnić generator liczb losowych:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    UstawEtykiety;
    Randomize;
end;
```

Dalsza rozbudowa kodu aplikacji wymaga zdefiniowania tablicy reprezentującej sortowany ciąg liczb całkowitych. Z uwagi na zapewnienie powtarzalności danych potrzebna jest jeszcze jedna tablica, w której będzie przechowywany ciąg wejściowy, podczas gdy ten sam ciąg będzie sortowany w drugiej tablicy. Zatem deklarujemy w sekcji prywatnej klasy *TForm1* dwie tablice otwarte:

```
a, b: array of integer;
```

Możemy teraz w sekcji publicznej klasy zadeklarować metodę *GenerujCiag*, a utworzony przez Delphi pusty szkielet metody uzupełnić, precyzując operację generowania ciągu zgodnie z dokonanymi przez użytkownika ustawieniami komponentów *SpinEdit1* i *RadioGroup1*. Pierwszy z nich określa długość ciągu, a drugi sposób jego uporządkowania (losowy, rosnący, malejący, stały):

```
procedure TForm1.GenerujCiag;
var
  k: integer;
begin
  SetLength(a, SpinEdit1.Value);
  for k:=0 to High(a) do
    case RadioGroup1.ItemIndex of
      0: a[k] := Random(MaxInt);
      1: a[k] := k;
      2: a[k] := High(a)-k;
      3: a[k] := 1;
    end;
  end;
```

Znaleźliśmy się w sytuacji, w której musimy zaprogramować operację pomiaru czasu wykonania dowolnego z algorytmów sortowania. Najwygodniej jest sformułować ją w postaci podprogramu, którego parametrem jest procedura sortująca. Rozwiązanie takie wymaga zdefiniowania typu proceduralnego [5, 14] reprezentującego ogólną procedurę sortowania ciągu liczb całkowitych:

```
type
  TSortProc = procedure(var a: array of integer);
```

Każda z dołączonych do aplikacji procedur sortowania jest kompatybilna z typem *TSortProc*, ponieważ posiada ten sam parametr, którym jest przekazywana przez zmienną tablica otwarta o elementach całkowitych.

Możemy teraz rozbudować klasę *TForm1* o kolejną metodę, która mierzy czas wykonania określonego algorytmu sortowania i po sformatowaniu za pomocą funkcji *FormatDateTime* wyświetla jako napis na etykiecie:

```
procedure TForm1.CzasSort(Czas: TLabel; Sort: TSortProc);
var
  Start: TDateTime;
begin
  b := Copy(a, 0, Length(a));
  Start := Time;
  Sort(b);
  Czas.Caption := FormatDateTime('hh:nn:ss:zzz', Time-Start);
end;
```

Warto zwrócić uwagę, że w metodzie *CzasSort* ciąg jest najpierw kopiowany do drugiej tablicy, a potem jest w niej sortowany. Dzięki temu możemy sprawdzić efektywność czasową różnych algorytmów dla tych samych danych.

Możemy wreszcie zdefiniować procedurę obsługi zdarzenia *OnClick* przycisku *Sortuj*. Jej zadaniem jest ustawienie wszystkich sześciu etykiet w stan początkowy, wygenerowanie ciągu liczb całkowitych oraz zmierzenie i wyświetlenie czasu sortowania go wybranymi algorytmami:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    UstawEtykiety;
    GenerujCiag;
    if CheckBox1.Checked then CzasSort(Label1, BubbleSort);
    if CheckBox2.Checked then CzasSort(Label2, SelectionSort);
    if CheckBox3.Checked then CzasSort(Label3, InsertionSort);
    if CheckBox4.Checked then CzasSort(Label4, BinInsertionSort);
    if CheckBox5.Checked then CzasSort(Label5, QuickSort);
    if CheckBox6.Checked then CzasSort(Label6, HeapSort);
end;
```

Rysunek 2.11 przedstawia przykładowy wynik wykonania utworzonej aplikacji w przypadku procesora 667 MHz i ciągu losowego o długości 100 tys. elementów.

Sortowanie	Czas wykonania
<input checked="" type="checkbox"/> bąbelkowe	00:01:56.428
<input checked="" type="checkbox"/> przez selekcję	00:01:05.814
<input checked="" type="checkbox"/> przez wstawianie	00:00:32.607
<input checked="" type="checkbox"/> przez wstawianie binarne	00:00:21.130
<input checked="" type="checkbox"/> szybkie	00:00:00.040
<input checked="" type="checkbox"/> kopcowe	00:00:00.070

Ciąg

☒ losowy  
☐ uporządkowany  
☐ uporządkowany odwrotnie  
☐ stały

n: 100000

Sortuj

**Rys. 2.11.** Efekt wykonania aplikacji porównującej czasy wykonania wybranych algorytmów sortowania wewnętrznego

Jak widać, sortowanie ciągu może dla dużych wartości  $n$  zająć sporo czasu, dlatego wypada poinformować użytkownika, że aplikacja nie pozostaje bezczynna. Zatem w metodzie *Button1Click* wyświetlamy kursor klepsydry i umieszczamy czasochłonne operacje generowania i sortowania ciągu w bloku **try ... finally**, a po ich zakończeniu lub uruchomieniu wyjątku przywracamy kursor domyślny.

Problem nie został jednak rozwiązany do końca, ponieważ aplikacja nie może podczas wykonywania algorytmów otrzymywać od Windows i przetwarzać żadnych komunikatów, w tym komunikatów dotyczących malowania okna. Oznacza to, że aktualna informacja o czasie sortowania pojawi się dopiero po zakończeniu wszystkich wybranych algorytmów sortowania, nie zaś bezpośrednio po wykonaniu każdego z osobna. Użytkownik odniesie wrażenie, że pierwszy wykonuje się bardzo długo, zaś pozostałe natychmiast. Najprostszym rozwiązaniem problemu jest wykorzystanie metody *ProcessMessages* obiektu *Application* [6], która spowoduje wstrzymanie wykonywania programu, odebranie i obsługę komunikatów, a następnie wznowienie wykonywania go. Umieszczamy zatem wywołania metody *ProcessMessages* na końcu kodu metod *GenerujCiag* i *CzasSort*. Oto ostateczna wersja modułu formularza okna głównego aplikacji:

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls, Spin,
  BubbleUnit, SelectUnit, InsertUnit, BinInsertUnit,
  QuickUnit, HeapUnit;

type
  TSortProc = procedure(var a: array of integer);

  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    CheckBox5: TCheckBox;
    CheckBox6: TCheckBox;
    GroupBox2: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    RadioGroup1: TRadioGroup;
```

```

    Bevel1: TBevel;
    Label7: TLabel;
    SpinEdit1: TSpinEdit;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
private
    a, b: array of integer;
public
    procedure UstawEtykiety;
    procedure GenerujCiag;
    procedure CzasSort(Czas: TLabel; Sort: TSortProc);
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
    UstawEtykiety;
    Randomize;
end;

procedure TForm1.UstawEtykiety;
begin
    Label1.Caption := 'x';
    Label2.Caption := 'x';
    Label3.Caption := 'x';
    Label4.Caption := 'x';
    Label5.Caption := 'x';
    Label6.Caption := 'x';
end;

procedure TForm1.GenerujCiag;
var
    k: integer;
begin
    SetLength(a, SpinEdit1.Value);
    for k:=0 to High(a) do
        case RadioGroup1.ItemIndex of
            0: a[k] := Random(MaxInt);
            1: a[k] := k;
            2: a[k] := High(a)-k;
            3: a[k] := 1;
        end;
    Application.ProcessMessages;
end;

```

```
procedure TForm1.CzasSort(Czas: TLabel; Sort: TSortProc);
var
  Start: TDateTime;
begin
  b := Copy(a, 0, Length(a));
  Start := Time;
  Sort(b);
  Czas.Caption := FormatDateTime('hh:nn:ss:zzz', Time-Start);
  Application.ProcessMessages;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  UstawEtykiety;
  Screen.Cursor := crHourGlass;
  try
    GenerujCiag;
    if CheckBox1.Checked then CzasSort(Label1, BubbleSort);
    if CheckBox2.Checked then CzasSort(Label2, SelectionSort);
    if CheckBox3.Checked then CzasSort(Label3, InsertionSort);
    if CheckBox4.Checked then CzasSort(Label4, BinInsertionSort);
    if CheckBox5.Checked then CzasSort(Label5, QuickSort);
    if CheckBox6.Checked then CzasSort(Label6, HeapSort);
  finally
    Screen.Cursor := crDefault;
  end;
end;

end.
```

Przedstawiona aplikacja umożliwia eksperymentalne porównanie efektywności czasowej opisanych w niniejszym rozdziale algorytmów sortowania. Uzyskiwane wyniki pozwalają wyraźnie odróżnić algorytmy o złożoności czasowej  $O(n^2)$  od algorytmów  $O(n \log n)$ . Sortowanie bąbelkowe jest dla większych  $n$  zdecydowanie najgorsze, a usprawnienie sortowania przez wstawianie, polegające na wykorzystaniu przeszukiwania binarnego, jest zbyt mało znaczące, by poprawiony algorytm mógł konkurować z algorytmami sortowania szybkiego lub kopcowego. Sortowanie szybkie jest średnio dwukrotnie lepsze od sortowania kopcowego, ale wymaga dodatkowej pamięci i bywa gorsze. Z kolei sortowanie kopcowe używa niezwykle interesującej struktury danych, która nadaje się do innych zastosowań.

Zwróćmy uwagę, że uzyskane wyniki dotyczą jedynie sortowania liczb, czyli samych kluczy, a nie rekordów. W rzeczywistych warunkach kluczom towarzyszą inne dane. Przemieszczanie ich wraz z kluczami podczas sortowania może mieć znaczący wpływ na efektywność czasową algorytmów [17, 28].

Należy podkreślić, że nie ma ogólnie najlepszej metody sortowania. Nawet sortowanie bąbelkowe, które wydaje się najgorsze, daje dobre wyniki w przypadku małych  $n$ . Zważywszy na prostotę implementacji, zarówno sortowanie bąbelkowe, jak i inne proste algorytmy sortowania wydają się najodpowiedniejsze dla małej liczby sortowanych elementów.

Wszystkie omówione algorytmy sortowania są oparte na porównaniach kluczy, a ich złożoność czasowa wyraża się liczbą tych porównań. Dowodzi się, że każdy algorytm, który sortuje ciąg złożony z  $n$  elementów poprzez ich porównywanie, nie może działać w czasie lepszym niż  $O(n \log n)$  [2, 3, 4, 17, 19].

## 2.7. Znajdowanie mediany

**Medianą** ciągu nazywamy jego środkowy co do wielkości element, czyli taki, który po uporządkowaniu wszystkich elementów znalazłby się dokładnie w środku ciągu. Jest to więc element, który jest większy od połowy (lub równy połowie) i mniejszy od drugiej połowy (lub równy drugiej połowie) elementów. Na przykład medianą ciągu liczb całkowitych:

22 10 121 100 85 28 12

jest liczba 28.

Zagadnienie znajdowania mediany przypomina sortowanie, ponieważ można go rozwiązać, sortując ciąg i wybierając element środkowy. Istnieją jednak szybsze algorytmy znajdowania mediany, a ponadto dają się łatwo uogólnić do rozwiązywania zagadnienia znanego pod nazwą znajdowania tzw. **statystyki pozycyjnej** (ang. *order statistic*). Problem polega na znalezieniu  $k$ -tego elementu, tzn. elementu, który po uporządkowaniu wszystkich elementów znalazłby się na pozycji  $k$ . Znajdowanie mediany jest przypadkiem szczególnym dla  $k = n/2$ . Specyfikacja problemu znajdowania statystyki pozycyjnej jest następująca:

**Dane:** Ciąg  $n$  elementów  $a_1, a_2, \dots, a_n$  i liczba całkowita  $k$  taka, że  $1 \leq k \leq n$ .

**Wynik:**  $k$ -ty co do wielkości element ciągu.

Jeden z najbardziej znanych i najszybszych algorytmów znajdowania  $k$ -tej statystyki, wynaleziony przez C.A.R. Hoare'a, polega na tym samym podziale ciągu na część elementów mniejszych i większych, co w algorytmie sortowania szybkiego. Operację podziału wykonujemy dla osi  $a[k]$ , przesuwając indeksy  $i, j$  odpowiednio od lewego i prawego końca ciągu w kierunku środka i zamieniając w razie potrzeby elementy  $a[i]$  z  $a[j]$ , aż do uzyskania nierówności  $i > j$ :



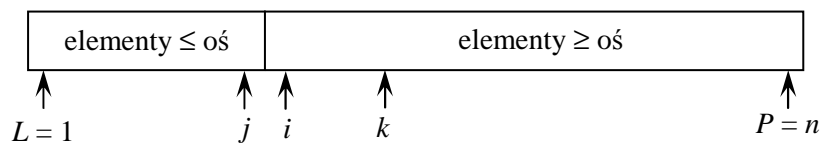
```

i := 1
j := n
oś := a[k].key
repeat
  while a[i].key < oś do i := i+1
  while oś < a[j].key do j := j-1
  if i ≤ j then
    Zamień a[i] z a[j]
    i := i+1
    j := j-1
until i > j

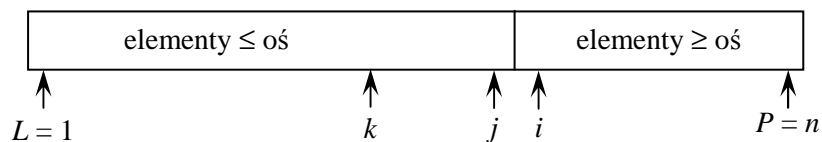
```

W rezultacie mogą wystąpić trzy przypadki (por. rys. 2.12):

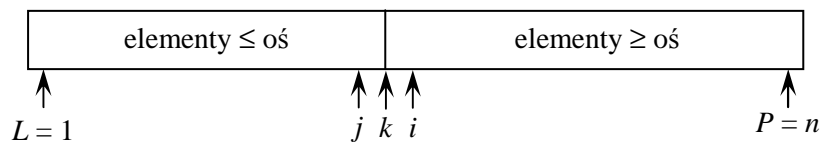
- oś jest za mała, granica podziału między dwoma częściami przebiega na lewo od indeksu  $k$ , poszukiwanie należy kontynuować w części  $a[j+1] \dots a[P]$ ;
- oś jest za duża, granica podziału między dwoma częściami przebiega na prawo od indeksu  $k$ , poszukiwanie należy kontynuować w części  $a[L] \dots a[i-1]$ ;
- oś dzieli ciąg na dwie części w żądanej proporcji, element  $a[k]$  jest szukaną statystyką pozycyjną.



a) oś za mała



b) oś za duża



c) oś właściwa

**Rys. 2.12.** Podział ciągu względem osi  $a[k]$

Możemy teraz przystąpić do sformułowania algorytmu znajdowania  $k$ -tej statystyki pozycyjnej ciągu reprezentowanego przez tablicę. Najpierw dzielimy całą tablicę na część elementów mniejszych i część elementów większych, wybierając za oś podziału element  $a[k]$  oraz przyjmując  $L = 1$  i  $P = n$  (lewy i prawy indeks). W wyniku porównania wartości  $k$  z indeksami  $i$  oraz  $j$  można określić, w której części podziału znajduje się  $k$ -ty co do wartości element. I tak, jeżeli  $j < k$ , szukany element znajduje się w prawej części podziału, a jeżeli  $k < i$ , w lewej. Ograniczamy zatem poszukiwania do jednej z tych części, korygując indeks  $L$  lub  $P$  i dzieląc ją na kolejne dwie części względem osi  $a[k]$ . Proces dzielenia powtarzamy, aż w końcu wystąpi trzeci z opisanych przypadków, w którym wartości indeksów  $L$  i  $P$  się zrównają. Oto zapis tego algorytmu w pseudojęzyku programowania:

```

procedure ZNAJDZ( $a[1..n]$ ,  $k$ )
   $L := 1$ 
   $P := n$ 
  while  $L < P$  do
     $i := L$ 
     $j := P$ 
     $os := a[k].key$ 
    repeat
      while  $a[i].key < os$  do  $i := i+1$ 
      while  $os < a[j].key$  do  $j := j-1$ 
      if  $i \leq j$  then
         $tmp := a[i]$ 
         $a[i] := a[j]$ 
         $a[j] := tmp$ 
         $i := i+1$ 
         $j := j-1$ 
    until  $i > j$ 
    if  $j < k$  then  $L := j+1$ 
    if  $k < i$  then  $P := i-1$ 

```

W implementacji powyższego algorytmu w języku Object Pascal w przypadku ciągu liczb całkowitych reprezentowanego przez tablicę otwartą należy uwzględnić zakres indeksów od 0 do  $n - 1$  również przy określaniu parametru  $k$ . Implementacja ta może wyglądać następująco:

```

procedure Znajdz( $var a: array of integer$ ;  $k: integer$ );
var
   $L, P, i, j, tmp, os: integer$ ;
begin
   $L := 0$ ;
   $P := High(a)$ ;
  while  $L < P$  do
    begin
       $i := L$ ;
       $j := P$ ;

```

```

os := a[k];
repeat
  while a[i] < os do Inc(i);
  while os < a[j] do Dec(j);
  if i <= j then
    begin
      tmp := a[i];
      a[i] := a[j];
      a[j] := tmp;
      Inc(i);
      Dec(j);
    end;
until i > j;
if j < k then L := j+1;
if k < i then P := i-1;
end;
end;

```

Podobnie jak sortowanie szybkie, algorytm Hoare’a należy do najszybszych. Dowodzi się [3], że jego złożoność oczekiwana wynosi:

$$A(n) \approx 4n = O(n),$$

choć w przypadku pesymistycznym algorytm działa w czasie  $O(n^2)$ . Wypada wspomnieć, że istnieją algorytmy znajdowania  $k$ -tej statystyki pozycyjnej, które wykonują się w czasie liniowym w każdym przypadku [1, 2, 3, 7, 17].

## 2.8. Algorytmy sortowania zewnętrznego

Specyfika dostępu do danych zawartych w plikach sekwencyjnych przechowywanych w pamięci masowej (dyski i taśmy magnetyczne<sup>3</sup>) jest zupełnie inna niż w przypadku pamięci wewnętrznej. Ponieważ w danej chwili dostępny jest tylko jeden rekord pliku<sup>4</sup>, trzeba stosować inne metody niż w przypadku sortowania ciągów implementowanych za pomocą tablic.

Ogólną cechą **sortowania zewnętrznego**, zwanego również **sortowaniem sekwencyjnym**, jest tworzenie uporządkowanych **serii** (sekwensów) elementów i łączenie ich (scalanie) w dłuższe serie uporządkowane. W kolejnych przebiegach sortowania rozmiary serii wzrastają, a ich liczba maleje, aż w końcu powstanie

<sup>3</sup> Taśmy magnetyczne przeszły już do historii, ale metody sortowania przechowywanych na nich danych idealnie pasują do danych pamiętanych na dyskach i innych podobnych nośnikach.

<sup>4</sup> Do pamięci wewnętrznej można wprowadzić większą liczbę rekordów, tzw. blok.

jedna seria stanowiąca posortowany plik. Za operacje dominujące przyjmuje się zazwyczaj przesyłania rekordów (niekiedy bloków) między pamięcią zewnętrzną a wewnętrzną i ewentualnie porównywania kluczy.

W algorytmach sortowania zewnętrznego wyróżnia się dwie zasadnicze fazy:

- 1) **rozdzielanie** – tworzenie serii początkowych i rozdzielanie ich do dwóch lub większej liczby plików,
- 2) **scalanie** – łączenie serii z dwóch lub więcej plików w dłuższe serie i zapis połączonych serii do jednego lub większej liczby plików.

**Scalanie (łączenie) proste z 4 plikami.** Sortowanie rozpoczynamy od rozdzielania  $n$  rekordów pliku wejściowego na pliki  $F_1$  i  $F_2$ , zapisując w nich serie jednoelementowe. Dalej proces polega na odczytywaniu po jednej serii z plików  $F_1$  i  $F_2$ , łączeniu obu serii w serię dwukrotnie dłuższą i zapisywaniu tak scalonych serii na przemian do plików  $F_3$  i  $F_4$ . Po wyczerpaniu plików  $F_1$  i  $F_2$  wykorzystujemy je do zapisu serii powstałych w wyniku łączenia serii z plików  $F_3$  i  $F_4$  itd. Tak więc, w kolejnych fazach scalania role plików  $F_1$  i  $F_2$  oraz  $F_3$  i  $F_4$  się odwracają, a powstające serie wyjściowe są dwukrotnie dłuższe od serii wejściowych. Na początku serie mają długość 1, a w każdej fazie scalania są podwajane. Po  $k$  fazach scalania długości serii wynoszą  $2^k$ . Dla  $2^k \geq n$  jeden z dwóch plików wyjściowych będzie pusty, a drugi będzie zawierał tylko jedną serię o długości  $n$ , czyli oczekiwany wynik sortowania.

Algorytm scalania prostego z 4 plikami można opisać w postaci następującej listy kroków:

- K1.** Rozrzucić wstępne serie jednoelementowe z pliku źródłowego na przemian do plików  $F_1$  i  $F_2$ .
- K2.** Scal serie z plików  $F_1$  i  $F_2$ , zapisując serie wynikowe na przemian do plików  $F_3$  i  $F_4$ .
- K3.** Jeżeli powstała tylko jedna seria, zakończ wykonanie algorytmu.
- K4.** Odwróć role plików  $F_1$  i  $F_2$  z  $F_3$  i  $F_4$ , a następnie wróć do punktu K2.

Zapis algorytmu w języku programowania wymaga uwzględnienia przypadku, gdy liczba  $n$  wszystkich rekordów pliku źródłowego nie jest całkowitą potęgą dwójki. Na końcu jednego z plików roboczych może się wtedy pojawić seria o mniejszej liczbie rekordów, zwana **ogonem**. Mówiąc ściślej, powstałe w  $k$ -tej fazie scalania serie mają długości  $m = 2^k$ , a ewentualny ogon, zapisany na końcu jednego z plików wyjściowych, ma długość  $r = n \bmod m$ . Ponadto liczby serii w obu plikach wyjściowych, z uwzględnieniem ogona, mogą się różnić o jeden.

Liczba operacji przesyłania rekordów pomiędzy pamięcią zewnętrzną a wewnętrzną (odczyt i zapis rekordu) wynosi w każdym kroku  $2n$ . Stąd złożoność czasowa algorytmu, mierzona ogólną liczbą tych operacji, wynosi:

$$T(n) = 2n + 2n \log n = O(n \log n)$$

Pierwszy składnik określa liczbę operacji w kroku rozdzielania wstępnego, a drugi w powtarzanych wielokrotnie krokach scalania, których jest  $\log n$ . Oszacowanie to dotyczy zarówno złożoności pesymistycznej, jak i oczekiwanej.

Algorytm można uogólnić na przypadek, gdy ma się do dyspozycji  $2p$  plików, tj.  $p$  plików wejściowych i  $p$  plików wyjściowych ( $p \geq 2$ ). Proces sortowania składa się wówczas z szeregu przebiegów, podczas których serie z  $p$  plików wejściowych są łączone i zapisywane do  $p$  plików wyjściowych. Na końcu każdego przebiegu pliki wejściowe są zamieniane z plikami wyjściowymi. Złożoność drugiego kroku sortowania jest mniejsza dla większej od 4 liczby plików [3, 17].

**Scalanie (łączenie) proste z 3 plikami.** Liczbę plików używanych w sortowaniu przez scalanie proste można zmniejszyć do 3. Każdy przebieg składa się wtedy z fazy rozdzielania, w której serie jednego pliku są rozdzielane równomiernie do dwóch pozostałych plików, i fazy scalania, w której serie z dwóch plików są łączone i zapisywane do trzeciego. Ogólnie, alternatywą dla sortowania prostego z  $2p$  plikami może być odmiana wykorzystująca  $p + 1$  plików ( $p \geq 2$ ). W każdej fazie scalania serie wejściowe są wczytywane z  $p$  plików, a serie wynikowe są zapisywane w pozostałym,  $p + 1$  pliku. Proces sortowania składa się z szeregu takich przebiegów, podczas których po fazie rozdzielania jest wykonywana faza scalania. Jego pesymistyczna złożoność czasowa również wynosi  $O(n \log n)$ , ale ma nieco gorszy współczynnik przy  $n \log n$ .

Algorytm scalania prostego z 3 plikami można opisać następująco:

- K1.** Rozrzuć wstępne serie jednoelementowe z pliku źródłowego na przemian do plików  $F_1$  i  $F_2$ .
- K2.** Scal serie z plików  $F_1$  i  $F_2$ , zapisując serie wynikowe do pliku  $F_3$ .
- K3.** Jeżeli powstała tylko jedna seria, zakończ wykonanie algorytmu.
- K4.** Rozrzuć serie z pliku  $F_3$  na przemian do plików  $F_1$  i  $F_2$ , a następnie wróć do punktu K2.

Algorytm ten można udoskonalić, unikając połowy operacji kopiowania, co prowadzi do skrócenia czasu sortowania o około 25% [17]:

- K1.** Rozrzuć wstępne serie jednoelementowe z pliku źródłowego na przemian do plików  $F_1$  i  $F_2$ .
- K2.** Scal serie z plików  $F_1$  i  $F_2$ , zapisując serie wynikowe do pliku  $F_3$ .
- K3.** Jeżeli plik  $F_3$  zawiera tylko jedną serię, zakończ wykonanie algorytmu.
- K4.** Skopiuj połowę serii z pliku  $F_3$  do pliku  $F_1$ .
- K5.** Scal serie z pliku  $F_1$  z pozostałymi seriami z pliku  $F_3$ , zapisując serie wynikowe do pliku  $F_2$ .
- K6.** Jeżeli plik  $F_2$  zawiera tylko jedną serię, zakończ wykonanie algorytmu.
- K7.** Odwróć role pliku  $F_2$  z  $F_3$  i wróć do punktu K4.

**Scalanie (łączenie) naturalne.** W algorytmie sortowania przez scalanie proste nie korzysta się z być może istniejącego częściowego uporządkowania rekordów. Długości wszystkich serii łączonych w przebiegu  $k$ , z wyjątkiem ewentualnego ogona, wynoszą  $2^k$ , chociaż mogą pojawiać się dłuższe uporządkowane ciągi rekordów. Ba, nawet wykonuje się tę samą liczbę operacji czytania i zapisywania rekordów niezależnie od tego, czy plik źródłowy jest uporządkowany czy nie. Oczywistym usprawnieniem algorytmu jest więc uwzględnienie już istniejącego uporządkowania pliku [17, 28].

W sortowaniu przez łączenie, w którym operacje scalania są wykonywane na możliwie najdłuższych seriach, liczba serii w kolejnych przebiegach zmniejsza się co najmniej dwukrotnie. W przypadku pesymistycznym złożoność czasowa algorytmu wynosi  $O(n \log n)$ , ale przeciętnie jest lepsza od scalania prostego.

**Sortowanie (łączenie) wielofazowe z 3 plikami.** W sortowaniu przez łączenie proste można całkowicie wyeliminować kopiowanie (rozdzielanie) serii rekordów występujące po fazach scalania. Po wstępnej fazie rozdzielania, w której są tworzone serie początkowe i zapisywane do  $p$  plików, następują fazy scalania, w których serie czytane z  $p$  plików są łączone i odsyłane do  $p + 1$  pliku. Wybór pliku wyjściowego, do którego są przesyłane scalone serie, dokonywany jest jednak według innej zasady: gdy jeden z plików wejściowych się wyczerpie, staje się plikiem wyjściowym dla operacji scalania niewyczerpanych jeszcze plików wejściowych i pliku, który uprzednio był wyjściowym. Rezygnuje się tym samym z przebiegów, podczas których wszystkie rekordy pliku źródłowego są czytane i rozrzucone do pozostałych plików. Metoda ta, wynaleziona przez R.L. Gilstada w 1960 roku, nosi nazwę **sortowania wielofazowego** lub **polifazowego**.

Jeżeli  $p = 2$ , tj. gdy mamy do dyspozycji 3 pliki, rozpoczynamy od dwóch plików  $F_1$  i  $F_2$  zawierających serie o długości 1. Z rekordów otrzymywanych z plików  $F_1$  i  $F_2$  tworzymy serie o długości 2 i zapisujemy je do pliku  $F_3$ . Proces ten kontynuujemy, aż do wyczerpania jednego z plików wejściowych. Załóżmy, że będzie to

plik  $F_1$ . Wówczas, w drugiej już fazie scalania, łączymy pozostałe jeszcze w pliku  $F_2$  serie o długości 1 z seriami o długości 2 znajdującymi się w pliku  $F_3$ . Wynikiem tej fazy scalania są serie o długości 3 zapisywane do pliku  $F_1$ . Po wyczerpaniu pliku  $F_2$  łączymy niewykorzystane jeszcze serie o długości 2 z pliku  $F_3$  z seriami o długości 3 z pliku  $F_1$ . Podczas tej fazy powstają serie o długości 5, które są odsyłane do pliku  $F_2$ . Całe postępowanie kończymy, gdy powstanie jeden plik z jedną serią, a pozostałe dwa pliki będą puste.

Długości serii w kolejnych fazach sortowania wynoszą: 1, 1, 2, 3, 5, 8, 13, ... Liczby te tworzą omawiany w podrozdziale 1.8 ciąg Fibonacciego:

$$f_k = \begin{cases} 0 & (k = 0) \\ 1 & (k = 1) \\ f_{k-1} + f_{k-2} & (k \geq 2) \end{cases}$$

Przykład sortowania wielofazowego 55 rekordów z wykorzystaniem 3 plików roboczych przedstawia tabela 2.6. Przed nawiasami podane są liczby serii w pliku, a w nawiasach długości tych serii. Myślnik oznacza plik pusty.

**Tabela 2.6.** Przykład sortowania wielofazowego z 3 plikami

Po kroku	Plik $F_1$	Plik $F_2$	Plik $F_3$
rozd.	21(1)	34(1)	–
1	–	13(1)	21(2)
2	13(3)	–	8(2)
3	5(3)	8(5)	–
4	–	3(5)	5(8)
5	3(13)	–	2(8)
6	1(13)	2(21)	–
7	–	1(21)	1(34)
8	1(55)	–	–

Aby opisany proces mógł funkcjonować, początkowe liczebności plików  $F_1$  i  $F_2$  muszą być starannie dobrane, tj. powinny być dwiema kolejnymi liczbami Fibonacciego. Taki układ jest możliwy do zrealizowania, gdy liczba rekordów pliku źródłowego jest liczbą Fibonacciego. Należy przypuszczać, że gdy warunek ten nie jest spełniony, można temu jakoś zaradzić. Istotnie, sztuczka polega na zasymulowaniu istnienia fikcyjnych, pustych serii tak, by suma serii rzeczywistych i fikcyjnych była liczbą Fibonacciego. Oczywiście, rodzi to kolejny problem, którym jest rozpoznawanie serii fikcyjnych podczas fazy scalania. Rozwiązuje się go

w ten sposób, że odczyt serii fikcyjnej nie powoduje faktycznego czytania pliku. Oznacza to, że plik taki nie bierze udziału w łączeniu serii, a jeśli drugi plik wejściowy zawiera serię rzeczywistą, to jest ona przepisywana do pliku wyjściowego jako połączona z serią fikcyjną<sup>5</sup>.

Algorytm scalania wielofazowego z 3 plikami można opisać następująco:

- K1.** Rozrzuć wstępne serie jednoelementowe pliku źródłowego do plików  $F_1$  i  $F_2$  tak, aby liczebności plików  $F_1$  i  $F_2$  były kolejnymi liczbami Fibonacciego.
- K2.** Jeżeli powstała tylko jedna seria, zakończ wykonanie algorytmu.
- K3.** Scalaj serie z plików  $F_1$  i  $F_2$ , zapisując serie wynikowe do pliku  $F_3$ , aż do wyczerpania pliku  $F_1$  lub  $F_2$ .
- K4.** Jeżeli oba pliki wejściowe zostały wyczerpane, zakończ wykonanie algorytmu.
- K5.** Odwróć role wyczerpanego pliku wejściowego ( $F_1$  lub  $F_2$ ) z plikiem  $F_3$  i wróć do punktu K3.

Zauważmy, że spośród wszystkich faz scalania tylko w ostatniej są czytane i zapisywane wszystkie rekordy pliku źródłowego. Każda z pozostałych dotyczy tylko pewnego jego podzbioru. Oczywiście, początkowy krok rozdzielania pliku źródłowego na dwa pliki  $F_1$  i  $F_2$  wymaga przetworzenia całego pliku źródłowego. Precyzowanie tego przebiegu wymaga szczególnej uwagi, gdyż jest on bardziej złożony niż w przypadku sortowania prostego. Trzeba bowiem nieznana na początku liczbę  $n$  rekordów pliku źródłowego<sup>6</sup>, po ewentualnym uzupełnieniu o liczbę  $d$  serii fikcyjnych, co wymaga dość specyficznych technik [17, 28], przedstawić w postaci sumy dwóch kolejnych liczb Fibonacciego:

$$n + d = f_k + f_{k-1}.$$

Korzystając z własności liczb Fibonacciego, można udowodnić, że złożoność scalania wielofazowego z 3 plikami, wyrażona jako liczba przesłań rekordów między pamięcią zewnętrzną a wewnętrzną, wynosi  $2,08 \cdot n \log n + O(n)$  [17]. Scalanie wielofazowe z 3 plikami jest więc nieco gorsze od scalania prostego z 4 plikami, ale wymaga mniejszej liczby plików i jest bardziej interesujące.

<sup>5</sup> Długość takiej serii jest mniejsza od długości normalnych serii powstałych w tej fazie scalania.

<sup>6</sup> Nie jest to do końca prawdą, ponieważ w niektórych systemach można się dowiedzieć, jaka jest liczba rekordów pliku. Na przykład w Turbo Pascalu i Object Pascalu umożliwia to funkcja *FileSize*.



W rozpatrywanych algorytmach sortowania zewnętrznego, oprócz scalania naturalnego, właściwe sortowanie rozpoczyna się od serii o długości 1 tworzonych we wstępnym przebiegu rozdzielania pliku źródłowego. Proces sortowania można znacznie przyspieszyć, wykorzystując sortowanie wewnętrzne do tworzenia dłuższych serii początkowych. Modyfikacja wstępnego przebiegu rozdzielania polega na wczytywaniu do pamięci po  $m$  rekordów (dla pewnego ustalonego  $m$ ), sortowaniu ich i zapisywaniu jako serie o długości  $m$ . Jeżeli np. plik źródłowy zawiera 10 mln rekordów, to sortowanie ich metodą scalania prostego z 4 plikami wymaga w przypadku serii początkowych o długości 1 wykonania  $\log 10^7 \approx 23$  przebiegów, podczas gdy w przypadku serii o długości 1000, których jest 10000, tylko  $\log 10^4 \approx 12$  przebiegów.

Algorytmy sortowania zewnętrznego przez scalanie można przystosować do sortowania tablic. Prowadzi to do grupy algorytmów sortowania wewnętrznego o złożoności czasowej pesymistycznej i oczekiwanej rzędu  $n \log n$ . Wymagają one jednak pewnego narzutu pamięci, przez co ich złożoność pamięciowa jest większa niż w przypadku typowych algorytmów sortowania wewnętrznego.

Omawiając algorytmy sortowania zewnętrznego, nie zapisywaliśmy ich ani w pseudojęzyku, ani w konkretnym języku programowania ze względu na duże rozmiary kodu źródłowego. Zainteresowany Czytelnik może kod tych algorytmów w języku Pascal znaleźć w książce [28]. Wiele dodatkowych wiadomości na temat sortowania wewnętrznego jest zawartych w książce [17].

## Ćwiczenia

**2.1.** Napisz w pseudojęzyku programowania algorytm odwracania kolejności elementów ciągu i wyznacz jego złożoność obliczeniową. Opracuj implementację komputerową tego algorytmu.

**2.2.** Napisz w pseudojęzyku programowania algorytm łączenia dwóch ciągów uporządkowanych niemalejąco w jeden ciąg również uporządkowany niemalejąco. Wyznacz złożoność obliczeniową tego algorytmu. Opracuj dwie jego implementacje komputerowe, pierwszą dla ciągów przechowywanych w pamięci wewnętrznej, drugą dla plików tekstowych lub binarnych.

**2.3.** Opracuj implementację komputerową wersji algorytmu sortowania przez wstawianie z wykorzystaniem wartownika oraz program, który mierzy i porównuje czasy wykonania wersji bez wartownika i z wartownikiem.

**2.4.** Opracuj program sortowania w pamięci wewnętrznej krótkiego pliku binarnego złożonego z rekordów o polach: *imię*, *nazwisko*, *pleć*, *rok studiów*, *numer grupy*, *numer indeksu* i wyniki z egzaminów z trzech przedmiotów. Wykorzystaj algorytm sortowania przez selekcję lub przez wstawianie. Program powinien umożliwić użytkownikowi wybór klucza:

*nazwisko + imię*,

*średnia ocena + nazwisko + imię*.

**2.5.** Algorytm sortowania nazywamy **stabilnym**, gdy zachowuje początkowe ustawienie względem siebie elementów o takich samych kluczach. W przypadku złożonego klucza sortowanie stabilne umożliwia uporządkowanie ciągu w kilku procesach, oddzielnych dla każdej części klucza<sup>7</sup>. Na przykład plik z ćwiczenia 2.4 można uporządkować według drugiego klucza, sortując go najpierw według pola *imię*, potem według pola *nazwisko*, a na końcu według wyliczonej średniej oceny. Które z omówionych w niniejszym rozdziale algorytmów są stabilne?

**2.6.** Dokonaj analizy złożoności czasowej algorytmu sortowania przez wstawianie binarne, wykorzystując wzór Stirlinga.

**2.7.** Korzystając z równania rekurencyjnego (2.4), udowodnij przez indukcję matematyczną, że w przypadku optymistycznym złożoność czasowa algorytmu sortowania szybkiego dla dowolnych  $n$  wynosi  $O(n \log n)$ .

**2.8.** Wyjaśnij działanie algorytmu sortowania szybkiego, w którym rolę osi podziału pełni pierwszy element ciągu. Rozważ przypadki ciągu stałego, uporządkowanego i uporządkowanego odwrotnie.

**2.9.** Zmodyfikuj algorytm sortowania szybkiego tak, aby oś była medianą trzech elementów ciągu. Opracuj implementację komputerową tej wersji i porównaj jej czas wykonania z wersją omówioną w tym rozdziale.

**2.10.** Znajdź sposób podziału ciągu na trzy grupy: elementów mniejszych od osi, równych jej i większych od niej. Opracuj wersję algorytmu sortowania szybkiego wykorzystującą taki podział.

---

<sup>7</sup> Metoda ta była stosowana w sorterach – maszynach do sortowania kart dziurkowanych. Sortowanie pliku kart według pola złożonego z kilku kolumn było realizowane oddzielnie dla każdej kolumny, począwszy od ostatniej, a skończywszy na pierwszej.

**2.11.** Rozbuduj aplikację sortowania bąbelkowego z podrozdziału 1.3 tak, aby umożliwiała wybór dowolnego algorytmu omówionego w niniejszym rozdziale.

**2.12.** Zilustruj na rysunku przedstawiającym drzewo binarne przebieg procesu budowy i sortowania kopca pokazanego w tabelach 2.4 i 2.5.

**2.13.** Opracuj algorytm, który sprawdza, czy tablica o elementach całkowitych reprezentuje kopiec zupełny.

**2.14.** Opracuj algorytm, który sortuje w czasie liniowym ciąg  $n$  liczb całkowitych o wartościach z zakresu od 1 do  $m$ . Jaka jest dokładnie złożoność czasowa tego algorytmu?

**Wskazówka.** Użyj pomocniczej tablicy  $m$ -elementowej do zliczania liczby wystąpień poszczególnych wartości w ciągu.

**2.15.** Opracuj algorytm, który znajduje najmniejszy i największy element ciągu, po czym wymienia je z elementami znajdującymi się na pierwszej i ostatniej pozycji, a po każdym takim kroku redukuje ciąg o dwa elementy znajdujące się na właściwych pozycjach i powtarza postępowanie dla pozostałej części ciągu, aż w końcu cały zostanie uporządkowany. Dokonaj analizy złożoności czasowej tego algorytmu sortowania.

**2.16.** Napisz program, który dla danej liczby całkowitej  $n$  takiej, że  $1 \leq n \leq 12$ , generuje wszystkie kombinacje zbioru  $\{1, 2, \dots, n\}$ : najpierw 1-elementowe, potem 2-elementowe itd., a w końcu  $n$ -elementowe. Każda z tych grup ma być uporządkowana leksykograficznie. Na przykład dla  $n = 3$  poprawnym wynikiem programu są ciągi liczb:

```
1
2
3
1 2
1 3
2 3
1 2 3
```

**Wskazówka.** Dowolna kombinacja zbioru  $\{1, 2, \dots, n\}$  może być zapisana jako wartość typu *word*, w której bit o wartości 1 oznacza, że określony element zbioru należy do kombinacji, a bit o wartości 0, że nie. Zbiór wszystkich kombinacji może więc być reprezentowany przez tablicę o elementach typu *word*.

**2.17.** Usprawnienie algorytmu *insertion sort*, które zaproponował D.L. Shell w 1959 roku, polega na posortowaniu wszystkich rekordów oddalonych od siebie o stały przyrost, następnie posortowaniu wszystkich rekordów oddalonych od siebie o mniejszy przyrost itd. Na końcu sortowane są wszystkie rekordy oddalone o przyrost 1, jak w prostym algorytmie sortowania przez wstawianie. Jako ciąg przyrostów można wziąć dowolny ciąg liczb całkowitych taki, że:

$$k_m > k_{m-1} > \dots > k_2 > k_1 = 1.$$

Istnieją różne propozycje wyboru ciągu przyrostów [17, 28]. Niektóre pozwalają zredukować czas sortowania do  $O(n^{1.5})$ , a nawet do  $O(n^{1.2})$ .

Oto jedna z możliwych wersji algorytmu Shella, nazywanego **sortowaniem za pomocą malejących przyrostów**:

```

procedure SHELL-SORT(a[1..n])
  k := n div 2
  while k > 0 do
    for i := k+1 to n do
      tmp := a[i]
      j := i-k
      while (j > 0) and (tmp.key < a[j].key) do
        a[j+k] := a[j]
        j := j-k
      a[j+k] := tmp
    k := k div 2

```

Zilustruj na przykładzie tablicy liczb całkowitych działanie tego algorytmu. Opracuj jego implementację w Object Pascalu i porównaj sprawność z innymi algorytmami omówionymi w niniejszym rozdziale.

**2.18.** Opracuj w Delphi uniwersalną implementację algorytmu znajdowania statystyki pozycyjnej dla liczb całkowitych, rzeczywistych i łańcuchów.

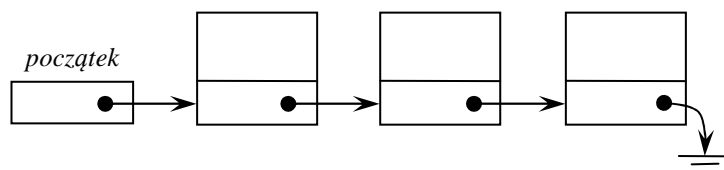
## Rozdział 3.

### Listowe struktury danych

W każdym języku programowania istnieją elementarne typy danych. Na przykład w języku Pascal są nimi m.in.: *integer*, *real*, *boolean*, *char*, a w językach C i C++: *int*, *long*, *float*, *double*, *char*. Typy elementarne służą jako elementy bazowe do tworzenia najprostszych struktur danych. W Pascalu są nimi tablice i rekordy, a w C i C++ tablice, struktury i unie<sup>1</sup>. Zmienne typów elementarnych i strukturalnych zajmują obszary pamięci o stałym rozmiarze wynikającym ze sposobu zapisu ich wartości w pamięci komputera<sup>2</sup>.

Istnieje wiele zagadnień, których naturalne rozwiązanie wiąże się z wykorzystaniem danych o zmiennym rozmiarze, zwanych **dynamicznymi strukturami danych**, którym pamięć jest przydzielana dynamicznie w trakcie wykonywania programu. Struktura dynamiczna składa się z elementów (rekordów w Pascalu i struktur lub unii w C i C++) zawierających pewne dane i jeden lub większą liczbę wskaźników (dowiązań) do innych jej elementów.

Najprostszą strukturą dynamiczną jest lista jednokierunkowa z pojedynczymi dowiązaniem (rys. 3.1). Każdy element tej struktury zawiera, oprócz właściwych danych, wskaźnik do następnego elementu. W ostatnim elemencie jest nim wskaźnik pusty (**nil** w Pascalu, **NULL** w C i C++), który oznacza, że element ten nie ma następnika. Pomiedzy elementami listy jednokierunkowej można przechodzić tylko w jednym kierunku, od jej początku do końca, wykorzystując wskaźnik do następnego elementu. Rozmiar listy może się zmieniać, elementy mogą być tworzone i wstawiane do listy, mogą też być z niej usuwane [1, 8, 11, 13, 28, 30].

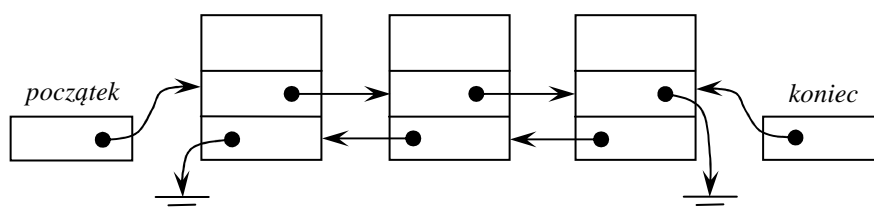


Rys. 3.1. Lista jednokierunkowa

<sup>1</sup> We wczesnych językach programowania, np. takich jak Fortran i Algol, tablica jest jedynym strukturalnym typem danych.

<sup>2</sup> W Delphi 4 wprowadzono tablice dynamiczne, których rozmiar można w dowolnej chwili zmienić za pomocą procedury *SetLength*. Z tego udogodnienia korzystaliśmy w poprzednich rozdziałach.

Bardziej skomplikowane struktury dynamiczne, takie jak listy dwukierunkowe, struktury drzewiaste itp., posiadają większe możliwości zarządzania danymi. Na przykład lista dwukierunkowa, której elementy zawierają po dwa dowiązania, do następnego i poprzedniego elementu, umożliwia przechodzenie w dwóch kierunkach, od początku do końca i od końca do początku (rys. 3.2). Pełne potraktowanie tematyki przetwarzania list i struktur drzewiastych w języku Pascal można znaleźć w znakomitej książce Niklausa Wirtha [28].



Rys. 3.2. Lista dwukierunkowa

W kolejnych podrozdziałach niniejszego rozdziału zajmiemy się najważniejszymi abstrakcyjnymi strukturami danych, takimi jak listy, stosy, kolejki, zbiory, słowniki, grafy i drzewa. Abstrakcyjność tych struktur przejawia się w tym, że stanowią one model danych, z którym jest skojarzony zestaw podstawowych operacji określających ich zachowanie. Przedstawimy sposoby posługiwania się nimi, metody reprezentowania w pamięci komputera za pomocą tablic i struktur dynamicznych oraz implementowania w środowisku Delphi.

### 3.1. Listy, stosy, kolejki

Podstawową abstrakcyjną strukturą danych jest **lista**, która jest ciągiem pewnej liczby  $n \geq 0$  elementów określonego zbioru. Najprostszym sposobem zapisu listy jest wyszczególnienie jej kolejnych elementów:

$$x_1, x_2, \dots, x_n.$$

Wartość  $n$  nazywamy **długością** lub **rozmiarem** listy. Jeżeli  $n = 0$ , listę nazywamy **pustą**. Istotne jest, że elementy listy zajmują w niej określone pozycje, czyli że są liniowo uporządkowane. Mówimy np., że element  $x_1$  jest pierwszym elementem listy, a  $x_n$  ostatnim, a także, że element  $x_k$  poprzedza element  $x_{k+1}$  lub jest jego poprzednikiem, oraz że element  $x_k$  następuje po elemencie  $x_{k-1}$  lub jest jego następnikiem. Jest oczywiste, że element  $x_1$  nie ma poprzednika, a element  $x_n$  nie ma następnika. Elementy te nazywamy lewym i prawym **końcem** listy.

Na listach można wykonywać pewne operacje. Zbiór tych operacji nie jest ściśle określony. Trudno jest bowiem zaproponować taki ich zestaw, aby wszystkie były przydatne we wszelkich zastosowaniach, a jednocześnie były wydajne. Przykładowy zestaw może obejmować następujące operacje:

- ♦ uzyskanie dostępu do elementu listy,
- ♦ wstawienie nowego elementu do listy,
- ♦ usunięcie elementu z listy,
- ♦ wyznaczenie liczby elementów listy,
- ♦ połączenie (złożenie) dwóch lub większej liczby list w jedną,
- ♦ tworzenie kopii listy lub jej ciągłego fragmentu (podlisty),
- ♦ znajdowanie elementu listy o zadanej wartości pola,
- ♦ sortowanie listy względem określonego pola.

Nie są to bynajmniej wszystkie możliwe operacje na listach. Zazwyczaj nie korzysta się ze wszystkich wymienionych operacji, ale z pewnego ich podzbioru. Za pomocą zestawu kilku podstawowych operacji na listach można definiować inne operacje, dbając przy tym o ich wydajność, która zależy od reprezentacji listy w pamięci komputera.

Do podstawowych operacji na listach zalicza się przede wszystkim operacje dotyczące lewego i prawego końca listy [3, 7]:

- 1) *push*( $q, x$ ) – wstawienie elementu  $x$  na lewy koniec listy  $q$ ,
- 2) *pop*( $q$ ) – usunięcie bieżącego lewego końca listy  $q$ ,
- 3) *front*( $q$ ) – pobieranie (odczyt) lewego końca listy  $q$ ,
- 4) *inject*( $q, x$ ) – wstawienie elementu  $x$  na prawy koniec listy  $q$ ,
- 5) *eject*( $q$ ) – usunięcie bieżącego prawego końca listy  $q$ ,
- 6) *rear*( $q$ ) – pobieranie (odczyt) prawego końca listy  $q$ .

Operacje te, oprócz *push* i *pop*, bywają nazywane inaczej. Na przykład zamiast *front* można często spotkać nazwę *top* a czasem *first*, a zamiast *rear* i *inject* nazwy *last* i *add*. Niekiedy operacja *eject* nazywana jest *delete*, chociaż zazwyczaj pod tą nazwą kryje się operacja usuwania dowolnego elementu listy.

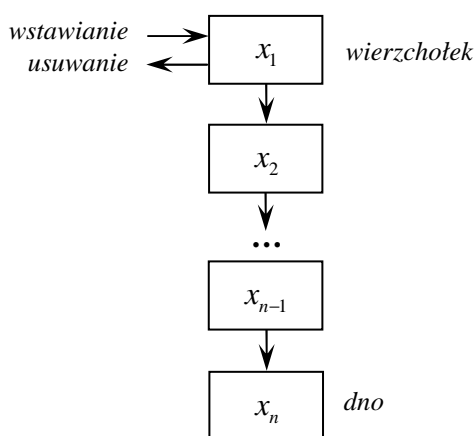
Zależnie od tego, które z wymienionych podstawowych operacji można wykonać na liście, rozróżnia się trzy zasadnicze rodzaje list:

- 1) **stos** – wstawianie, usuwanie i odczyt lewego końca listy (*push*, *pop*, *front*),
- 2) **kolejka** (pojedyncza, jednokierunkowa) – wstawianie prawego, a usuwanie i odczyt lewego końca listy (*inject*, *pop*, *front*),
- 3) **kolejka podwójna** (dwustronna, dwukierunkowa) – wszystkie operacje dotyczą lewego i prawego końca listy.

Szeroko spotykane są inne określenia stosu (ang. *stack*) i kolejki (ang. *queue*), doskonale odzwierciedlające zasadę działania obu struktur. I tak, stos jest często określany mianem **LIFO** (ang. *last in first out* – kto ostatni wchodzi, ten pierwszy wychodzi), a kolejka – **FIFO** (ang. *first in first out* – kto pierwszy wchodzi, ten pierwszy wychodzi).

Sytuacja przedstawia się tak, jak np. z ludźmi w kościele. Stos przypomina kościół wypełniony ludźmi: osoba, która ostatnia weszła do kościoła, pierwsza z niego wychodzi. Podobnie jest z kolejką: osoba, która pierwsza stanęła w kolejce do spowiedzi, pierwsza tę kolejkę opuszcza. Wprawdzie rzeczywistość może być inna, bo ktoś może przecisnąć się przez tłum, albo wejść do kolejki z boku, albo nawet z niej uciec, ale tu kończy się informatyczna teoria stosów i kolejek, a zaczynają mniej istotne problemy osoby wierzącej.

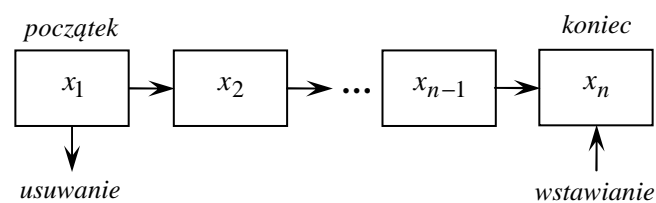
Mówiąc o stosie, często używamy specyficznej terminologii. Mówimy mianowicie, że element **wstawiamy** na **wierzchołek** lub **szczyt** stosu (ang. *top*), bądź go stamtąd **zdejmujemy**, a o elemencie, którego nie daje się zdjąć, dopóki nie zdejmie się wszystkich innych elementów, mówimy, że leży na **dnie** stosu. Operacje wstawiania i usuwania elementu dotyczą zawsze wierzchołka stosu (por. rys. 3.3).



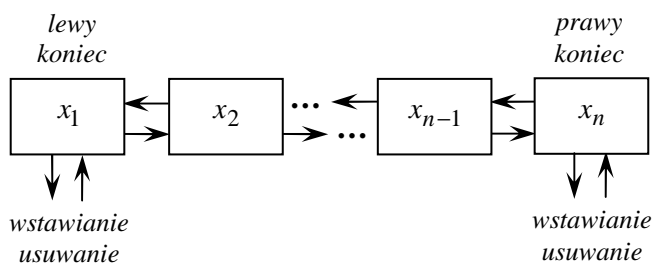
Rys. 3.3. Organizacja stosu

Inaczej jest w przypadku kolejki, kiedy to wygodnie jest posługiwać się terminami **początek** i **koniec** lub **głowa** i **ogon** kolejki, a w przypadku kolejki podwójnej wygodnie jest mówić o jej **lewym** i **prawym końcu**. Elementy wstawiamy na koniec kolejki, a usuwamy te, które znajdują się na jej początku (por. rys. 3.4a). Oczywiście, w przypadku kolejki podwójnej operacje wstawiania i usuwania elementów dotyczą jej dowolnego końca (por. rys. 3.4b).





a) kolejka pojedyncza



b) kolejka podwójna

**Rys. 3.4.** Organizacja kolejki

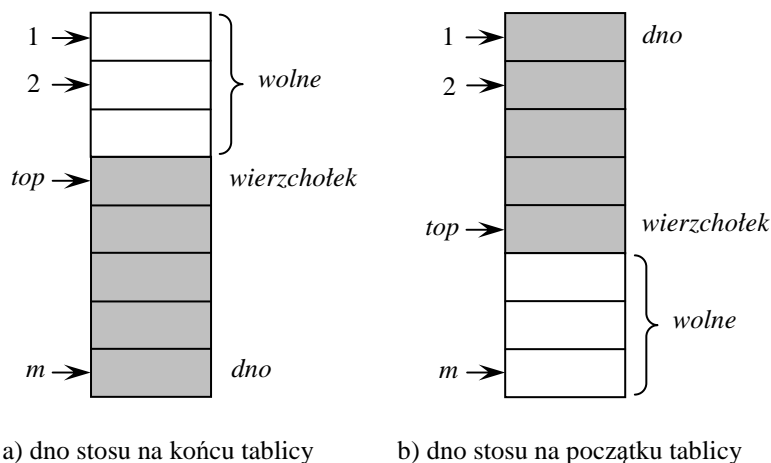
Istnieją dwie podstawowe metody reprezentowania (implementowania) listy w pamięci komputera: tablicowa i dowiązaniowa. W ramach każdej z nich istnieją różne warianty, które prowadzą do mniej lub bardziej efektywnego wykonywania wymienionych wyżej operacji na listach.

## 3.2. Implementacja tablicowa listy

W reprezentacji tablicowej elementy listy są kolejnymi elementami tablicy. Pozycje poszczególnych elementów na liście są utożsamiane z indeksami tych elementów w tablicy. Liczba elementów tablicy powinna być odpowiednio duża, aby tablica wystarczyła do zapamiętania najdłuższej spodziewanej listy. Większość operacji na listach daje się łatwo implementować. Niektóre operacje mogą być czasochłonne, np. wstawienie elementu do listy lub usunięcie go może wymagać przesunięcia innych elementów tablicy w kierunku jej początku lub końca.

Dodawanie i usuwanie elementów dokonywane jest najczęściej na jednym z końców listy, dlatego warto te operacje zaprogramować tak, aby uniknąć przesuwania elementów tablicy. Na przykład dno stosu można przycumować do końca tablicy tak, że zawsze zajmuje ono jej ostatni element, a położenie wierzchołka przesuwają się w kierunku początku tablicy w miarę odkładania elementów na stos,

bądź w kierunku jej końca w miarę zdejmowania ich ze stosu (rys. 3.5a). Równie dobrze można zakotwiczyć dno stosu na początku tablicy i przesuwając wierzchołek w odwrotnym kierunku (rys. 3.5b).



**Rys. 3.5.** Stos w implementacji tablicowej

Wszystkie podstawowe operacje na stosie reprezentowanym przez tablicę jest łatwo zaimplementować. Jako przykład rozważmy stos składający się z elementów  $S[top..m]$  tablicy  $S[1..m]$ . Jej elementy wskazywane przez indeksy  $top$  i  $m$  stanowią wierzchołek i dno stosu (por. rys. 3.5a). Jeśli  $top = m + 1$ , stos jest pusty, a jeśli  $top = 1$ , stos jest wypełniony. Początkowo  $top = m + 1$ . Operacje *push* i *pop* można w pseudokodzie programowania zapisać następująco:

```

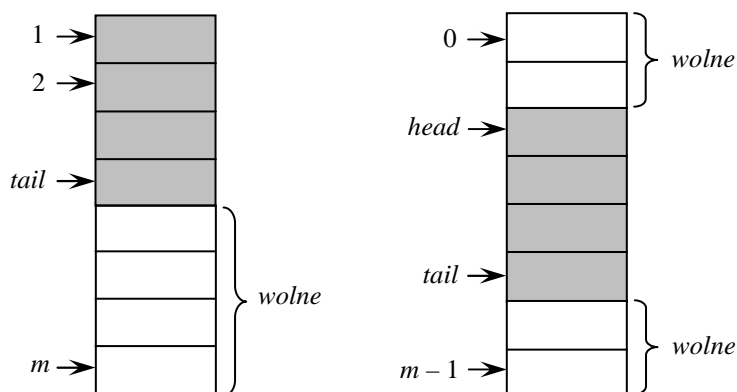
procedure PUSH( $S, x$ )
  if  $top > 1$  then
     $top := top - 1$ 
     $S[top] := x$ 
  else
    error('Stos wypełniony')

function POP( $S$ )
  if  $top \leq m$  then
     $top := top + 1$ 
    return  $S[top - 1]$ 
  else
    error('Stos pusty')

```

Obie operacje mają stałą złożoność czasową  $O(1)$ . Próby wstawienia elementu na wypełniony stos i zdjęcia elementu z pustego stosu, kończą się niepowodzeniem.

Reprezentacja kolejki, w której elementy są dodawane na końcu, a usuwane na początku, jest nieco bardziej skomplikowana. W prostszym rozwiązaniu początek kolejki jest przytwierdzony do pierwszego elementu tablicy, a koniec jest przesuwany (por. rys. 3.6a). Inne rozwiązanie polega na utrzymywaniu dwóch indeksów, które określają zmienny początek i koniec kolejki (por. rys. 3.6b).



a) stały początek, przesuwany koniec      b) przesuwany początek i koniec

**Rys. 3.6.** Kolejka w implementacji tablicowej

W przypadku, gdy początek kolejki jest zawsze pierwszym elementem tablicy  $Q[1..m]$ , a koniec przesuwa się w miarę zmieniania się jej rozmiaru, kolejka składa się z elementów  $Q[1..tail]$  (rys. 3.6a). Początkowo kolejka jest pusta, tj.  $tail = 0$ . Operacje dokładania elementu na koniec kolejki i usuwania elementu z początku kolejki można zaprogramować następująco:

```

procedure INJECT(Q, x)
  if tail < m then
    tail := tail+1
    Q[tail] := x
  else
    error('Kolejka zapełniona')

function POP(Q)
  if tail > 0 then
    x = Q[1]
    tail := tail-1
    for i:=1 to tail do
      Q[i] := Q[i+1]
    return x
  else
    error('Kolejka pusta')

```

Jak widać, operacja *inject* działa w czasie  $O(1)$ , a *pop* w czasie  $O(n)$ , gdzie  $n$  jest rozmiarem kolejki. Gorsza złożoność czasowa drugiej operacji jest konsekwencją przesuwania wszystkich elementów kolejki o jedną pozycję w kierunku początku po usunięciu pierwszego elementu.

Stałą złożoność czasową operacji *pop* możemy uzyskać, ustanawiając zmienny początek kolejki. Jeśli przy okazji zmienimy indeksację elementów tablicy od 0 do  $m - 1$  zamiast od 1 do  $m$ , otrzymamy wygodną implementację cykliczną kolejki za pomocą tablicy  $Q[0..m - 1]$ , w której następnikiem elementu  $Q[m - 1]$  jest element  $Q[0]$ . Ogólnie, następnikiem elementu o indeksie  $i \in \{0, 1, \dots, m - 1\}$  jest element o indeksie  $(i + 1) \bmod m$ . Początek kolejki wskazuje indeks *head*, a koniec indeks *tail*. Jeżeli  $head \leq tail$ , kolejka składa się z elementów  $Q[head..tail]$  (por. rys. 3.6b). W wyniku dodawania i usuwania elementów kolejka „przetacza się” przez tablicę, aż jej koniec osiągnie koniec tablicy, a wtedy dodanie kolejnego elementu do kolejki spowoduje umieszczenie go na początku tablicy. Podobnie „migruje” początek kolejki, gdy usuwane są z niej elementy.

Pewną niedogodnością cyklicznej implementacji tablicowej kolejki jest brak możliwości odróżnienia kolejki pustej od całkowicie zapełnionej, kiedy to następnikiem elementu wskazywanego przez indeks *tail* jest element wskazywany przez indeks *head*. Problem można rozwiązać, ograniczając rozmiar kolejki tak, by jeden element tablicy pozostawał niewykorzystany [1]. Można też na bieżąco pamiętać rozmiar  $n$  kolejki, co właśnie uczynimy. Przyjmując, że na początku  $head = 0$ ,  $tail = m - 1$  i  $n = 0$ , możemy obie operacje zaprogramować następująco:

```

procedure INJECT(Q, x)
  if n < m then
    tail := (tail + 1) mod m
    Q[tail] := x
    n := n+1
  else
    error('Kolejka zapełniona')

function POP(Q)
  if n > 0 then
    x := Q[head]
    head := (head + 1) mod m
    n := n-1
    return x
  else
    error('Kolejka pusta')

```

Implementację tablicową kolejki podwójnej łatwo otrzymujemy, rozszerzając opisaną wyżej implementację cykliczną o dwie operacje *push* i *eject* o złożoności czasowej  $O(1)$ . Pierwsza powoduje wstawienie elementu na lewym końcu kolejki, a druga usunięcie elementu z jej prawego końca:

```

procedure PUSH(Q, x)
  if n < m then
    head := (head + m - 1) mod m
    Q[head] := x
    n := n+1
  else
    error('Kolejka zapełniona')

function EJECT(Q)
  if n > 0 then
    x := Q[tail]
    tail := (tail + m - 1) mod m
    n := n-1
    return x
  else
    error('Kolejka pusta')

```

Poprzednikiem elementu o indeksie 0 jest w tablicy cyklicznej  $Q[0..m-1]$  element o indeksie  $m-1$ . Dlatego w powyższym kodzie przesunięcie indeksów *head* i *tail* o jedną pozycję w kierunku malejących indeksów wymagało użycia wyrażenia postaci  $(i + m - 1) \bmod m$ , a nie wyrażenia postaci  $(i - 1) \bmod m$ , które dla  $i = 0$  prowadziłyby do nieprawidłowego indeksu  $-1$  zamiast  $m - 1$ .

### 3.3. Implementacja dowiązaniowa listy

Użycie listy jednokierunkowej w języku Pascal wymaga zdefiniowania typu rekordowego, opisującego strukturę elementów listy, oraz związanego z nim typu wskaźnikowego, którego wartości wskazują na te elementy [11, 28]. Na przykład elementy listy liczb całkowitych można zdefiniować następująco:

```

type
  PElem = ^TElem;
  TElem = record
    x      : integer;
    next: PElem;
  end;

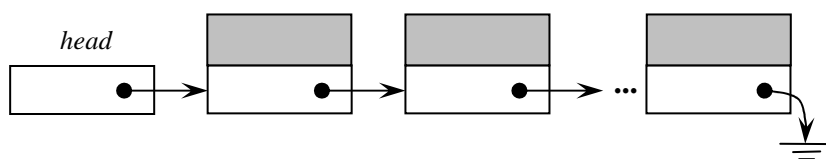
```

Pierwsza z tych definicji stwierdza, że wartości typu *PElem* są wskaźnikami do danych typu *TElem*, druga zaś orzeka, że każdy element typu *TElem* składa się z pola *x* typu całkowitego i wskaźnika *next* wskazującego na następny element typu *TElem*<sup>3</sup>. Jeśli *p* jest wskaźnikiem typu *PElem*, to *p*<sup>^</sup> jest rekordem typu *TElem*.

---

<sup>3</sup> Definicja typu *PElem* korzysta z nazwy *TElem*, która nie została jeszcze zdefiniowana. Jest to wyjątek od reguły zabraniającej użycia nazwy, której znaczenie nie zostało uprzednio określone.

Listę liczb całkowitych złożoną z powiązanych wskaźnikami elementów typu *TElem* ilustruje rysunek 3.7. Dodatkowa zmienna wskaźnikowa *head* wskazuje na pierwszy element listy. Każdy element listy zawiera pole *x* (oznaczone szarym prostokątem) i pole *next* zawierające wskaźnik do następnego elementu (przedstawiony za pomocą strzałki). Ostatni element nie ma następnika (strzałka wskazująca na „ziemię” oznacza wskaźnik pusty). W ogólnym przypadku *x* reprezentuje dane dowolnego typu przechowywane na liście.



Rys. 3.7. Lista w implementacji dowiązaniowej

Listę jednokierunkową można przebiegać w jednym kierunku, startując od jej pierwszego elementu i przesuwać się zgodnie ze wskaźnikami do elementów następnych, aż do napotkania elementu, który nie ma następnika. Poniższy przykład pokazuje, jak może wyglądać funkcja sekwencyjnego przeglądania listy złożonej z elementów typu *TElem* w poszukiwaniu wartości *k*. Wynikiem funkcji jest wskaźnik do znalezionej wartości, w którym przechowywana jest wartość *k*, lub wskaźnik pusty, gdy taki element nie istnieje.

```
function ListSearch(head: PElem; k: integer): PElem;
begin
  while (head <> nil) and (k <> head^.x) do
    head := head^.next;
  ListSearch := head;
end;
```

Równoważna definicja struktury w języku C++, opisującej elementy listy jednokierunkowej liczb całkowitych, ma postać następującą<sup>4</sup>:

```
struct Elem
{
  int x;
  Elem *next;
};
```

Znalezienie elementu listy o podanej wartości *k* pola *x* można zaprogramować w C++ podobnie jak w Pascalu:

<sup>4</sup> W języku C definicja struktury wygląda nieco inaczej [13].

```
Elem *ListSearch(Elem *head, int k)
{
    while (head != NULL && k != head->x)
        head = head->next;
    return head;
}
```

W obu wersjach funkcji *ListSearch* parametr *head* określający początek listy jest przekazywany przez wartość, więc zmiany jego wartości wewnątrz funkcji są niewidoczne na zewnątrz niej. Skorzystanie z tej sposobności pozwoliło na uniknięcie użycia pomocniczej zmiennej wskaźnikowej i zapisanie funkcji w bardziej zwartej postaci. Czas wykonania tej funkcji wynosi jednak  $O(n)$ , gdzie  $n$  jest liczbą elementów listy. Nawet gdy lista jest uporządkowana, nie ma możliwości zastosowania szybszego algorytmu przeszukiwania binarnego.

Przytoczone przykłady w języku Pascal i C++ pokazują, że realizowana w tych językach notacja wskaźnikowa wymaga szczegółowego definiowania składowych struktury dowiązaniowej. Ścisły opis struktur danych w języku programowania i wykonywanych na nich operacji, a zwłaszcza przydzielania elementom struktur dowiązaniowych pamięci i zwalniania jej [11, 13, 28, 30], jest obciążony wieloma szczegółami implementacyjnymi, które ograniczają zakres wykorzystania algorytmu do konkretnych typów danych i utrudniają zrozumienie istoty jego działania. Z tego względu struktury danych wyraża się w kategoriach abstrakcji [1], przez co nabierają one pewnych cech uniwersalności i sprawiają, że łatwo jest wychwycić ideę algorytmu i utworzyć jego konkretną realizację.

Aby sformułować na poziomie abstrakcyjnym operację przeszukiwania listy jednokierunkowej w implementacji dowiązaniowej przyjmujemy, że  $L$  oznacza listę elementów danego typu,  $x$  element tego typu, a *head* pierwszy element listy. Jeżeli *head* = **nil**, lista jest pusta. Użyjemy również **abstrakcyjnej notacji funkcyjnej**, w której  $next(x)$  oznacza następnik elementu  $x$  [3]<sup>5</sup>. Jeżeli  $next(x) = \mathbf{nil}$ , element  $x$  nie ma następnika. Możemy wreszcie przedstawić kod zapowiedzianej funkcji w pseudojęzyku programowania:

```
function LIST-SEARCH(L, k)
    x := head
    while (x ≠ nil) and (k ≠ x) do
        x := next(x)
    return x
```

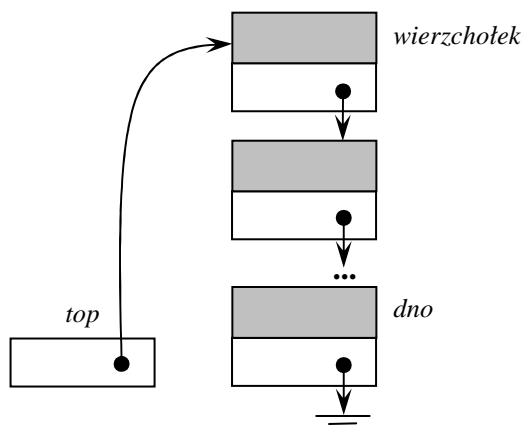
Uniwersalność funkcji *LIST-SEARCH* przejawia się w tym, że nie precyzujemy w niej typu elementów  $x$  listy  $L$  ani sposobu porównywania wartości  $k$  i  $x$ . Imple-

---

<sup>5</sup> Gwoli ścisłości należałoby również użyć zapisu  $head(L)$  zamiast *head*. Dla uproszczenia kodu podprogramów przyjmujemy, że takie jest znaczenie krótszego zapisu.

mentacje funkcji mogą być różne. Na przykład elementami listy mogą być rekordy składające się z szeregu pól, wśród których znajdują się pola *key* (klucz) i *next* (wskaźnik do następnego elementu), zaś parametr *L* wskaźnikiem do pierwszego elementu listy (wartością abstrakcyjnej funkcji *head*), a *k* wartością klucza szukanego elementu. Interesującą implementacją funkcji *LIST-SEARCH* w Delphi może być funkcja wykorzystująca komponent standardowej klasy *TList* omówionej w następnym podrozdziale niniejszego rozdziału.

Stos w implementacji dowiązaniowej jest najprostszą listą jednokierunkową, której operacje wstawiania i usuwania elementu (*push* i *pop*) są ograniczone tylko do jej lewego końca (*top*). Zwyczajowo ilustracja graficzna stosu ma postać przypominającą rzeczywisty stos książek lub skrzynek, z którego można je zdejmować w odwrotnej kolejności, niż były wstawiane (rys. 3.8).



Rys. 3.8. Stos w implementacji dowiązaniowej

Operacje *push* i *pop* działające na stosie można przestawić za pomocą następujących podprogramów w pseudojęzyku programowania:

```

procedure PUSH(L, x)
    next(x) := top
    top := x

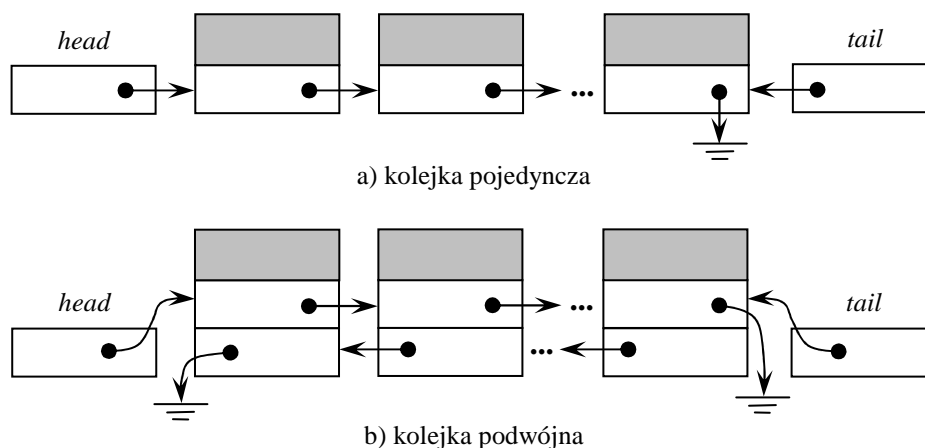
function POP(L)
    x := top
    if x ≠ nil then
        top := next(x)
        return x
    else
        error('Stos pusty')

```



Jak widać, wstawienie nowego elementu  $x$  na wierzchołek stosu sprowadza się do uaktualnienia dwóch wskaźników. Pierwsze określa, że następnikiem elementu  $x$  ma być dotychczasowy wierzchołek stosu, a drugie, że nowym wierzchołkiem stosu ma być element  $x$ . Z kolei zdjęcie elementu ze stosu polega na zapamiętaniu jego wskaźnika w pomocniczej zmiennej  $x$  i określeniu, że nowym wierzchołkiem ma być następnik elementu  $x$ . Oczywiście, próba zdjęcia elementu z pustego stosu jest sygnalizowana jako błąd. Obie operacje nie wymagają żadnego przesunięcia elementów w pamięci komputera, więc wykonują się w czasie  $O(1)$ .

Implementacja dowiązaniowa kolejki (rys. 3.9) jest nieco trudniejsza niż stosu, gdyż operacje wykonywane na kolejce mogą modyfikować wartości wskaźników do obu końców listy (*head* i *tail*). Ponadto w przypadku kolejki podwójnej jest wymagana obsługa dwóch wskaźników występujących w elementach listy, mianowicie *next* i *prev*. Jeżeli  $x$  jest elementem listy podwójnej, to *next*( $x$ ) jest, jak poprzednio, następnikiem  $x$ , zaś *prev*( $x$ ) jest poprzednikiem  $x$ . Jeśli *prev*( $x$ ) = **nil**, to  $x$  nie ma poprzednika, czyli jest pierwszym elementem listy.



Rys. 3.9. Kolejka w implementacji dowiązaniowej

Operacje *inject* i *pop* wstawienia elementu na koniec kolejki pojedynczej i usunięcia jej początkowego elementu można zaprogramować następująco:

```

procedure INJECT(L, x)
  if head  $\neq$  nil then
    next(tail) := x
  else
    head := x
    next(x) := nil
    tail := x

```

```

function POP(L)
  x := head
  if x ≠ nil then
    head := next(x)
    if head = nil then
      tail := nil
    return x
  else
    error('Kolejka pusta')

```

Wstawienie nowego elementu  $x$  na koniec kolejki pojedynczej wymaga sprawdzenia, czy nie jest ona pusta. Jeśli nie jest pusta, element  $x$  zostaje dowiązany do jej ostatniego elementu, a jeśli jest pusta, staje się jej pierwszym i jedynym elementem. Usunięcie pierwszego elementu kolejki pojedynczej przypomina operację *pop* wykonywaną na stosie, w której zamiast wskaźnika *top* używa się wskaźnika *head*. Obie operacje mają złożoność czasową  $O(1)$ .

Podobnie można zaprogramować operacje *push* i *pop* oraz *inject* i *eject* wstawiania i usuwania elementów na lewym i prawym końcu kolejki podwójnej:

```

procedure PUSH(L, x)
  next(x) := head
  if head ≠ nil then
    prev(head) := x
  else
    tail := x
  head := x
  prev(x) := nil

function POP(L)
  x := head
  if x ≠ nil then
    head := next(x)
    if head ≠ nil then
      prev(head) := nil
    else
      tail := nil
    return x
  else
    error('Kolejka pusta')

procedure INJECT(L, x)
  if head ≠ nil then
    next(tail) := x
  else
    head := x
  prev(x) := tail
  next(x) := nil
  tail := x

```

```

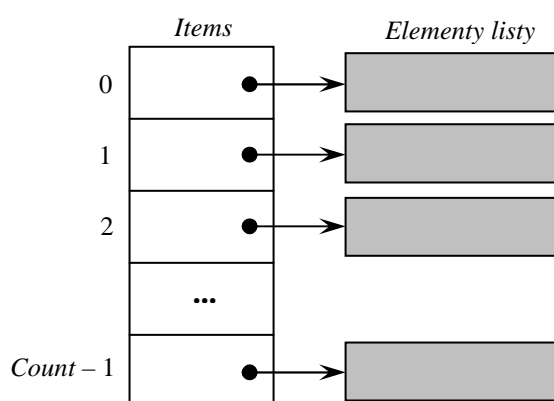
function EJECT(L)
  x := tail
  if x ≠ nil then
    tail := prev(x)
    if tail ≠ nil then
      next(tail) := nil
    else
      head := nil
  return x
else
  error('Kolejka pusta')

```

Niekiedy wygodnie jest korzystać z **listy cyklicznej** [3, 7, 15]. W pojedynczej liście cyklicznej ostatni element zawiera wskaźnik do pierwszego elementu, co oznacza, że następnikiem ostatniego elementu listy jest jej pierwszy element. W podwójnej liście cyklicznej dodatkowo poprzednikiem pierwszego elementu jest ostatni element listy.

### 3.4. Implementacje w Delphi

Podstawową klasą w Delphi reprezentującą listę jest *TList* [5, 6, 14]. Klasa ta działa na wskaźnikach ogólnego typu *pointer* przechowywanych we właściwości tablicowej *Items*. Na wskaźnikach można operować jak na elementach tablicy, ale dostęp do wskazywanych przez nie elementów wymaga odpowiedniej konwersji typu. Liczbę wskaźników określa właściwość *Count*, a ich indeksami są liczby całkowite od 0 do *Count* – 1 (rys. 3.10). Maksymalny rozmiar tablicy, określony przez właściwość *Capacity*, jest w razie potrzeby automatycznie powiększany.



Rys. 3.10. Implementacja listy w Delphi

Dostęp do elementów listy klasy *TList* i manipulowanie nimi umożliwia szereg metod, m.in. *Add* i *Insert* (dodanie), *Delete* i *Remove* (usunięcie), *Exchange* i *Move* (zamiana), *First* i *Last* (pierwszy i ostatni). Można nawet sortować elementy listy za pomocą metody *Sort*, która wymaga określenia funkcji porównywania dwóch elementów. Podprogram wykorzystuje metodę *quick sort*. Wykorzystując tablicę *Items*, do przeszukiwania posortowanej listy elementów można zastosować metodę przeszukiwania binarnego działającą w czasie  $O(\log n)$ . Począwszy od Delphi 6 można kopiować i łączyć listy za pomocą metody *Assign* [6, 14].

Klasą wywodzącą się od *TList* jest *TObjectList*. Podstawowa różnica między nimi polega na tym, że druga jest zdefiniowana jako lista obiektów, a nie jako lista wskaźników. Lista obiektów ma dodatkową właściwość *OwnsObjects*. Jeśli jest ustawiona na *true*, następuje automatyczne usuwanie obiektów, gdy są one w liście zastępowane przez inne obiekty, bądź gdy lista jest niszczone.

Listę łańcuchów reprezentuje klasa abstrakcyjna *TStrings*, która występuje w roli właściwości w wielu komponentach obsługujących łańcuchy<sup>6</sup>. Wywodzącą się od niej klasę *TStringList* można wykorzystać jako listę obiektów związanych z łańcuchami. Podobnie jak *TList*, klasy *TStrings* i *TStringList* mają wiele metod umożliwiających dostęp do elementów listy. Szczególnie użyteczna jest metoda *IndexOf*, która przegląda listę w poszukiwaniu łańcucha. Jej wartością jest indeks znalezionego łańcucha lub  $-1$ , gdy szukany łańcuch w liście nie występuje.

Jako przykład wykorzystania klasy *TStringList* rozpatrzmy problem konstrukcji **skorowidza** słów pliku tekstowego. Zakładamy, że oprócz normalnych znaków, o kodach od 32 do 255, plik zawiera pary znaków sterujących o kodach 13 i 10, tj. CR (ang. *carriage return*, powrót karetki) i LF (ang. *line feed*, wysuw wiersza), które kończą wiersze, i że słowami są spójne ciągi liter alfabetu polskiego, bez względu na sens i znaczenie. Małe i duże litery nie będą rozróżniane, ani nie będzie brana pod uwagę odmiana słów przez przypadki. Na przykład słowa *KOŃ* i *koń* będą traktowane jako te same, a *osioł* i *ośla* jako różne. Wynikiem wykonania aplikacji będzie uporządkowana alfabetycznie lista słów pisanych małą literą oraz listy numerów wierszy pliku wejściowego, w których te słowa występują<sup>7</sup>.

Projekt formularza okna aplikacji przedstawia rysunek 3.11. Na formularzu znajduje się komponent *MainMenu* obejmujący pozycje *Plik* i *Koniec*, komponent *OpenDialog* reprezentujący okienko dialogowe, które umożliwi wybór pliku wejściowego, oraz pole listy *ListBox*, w którym wyświetlony będzie skorowidz słów tego pliku. W komponencie *OpenDialog* warto ustawić właściwość *Filter* tak, by

<sup>6</sup> Klasa abstrakcyjna zawiera co najmniej jedną metodę niezaimplementowaną, która jest definiowana dopiero w klasie potomnej [6, 14].

<sup>7</sup> Rozwiązanie tego zagadnienia w postaci programów w językach Turbo Pascal i Borland C++ można znaleźć w książce [11].

ograniczyć dostęp tylko do plików pewnych rodzajów, np. plików tekstowych (\*.txt), plików Delphi (\*.dpr, \*.pas), plików C i C++ (\*.c, \*.cpp, \*.h), plików HTML (\*.html, \*.htm). Na koniec można zwyczajowo dopuścić do wyświetlenia wszystkich plików (\*.\*). Właściwość *Align* komponentu *ListBox* ustawiamy na *alClient*, a właściwość *Sorted* na *true*. Oznacza to, że lista zajmie cały obszar roboczy okna, a wyświetlone w niej pozycje będą posortowane alfabetycznie.



Rys. 3.11. Projekt formularza aplikacji generowania skorowidza

Utwórzmy najpierw pomocniczą metodę klasy formularza, która czyta plik tekstowy. Wynikiem jej działania ma być kolejne pobrane z pliku słowo i numer wiersza, w którym zostało ono znalezione. Jeśli dla uproszczenia pominiemy na razie polskie litery, metodę możemy zapisać w postaci następującej funkcji:

```
function TForm1.Slowo(var F: TextFile; var n: integer): string;
var
  c: char;
begin
  Result := '';
  c := ' ';
  while not Eof(F) and not (c in ['a'..'z', 'A'..'Z']) do
  begin
    Read(F, c);
    if c = #10 then Inc(n);
  end;
  while c in ['a'..'z', 'A'..'Z'] do
  begin
    Result := Result + c;
    if Eof(F) then c := ' ' else Read(F, c);
  end;
end;
```

Parametr  $F$  funkcji określa przeglądany plik, a  $n$  numer wiersza, w którym zostało znalezione słowo stanowiące jej wartość wynikową. Jeżeli cały plik został zanalizowany w poszukiwaniu kolejnego słowa, wynikiem funkcji jest łańcuch pusty. Działanie podprogramu opiera się na dwóch pętlach **while**. W pierwszej pomijane są wszystkie czytane znaki, które nie są literami. W przypadku znaku o kodzie 10, oznaczającego przejście do następnego wiersza, wartość parametru  $n$  jest zwiększana o 1. W drugiej pętli z pobieranych z pliku liter jest składane słowo, aż do napotkania znaku niealfabetycznego lub wykrycia końca pliku.

Możemy teraz przystąpić do precyzowania procedur obsługi zdarzeń *OnClick* pozycji menu *Plik* i *Koniec*. Cała praca aplikacji koncentruje się w pierwszej z tych procedur, ponieważ działanie drugiej sprowadza się jedynie do zamknięcia okna aplikacji, czyli wywołania metody *Close* klasy *TForm1*. Wstępna wersja pierwszej procedury może mieć postać następującą:

```
procedure TForm1.Plik1Click(Sender: TObject);
var
  Lista: TStringList;
  Plik: TextFile;
begin
  if OpenFileDialog1.Execute then
  begin
    AssignFile(Plik, OpenFileDialog1.FileName);
    Lista := TStringList.Create;
    try
      Reset(Plik);
      // Buduj skorowidz w komponencie Lista
      CloseFile(Plik);
      ListBox1.Clear;
      // Pokaż skorowidz w komponencie ListBox1
    finally
      // Usuń skorowidz z komponentu Lista
      Lista.Free;
    end;
  end;
end;
```

Wywołanie metody *Execute* komponentu *OpenDialog* powoduje wyświetlenie standardowego okna dialogowego Windows, w którym użytkownik może wybrać plik. Gdy potwierdzi wybór, nastąpi przypisanie zmiennej plikowej do rzeczywistego pliku, otwarcie tego pliku i utworzenie skorowidza występujących w nim słów, zamknięcie pliku i wyświetlenie skorowidza w polu listy *ListBox*, a w końcu usunięcie zbędnego już skorowidza. Lokalny obiekt *Lista* klasy *TStringList* służy jako pojemnik na dane skorowidza. Wywołanie konstruktora *Create* powoduje utworzenie obiektu *Lista*, a wywołanie metody *Free* zniszczenie tego obiektu. Operacje, które mogą uruchomić wyjątek, zostały umieszczone w bloku **try ... finally**,

przez co pamięć przydzielona obiektowi zostanie zwolniona niezależnie od tego, czy konstrukcja skorowidza się powiedzie, czy nie. Dalsza rozbudowa programu wymaga sprecyzowania trzech operacji oznaczonych komentarzami. Najwygodniej będzie zapisać je wszystkie w postaci metod klasy *TForm1*, a komentarze zastąpić wywołaniami tych metod.

Obiekt klasy *TStringList* zawiera zarówno listę łańcuchów, jak i listę obiektów związanych z łańcuchami. Obie listy są implementowane za pomocą tablic wskaźników *Strings* i *Objects*, których elementy są indeksowane od 0 do *Count* – 1. Dzięki temu możemy każde słowo skorowidza przechowywane w tablicy *Strings* związać z obiektem klasy *TList* w tablicy *Objects* zawierającym listę numerów wierszy pliku, w których to słowo występuje. Wartości typów *integer* i *pointer* zajmują tyle samo bajtów, więc wygodnie będzie przechowywać numery wierszy w tablicy *Items* obiektu *TList* zamiast wskaźników.

W metodzie budowy skorowidza należy pobierać w pętli kolejne słowa z pliku i dodawać je do listy *Strings* obiektu klasy *TStringList*. Jest oczywiste, że dodanie słowa wymaga uprzedniego upewnienia się, że nie występuje ono na liście (metoda *IndexOf*). Wraz z wstawieniem nowego słowa należy utworzyć nowy obiekt klasy *TList* i wstawić go do listy *Objects* (metoda *AddObject*). Kolejną operacją, którą trzeba wykonać po odnalezieniu słowa na liście lub wstawieniu go do niej, jest uaktualnienie listy numerów wierszy przechowywanych w skojarzonym ze słowem obiekcie *TList*. I tu trzeba upewnić się, że numer wiersza, w którym słowo zostało znalezione, nie występuje na liście, by nie wprowadzić go ponownie. Rozważania te prowadzą do następującej procedury:

```
procedure TForm1.BudujSkorowidz(var F: TextFile; L: TStringList);
var
  k, n: integer;
  s: string;
begin
  n := 1;
  repeat
    s := Slowo(F, n);
    if s = '' then Break;
    k := L.IndexOf(s);
    if k < 0 then k := L.AddObject(s, TList.Create);
    with L.Objects[k] as TList do
      if IndexOf(pointer(n)) < 0 then Add(pointer(n));
    until false;
  end;
```

Aby pod koniec kodu powyższej procedury móc skorzystać z metod *IndexOf* i *Add* klasy *TList*, przekształciliśmy za pomocą operatora **as** obiekt ogólnej klasy *TObject* na obiekt klasy *TList*. Druga konwersja dotyczyła przekształcenia wartości typu *integer* na wartość typu *pointer*.

Aby wygenerowany skorowidz wyświetlić w polu listy *ListBox1*, trzeba jego elementy wydobyć z obiektu klasy *TStringList*, dołączając do każdego słowa przekształcone na łańcuchy numery wierszy przechowywane jako wskaźniki. Kod tej metody może wyglądać następująco:

```

procedure TForm1.PokazSkorowidz(L: TStringList);
var
    i, k: integer;
    s: string;
begin
    for k:=0 to L.Count-1 do
    begin
        s := L.Strings[k];
        with L.Objects[k] as TList do
            for i:=0 to Count-1 do
                s := s + ' ' + IntToStr(integer(Items[i]));
            ListBox1.Items.Add(s);
        end;
    end;

```

Obiekty skojarzone z łańcuchami przechowywanymi na liście klasy *TStringList* nie są zwalniane ani usuwane z pamięci, gdy sama lista jest niszczone. Zatem aby nie doprowadzić do „wycieku” pamięci (ang. *memory leak*), gdy program skończy działanie, powinniśmy je zwolnić. Dokonujemy tego w zapowiedzianej w komentarzu metodzie usuwania skorowidza:

```

procedure TForm1.UsunSkorowidz(L: TStringList);
begin
    while L.Count > 0 do
    begin
        L.Objects[0].Free;
        L.Delete(0);
    end;
end;

```

Metoda zwalnia zerowy obiekt i usuwa zerowy łańcuch z listy, aż do usunięcia wszystkich elementów. Można by również zwolnić same obiekty, przebiegając je wszystkie od indeksu 0 do *Count* – 1, a potem zwolnić wszystkie łańcuchy na raz za pomocą metody *Clear*.

Możemy wreszcie zastąpić komentarze w procedurze *Plik1Click* wywołaniami metod *BudujSkorowidz*, *PokazSkorowidz* i *UsunSkorowidz*. W ostatecznej wersji modułu formularza wprowadzamy jeszcze trzy poprawki. I tak, w funkcji *Slowo* rozszerzamy zestaw liter o polskie znaki, w procedurze *BudujSkorowidz* wszystkie duże litery w pobranym z pliku słowie przekształcamy na małe litery za pomocą funkcji *AnsiLowerCase*, a w procedurze *PokazSkorowidz* oddzielamy przecinkami numery wierszy. Oto kod otrzymanego modułu formularza:



```

unit MainUnit;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls, Menus;

type
    TForm1 = class(TForm)
        MainMenu: TMainMenu;
        Plik1: TMenuItem;
        Koniec1: TMenuItem;
        OpenDialog1: TOpenDialog;
        ListBox1: TListBox;
        procedure Plik1Click(Sender: TObject);
        procedure Koniec1Click(Sender: TObject);
    public
        function Slowo(var F: TextFile; var n: integer): string;
        procedure BudujSkorowidz(var F: TextFile; L: TStringList);
        procedure PokazSkorowidz(L: TStringList);
        procedure UsunSkorowidz(L: TStringList);
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

function TForm1.Slowo(var F: TextFile; var n: integer): string;
const
    L = ['a'..'z', 'a', 'ć', 'ę', 'ł', 'ń', 'ó', 'ś', 'ź', 'ż',
        'A'..'Z', 'A', 'Ć', 'Ę', 'Ł', 'Ń', 'Ó', 'Ś', 'Ź', 'Ż'];
var
    c: char;
begin
    Result := '';
    c := ' ';
    while not Eof(F) and not (c in L) do
    begin
        Read(F, c);
        if c = #10 then Inc(n);
    end;
    while c in L do
    begin
        Result := Result + c;
        if Eof(F) then c := ' ' else Read(F, c);
    end;
end;

```

```
procedure TForm1.BudujSkorowidz(var F: TextFile; L: TStringList);
var
  k, n: integer;
  s: string;
begin
  n := 1;
  repeat
    s := AnsiLowerCase(Slowo(F, n));
    if s = '' then Break;
    k := L.IndexOf(s);
    if k < 0 then k := L.AddObject(s, TList.Create);
    with L.Objects[k] as TList do
      if IndexOf(pointer(n)) < 0 then Add(pointer(n));
    until false;
  end;

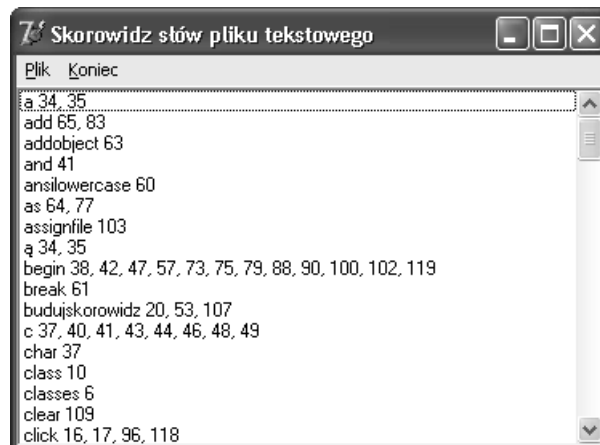
procedure TForm1.PokazSkorowidz(L: TStringList);
var
  i, k: integer;
  s: string;
begin
  for k:=0 to L.Count-1 do
  begin
    s := L.Strings[k];
    with L.Objects[k] as TList do
      for i:=0 to Count-1 do
      begin
        if i>0 then s := s + ',';
        s := s + ' ' + IntToStr(integer(Items[i]));
      end;
      ListBox1.Items.Add(s);
    end;
  end;

procedure TForm1.UsunSkorowidz(L: TStringList);
begin
  while L.Count > 0 do
  begin
    L.Objects[0].Free;
    L.Delete(0);
  end;
end;

procedure TForm1.Plik1Click(Sender: TObject);
var
  Lista: TStringList;
  Plik: TextFile;
begin
  if OpenFileDialog1.Execute then
  begin
    AssignFile(Plik, OpenFileDialog1.FileName);
```

```
Lista := TStringList.Create;  
try  
    Reset(Plik);  
    BudujSkorowidz(Plik, Lista);  
    CloseFile(Plik);  
    ListBox1.Clear;  
    PokazSkorowidz(Lista);  
finally  
    UsunSkorowidz(Lista);  
    Lista.Free;  
end;  
end;  
end;  
  
procedure TForm1.Koniec1Click(Sender: TObject);  
begin  
    Close;  
end;  
  
end.
```

Przykładowy wynik wykonania aplikacji generowania skorowidza słów pliku tekstowego jest pokazany na rysunku 3.12.



Rys. 3.12. Wynik wykonania aplikacji generowania skorowidza

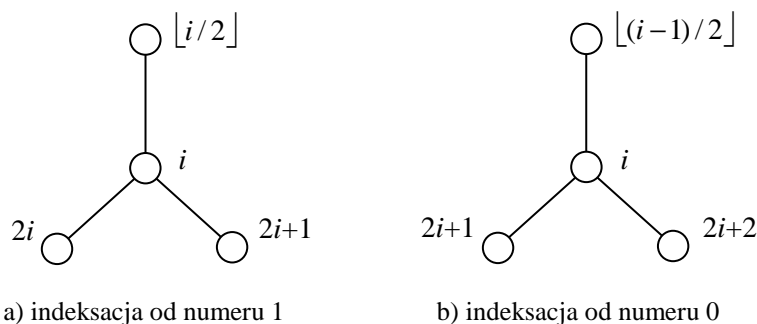
Delphi zawiera klasy *TStack* i *TObjectStack* reprezentujące stos wskaźników i obiektów zbudowany zgodnie z zasadą LIFO. Dostęp do elementów odbywa się w nich za pomocą metod: *Push* do wstawiania elementu na wierzchołek stosu, *Pop* do pobierania elementu z wierzchołka stosu i *Peek* do odczytywania elementu z wierzchołka bez usuwania go. Istnieją również klasy *TQueue* i *TObjectQueue*

reprezentujące kolejkę wskaźników i obiektów. Podobnie jak klasy *TStack* i *TObjectStack*, posiadają one metody *Push*, *Pop* i *Peek*, ale działające według zasady FIFO. Lista standardowych klas w Delphi reprezentujących wiele innych, mniej lub bardziej złożonych struktur danych jest obszerna. Najogólniej można je podzielić na proste listy, kolekcje i kontenery [6].

### 3.5. Kolejki priorytetowe

Sortowanie kopcowe, omówione w podrozdziale 2.5, opiera się na interesującej strukturze zwanej **kopcem**, **stogiem** lub **stertą** (ang. *heap*). Sortowany ciąg elementów  $a_1, a_2, \dots, a_n$  jest reprezentowany przez pełne drzewo binarne, które jest wypełnione na wszystkich poziomach, z wyjątkiem być może ostatniego, spójnie wypełnionego od lewej strony do pewnego miejsca. Dzięki regularnemu kształtowi drzewo takie może być reprezentowane w tablicy bez dodatkowych dowiązań. Wierzchołki drzewa, przechowywane w kolejnych elementach tablicy  $a[1..n]$ , mają specyficzną indeksację (por. rys. 3.13a):

- ♦ korzeń ma indeks 1,
- ♦ następniki wierzchołka o indeksie  $i$  (o ile istnieją) mają indeksy  $2i$  i  $2i + 1$ ,
- ♦ poprzednik wierzchołka o indeksie  $i$  (różnego od korzenia) ma indeks  $\lfloor i/2 \rfloor$ .



**Rys. 3.13.** Indeksacja wierzchołków kopca pełnego

W sortowaniu kopcowym wykorzystuje się specjalne ustawianie wierzchołków struktury, nazywane **uporządkowaniem kopcowym**, w którym wartość każdego wierzchołka jest nie mniejsza od wartości jego następników. Uporządkowanie to sprawia, że w korzeniu drzewa znajduje się największy element, a na ścieżkach drzewa, od korzenia do liścia (wierzchołka, który nie ma następnika), elementy są uporządkowane nierosnąco. Skoro największy element znajduje się w korzeniu

kopca, zamienia się go z elementem znajdującym się na ostatniej pozycji, po czym zmniejsza liczbę wierzchołków drzewa o 1. Inaczej mówiąc, element maksymalny jest usuwany z korzenia, a w jego miejsce jest wstawiany ostatni liść drzewa. Po tej operacji uporządkowanie kopcowe może zostać zaburzone, więc należy go przywrócić. Nowy korzeń jest poprzez zamianę z większymi od niego następnikami przesiewany w dół kopca dopóty, dopóki nie znajdzie się w odpowiednim miejscu. Opierając się na omówionym w podrozdziale 2.5 algorytmie przesiewania, możemy operację usuwania największego elementu kopca, reprezentowanego przez tablicę  $a[1..n]$ , zapisać następująco:

```
function deletemax(a)
  if n < 1 then
    error('Kopiec pusty')
  max := a[1]
  x := a[n]
  n := n-1
  if n = 0 then return max
  i := 1
  j := 2*i
  while j ≤ n do
    if j < n then
      if a[j].key < a[j+1].key then j := j+1
    if x.key ≥ a[j].key then break
    a[i] := a[j]
    i := j
    j := 2*i
  a[i] := x
  return max
```

Równie łatwo można zaprogramować operację odwrotną, którą jest wstawienie nowego elementu  $x$  do kopca. Najpierw należy ten element dołożyć na koniec struktury, rozszerzając liczbę elementów tablicy o 1, przez co stanie się on nowym, ostatnim liściem drzewa. Następnie trzeba przywrócić drzewu uporządkowanie kopcowe, przesuwając element  $x$  w górę poprzez zamienianie go z mniejszymi od niego poprzednikami. Stosowny kod może wyglądać następująco:

```
procedure insert(a, x)
  n := n+1
  a[n] := x
  j := n
  i := j div 2
  while (j > 1) and (a[i].key < x.key) do
    a[j] := a[i]
    j := i
    i := j div 2
  a[j] := x
```

Wartości kluczy elementów kopca są nazywane **priorytetami**, a uzyskana struktura **kolejką priorytetową**. Zazwyczaj priorytetem jest liczba całkowita lub rzeczywista, ale może nim być element dowolnego zbioru liniowo uporządkowanego. Podstawowymi operacjami wykonywanymi na kolejce priorytetowej są:

- ◆ uzyskanie dostępu do elementu o najwyższym priorytecie,
- ◆ usunięcie elementu o najwyższym priorytecie (*deletemax*),
- ◆ wstawienie nowego elementu do kolejki (*insert*).

Pierwsza z powyższych operacji ma złożoność czasową  $O(1)$ . Pozostałe dwie mają złożoność pesymistyczną  $O(\log n)$ , ponieważ liczby przesunięć elementów w dół lub w górę kopca nie mogą przekroczyć jego wysokości, która na podstawie wzoru (2.5) wynosi  $\lfloor \log n \rfloor$ .

Przykładem kolejki priorytetowej ze świata rzeczywistego jest kolejka pacjentów oczekujących na usługę w oddziale pomocy doraźnej. Kolejność ich obsługi jest inna niż np. klientów w kolejce sklepowej. Jest ona bowiem określona przez priorytet potrzeb osób przebywających w kolejce, którym jest stopień zagrożenia życia, a nie przez moment ich przybycia do kolejki.

Innym przykładem jest zbiór zadań realizowanych współbieżnie przez wielozadaniowy system operacyjny. Szeregowanie zadań i ich wykonywanie odbywa się zgodnie z priorytetami, jakie są im przypisane. Zadaniom o wyższych priorytetach system przydziela więcej czasu procesora i zasobów pamięciowych. W przypadku, gdy liczba zadań jest zbyt duża, zadania o niskim priorytecie mogą długo oczekiwać, a niekiedy mogą się nawet nie doczekać na czas procesora<sup>8</sup>.

Kolejkę priorytetową przypomina kolejka komunikatów (ang. *message queue*), którą prowadzi system MS Windows dla każdego działającego pod jego nadzorem programu<sup>9</sup>. Komunikaty te są obsługiwane niekoniecznie w kolejności ich pojawiania się. Na przykład komunikat *WM\_PAINT* określający, że obszar roboczy okna lub jego fragment wymaga przemalowania, ma dość niski priorytet. Oznacza to, że stosowny obszar zostanie odmalowany dopiero wtedy, gdy kolejka nie będzie zawierała ważniejszych komunikatów. Niski priorytet ma również generowany przez zegar komunikat *WM\_TIMER*, który oznacza, że upłynął zadany przedział czasu. Odtwarzanie animacji opartej na tym komunikacie może być zaburzane, gdy system w nawale zajęć nie nadąży obsługiwać ważniejszych komunikatów.

Biblioteka komponentów VCL w Delphi nie zawiera klasy obsługującej kolejkę priorytetową. Zdefiniujemy więc taką klasę, upodabniając ją do klas *TStack* i *TQueue*, które wywodzą się od klasy *TOrderedList*. Nazwiemy ją *THeap*. Klasa

---

<sup>8</sup> Zjawisko to nosi nazwę zagłodzenia procesu (ang. *process starvation*).

<sup>9</sup> Proces obsługi komunikatów Windows jest bardziej zagmatwany [20].

*TOrderedList* udostępnia m.in. właściwości *Count* (liczba elementów listy) i *List* (lista wskaźników do tych elementów), publiczne metody *Peek* (odczyt wskaźnika bez jego usuwania), *Pop* (usunięcie wskaźnika) i *Push* (wstawienie wskaźnika) oraz chronione metody *PeekItem*, *PopItem* i *PushItem*. Trzy ostatnie metody wymagają pokrycia w klasie pochodnej, w nich bowiem mają być ukryte szczegóły implementacyjne wymienionych wyżej operacji właściwych kolejce priorytetowej, dostępnych za pomocą metod *Peek*, *Pop* i *Push*.

Niestety, przesuwanie wierzchołka w dół lub w górę kopca wymaga porównywania kluczy (priorytetów), a tej operacji nie potrafimy w ogólnym przypadku zaprogramować. Dlatego klasa *Theap* będzie zawierać metodę abstrakcyjną, którą trzeba definiować w podklasach realizujących konkretne wersje kolejek priorytetowych. Nazwijmy tę metodę *Compare*. Zakładamy, że porównuje ona dwa klucze, a wynikiem porównania jest liczba całkowita ujemna, gdy pierwszy jest mniejszy od drugiego, zero, gdy są równe, liczba całkowita dodatnia, gdy pierwszy jest większy od drugiego<sup>10</sup>. Rozważania te prowadzą do następującej definicji:

```
type
  Theap = class(TOrderedList)
  protected
    function Compare(Item1, Item2: pointer): integer; virtual;
    abstract;
  private
    function PeekItem: pointer; override;
    function PopItem: pointer; override;
    procedure PushItem(AItem: pointer); override;
  end;
```

Metoda *Compare* jest zadeklarowana nie tylko jako abstrakcyjna (**abstract**), lecz także jako wirtualna (**virtual**), co pozwala na jej definiowanie w klasie potomnej. Dostęp do niej w klasach potomnych będzie możliwy, ponieważ została umieszczona w obszarze chronionym klasy (**protected**).

Metody *PeekItem*, *PopItem* i *PushItem* pokrywają (**override**) dotychczasowe metody wirtualne klasy bazowej *TOrderedList*, w której zostały zadeklarowane jako chronione. Zawężenie dostępu do tych metod poprzez umieszczenie ich w obszarze prywatnym klasy (**private**) uniemożliwi modyfikację sposobu ich działania w klasach potomnych. Uzasadnieniem takiego rozwiązania jest pełne zdefiniowanie tych metod i chęć ukrycia ich przed użytkownikami klasy *Theap*.

Implementacja metody *PeekItem* jest bardzo prosta i sprowadza się do jednej instrukcji przypisania, w której pobiera się wartość pierwszego elementu listy *List*. Z kolei metody *PopItem* i *PushItem* łatwo jest zaprogramować, opierając się na

<sup>10</sup> Taka konwencja jest często stosowana w językach C, C++ i Java.

przytoczonych wyżej kodach operacji *deletemax* i *insert*. Należy jednak zwrócić uwagę na inną numerację elementów tablicy reprezentującej kopiec (od 1 do  $n$ ) niż elementów listy (od 0 do  $Count - 1$ ). Prowadzi to do innej zasady obliczania indeksów sąsiednich wierzchołków kopca (zob. rys. 3.13b). Uwzględniając tę różnicę i wykorzystując metody obiektu *List*, możemy zbudować następujący moduł definiujący klasę *Theap*:

```

unit HeapUnit;

interface

uses
  Contnrs;

type
  THeap = class(TOrderedList)
  protected
    function Compare(Item1, Item2: pointer): pnteger; virtual;
    abstract;
  private
    function PeekItem: pointer; override;
    function PopItem: pointer; override;
    procedure PushItem(AItem: pointer); override;
  end;

implementation

function THeap.PeekItem: pointer;
begin
  Result := List.First;
end;

function THeap.PopItem: pointer;
var
  i, j, n: integer;
  x: pointer;
begin
  with List do
  begin
    Result := First;
    x := Last;
    Count := Count-1;
    if Count <= 0 then Exit;
    n := Count-1;
    i := 0;
    j := 2*i+1;
    while j<=n do
    begin
      if j < n then
        if Compare(Items[j], Items[j+1]) < 0 then Inc(j);

```



```

        if Compare(x, Items[j]) >= 0 then Break;
        Items[i] := Items[j];
        i := j;
        j := 2*i+1;
    end;
    Items[i] := x;
end;
end;

procedure THeap.PushItem(AItem: pointer);
var
    x: pointer absolute AItem;
    i, j: integer;
begin
    with List do
    begin
        j := Count;
        Add(x);
        i := (j-1) div 2;
        while (j > 0) and (Compare(Items[i], x) < 0) do
        begin
            Items[j] := Items[i];
            j := i;
            i := (j-1) div 2;
        end;
        Items[j] := x;
    end;
end;

end.

```

Podobnie jak standardowe klasy *TStack* i *TQueue*, klasa *THeap* reprezentuje listę wskaźników, które można obsługiwać za pomocą metod *Peek*, *Pop* i *Push*. Metody te zachowują się jednak odmiennie, bo zgodnie z zasadą działania kolejki priorytetowej. Definiując klasę wywodzącą się od klasy *THeap*, należy określić typ elementów przechowywanych w kolejce i funkcję *Compare* porównującą priorytety dwóch takich elementów wskazywanych przez parametry *Item1* i *Item2*. Jeśli np. klasa *TFloatHeap* reprezentuje kolejkę priorytetową elementów typu *real*, funkcję *Compare* można zdefiniować następująco:

```

function TFloatHeap.Compare(Item1, Item2: pointer): integer;
begin
    if real(Item1^) < real(Item2^) then Result := -1 else
    if real(Item1^) > real(Item2^) then Result := 1 else
        Result := 0;
end;

```

Działanie takiej kolejki możemy sprawdzić w praktyce, budując prostą aplikację, która generuje losowo liczby rzeczywiste i wstawia je do kolejki, a potem je

z niej pobiera. Projekt formularza tej aplikacji jest pokazany na rysunku 3.14. W pierwszym polu listy *ListBox*, opisanym etykietą *Dane*, zostanie wyświetlony ciąg liczb wstawianych do kolejki, a w drugim, opisanym etykietą *Wynik*, ciąg tych samych liczb pobieranych z kolejki. Pole edycji *SpinEdit*, którego właściwości *Value*, *MinValue* i *MaxValue* ustawiamy np. na 10, 1 i 100, posłuży do określania długości ciągu, a przycisk *Utwórz* do rozpoczęcia operacji zapełniania kolejki liczbami rzeczywistymi i opróżniania jej.



**Rys. 3.14.** Projekt formularza aplikacji obsługującej kolejkę priorytetową

Rozbudowę kodu modułu formularza rozpoczynamy od rozszerzenia jego listy **uses** o nazwę *HeapUnit* modułu definiującego klasę *THeap* i zdefiniowania klasy potomnej *TFloatHeap* w części **implementation**:

```
type
  TFloatHeap = class(THeap)
  protected
    function Compare(Item1, Item2: pointer): integer; override;
  end;
```

Oczywiście, aby klasa *TFloatHeap* była kompletnie zdefiniowana, włączamy do programu przytoczony wyżej kod metody *Compare*. Następnie w prywatnej części klasy *TForm1* deklarujemy pole, które będzie zawierało odwołanie do obiektu reprezentującego kolejkę liczb rzeczywistych. Nie może ono być typu *TFloatHeap*, gdyż jego deklaracja wyprzedza definicję klasy *TFloatHeap*. Często stosowanym rozwiązaniem jest w takich przypadkach wykorzystanie klasy bazowej zamiast klasy potomnej. Oto poprawna deklaracja tego pola:

```
Heap: THeap;
```

Zgodnie z zasadą kompatybilności klas, pole *Heap* może odnosić się do obiektu klasy *TFloatHeap* wywodzącej się od klasy *THeap*<sup>11</sup>. Obiekt klasy *TFloatHeap* stworzymy w procedurze obsługi zdarzenia *OnCreate*, a niszczymy w procedurze obsługi zdarzenia *OnDestroy* formularza:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Randomize;
    Heap := TFloatHeap.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    Heap.Free;
end;
```

Możemy teraz zaprogramować operacje zapełniania kolejki, reprezentowanej przez obiekt *Heap* klasy *TFloatHeap*, losowymi liczbami rzeczywistymi i opróżniania jej. Obie operacje zapisujemy jako metody klasy *TForm1*:

```
procedure TForm1.ZapelnijKolejke;
var
    k: integer;
    p: ^real;
begin
    for k:=1 to SpinEdit1.Value do
    begin
        New(p);
        p^ := 100*Random;
        Heap.Push(p);
        ListBox1.Items.Add(Format('%7.2f', [p^]));
    end;
end;

procedure TForm1.OproznijKolejke;
var
    p: ^real;
begin
    while Heap.Count > 0 do
    begin
        p := Heap.Pop;
        ListBox2.Items.Add(Format('%7.2f', [p^]));
        Dispose(p);
    end;
end;
```

---

<sup>11</sup> Według tej zasady można używać klasy potomnej wszędzie tam, gdzie oczekiwana jest klasa bazowa [5, 6]. Odwrotne zastępowanie klas jest niedozwolone.

Klasa *TFloatHeap* reprezentuje kolejkę priorytetową, obsługując jej elementy za pośrednictwem wskaźników, ale nie jest właścicielem wstawianych do kolejki i usuwanych z niej elementów. Przydzielanie pamięci elementom i zwalnianie jej, gdy stają się niepotrzebne, nie odbywa się automatycznie podczas wykonywania operacji *Push* i *Pop*. Dlatego w podprogramach *ZapelnijKolejke* i *OproznijKplejke* używa się pomocniczej zmiennej wskaźnikowej do elementu typu *real* oraz stosuje tradycyjne procedury *New* i *Dispose*, które służą do dynamicznego przydzielania i zwalniania pamięci [11, 25, 26, 28].

Aby zakończyć aplikację, definiujemy procedury obsługi zdarzeń przycisków *Utwórz* i *Zamknij* widniejących na formularzu. Kod pierwszej sprowadza się do wyczyszczenia obydwu pól listy *ListBox* oraz wywołania metod *ZapelnijKolejke* i *OproznijKolejke*, a drugiej do wywołania metody *Close* klasy *TForm1*. Oto pełny kod modułu formularza tej klasy:

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls, Spin, HeapUnit;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    ListBox2: TListBox;
    SpinEdit1: TSpinEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Button1: TButton;
    Button2: TButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    Heap: THeap;
    procedure ZapelnijKolejke;
    procedure OproznijKolejke;
  end;

var
  Form1: TForm1;

implementation
```

```
{ $R *.dfm }

type
  TFloatHeap = class(THeap)
  protected
    function Compare(Item1, Item2: pointer): integer; override;
  end;

function TFloatHeap.Compare(Item1, Item2: Pointer): integer;
begin
  if real(Item1^) < real(Item2^) then Result := -1 else
    if real(Item1^) > real(Item2^) then Result := 1 else
      Result := 0;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  Heap := TFloatHeap.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  Heap.Free;
end;

procedure TForm1.ZapelnijKolejke;
var
  k: integer;
  p: ^real;
begin
  for k:=1 to SpinEdit1.Value do
  begin
    New(p);
    p^ := 100*Random;
    Heap.Push(p);
    ListBox1.Items.Add(Format('%7.2f', [p^]));
  end;
end;

procedure TForm1.OproznijKolejke;
var
  p: ^real;
begin
  while Heap.Count > 0 do
  begin
    p := Heap.Pop;
    ListBox2.Items.Add(Format('%7.2f', [p^]));
    Dispose(p);
  end;
end;
```

```

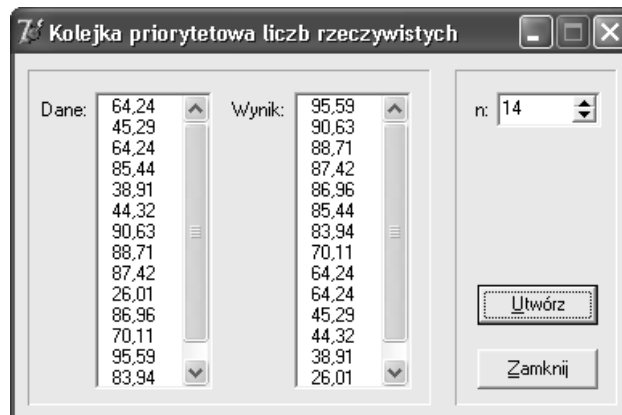
procedure TForm1.Button1Click(Sender: TObject);
begin
    ListBox1.Clear;
    ListBox2.Clear;
    ZapelnijKolejke;
    OproznijKolejke;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Close;
end;

end.

```

Rysunek 3.15 przedstawia przykładowy wynik wykonania aplikacji generującej kolejkę priorytetową liczb rzeczywistych. Lista po lewej stronie pokazuje ciąg liczb wstawianych do kolejki, a po prawej pobieranych z niej. Jak widać, ciąg wynikowy jest uporządkowany niemalejąco.



Rys. 3.15. Wynik wykonania aplikacji obsługującej kolejkę priorytetową

## 3.6. Zbiory

Zbiór składa się z elementów, które nie powtarzają się i nie występują w żadnej określonej kolejności. Na przykład matematyczne zapisy  $\{1, 3\}$  i  $\{3, 1\}$  oznaczają ten sam zbiór. W informatyce zbiory mają skończoną liczbę elementów i mogą się zmieniać w czasie poprzez wykonywanie na nich różnych operacji, np. mogą się zwiększać lub zmniejszać. Elementami zbiorów są często liczby całkowite lub rzeczywiste, znaki, łańcuchy i rekordy. W przypadku rekordów jedno z pól jest

zwykle wyróżnione jako **klucz** (ang. *key*). Jeżeli klucz jednoznacznie identyfikuje rekord, zbiór rekordów można traktować jako zbiór kluczy, ponieważ pozostałe pola nie mają znaczenia dla realizacji zbioru.

W matematyce na zbiorach rozpatruje się takie operacje, jak suma, różnica, różnica symetryczna i iloczyn zbiorów, a także operacje zawierania się zbiorów, oznaczane symbolami  $A \cup B$ ,  $A - B$ ,  $A \div B$ ,  $A \cap B$ ,  $A \subset B$  i  $A \subseteq B$ . W konkretnych zastosowaniach informatycznych mogą być wymagane implementacje niektórych z tych operacji. Najczęściej wykonywanymi na zbiorach operacjami są:

- ♦  $insert(S, x)$  – wstawienie elementu  $x$  do zbioru  $S$ ,
- ♦  $delete(S, x)$  – usunięcie elementu  $x$  ze zbioru  $S$ ,
- ♦  $member(S, x)$  – sprawdzenie, czy element  $x$  należy do zbioru  $S$ .

Jest oczywiste, że operacja  $insert$  nie wprowadza elementu  $x$  do zbioru  $S$ , gdy  $x \in S$ , zaś operacja  $delete$  nie usuwa elementu  $x$  ze zbioru  $S$ , gdy  $x \notin S$ .

Powszechnie zbiory reprezentuje się za pomocą listy. Wynika stąd, że chociaż nie określa się żadnej kolejności elementów zbioru, w rzeczywistości są one ustawione w pewnym porządku. Stosuje się wszystkie rodzaje list do reprezentowania zbioru. Złożoność obliczeniowa podstawowych operacji na zbiorach zależy od liczby  $n$  elementów zbioru i wynosi zwykle  $O(n)$ , ale w niektórych implementacjach listy może być inna. Na przykład implementacje list uporządkowanych w Delphi, w których korzysta się z dynamicznych tablic wskaźników do elementów zbioru, umożliwiają wykonywanie wymienionych operacji podstawowych w czasie  $O(\log n)$ .

W przypadku, gdy w rozpatrywanym zagadnieniu wszystkie używane zbiory są podzbiorami pewnego uniwersalnego zbioru  $U$  (uniwersum) o niezbyt dużym rozmiarze  $N$ , każdemu elementowi zbioru  $U$  można przypisać jednoznacznie liczbę całkowitą od 1 do  $N$ , a wówczas wygodną reprezentacją zbiorów tych elementów są tzw. **wektory charakterystyczne**, które są tablicami o elementach typu *boolean* indeksowanymi od 1 do  $N$ . Jeżeli element  $x \in U$  jest identyfikowany za pomocą liczby  $i \in \{1, 2, \dots, N\}$  i  $C$  jest wektorem charakterystycznym zbioru  $S$ , to:

$$C[i] = \begin{cases} false & (x \notin S) \\ true & (x \in S) \end{cases}$$

Wektory charakterystyczne można zapisywać jako tablice o elementach typu *boolean*. Oszczędną pamięciowo ich reprezentacją są **wektory bitów**, w których bit 0 oznacza wartość *false*, a bit 1 wartość *true*<sup>12</sup>. Podstawowe operacje na zbiorach

<sup>12</sup> Taka jest implementacja zbiorów w Pascalu definiowanych jako **set of ...**. Ich rozmiar jest ograniczony do 256 elementów.

rach implementowanych za pomocą wektorów charakterystycznych wykonują się w czasie  $O(1)$ , gdyż wymagają jedynie dostępu do elementu tablicy reprezentującej zbiór, którego dotyczą.

Wygodną konstrukcją językową w pseudojęzyku programowania, ułatwiającą zapis i zrozumienie algorytmów działających na zbiorach, jest **for each ...**. Używa się jej do zapisu operacji wykonywanych dla każdego elementu należącego do określonego zbioru. Jeśli np. zbiór  $L$  reprezentuje wylosowany zestaw numerów totolotka, wypisanie ich wszystkich można zaprogramować następująco:

```
for each  $x \in L$  do  
  Write( $x$ )
```

a usunięcie ze zbioru  $A$  wszystkich elementów zbioru  $B$ , czyli operację  $A := A - B$ , następująco:

```
for each  $x \in B$  do  
  delete( $A, x$ )
```

Ścisły zapis pętli **for each ...** w konkretnym języku programowania wymaga uwzględnienia składni tego języka i sprecyzowania mechanizmu przeglądania elementów zbioru. Na przykład w języku Pascal można pierwszą z przytoczonych wyżej pętli, przy deklaracji zbioru  $L$  postaci:

```
L: set of 1..49;
```

zapisać następująco:

```
for  $x:=1$  to 49 do  
  if  $x$  in L then Write( $x:3$ );
```

### 3.7. Słowniki

W wielu zastosowaniach występują zbiory danych, na których są wykonywane jedynie operacje wstawiania elementów do zbioru, usuwania elementów ze zbioru i sprawdzania, czy dany element znajduje się w zbiorze, zaś teoriomnogościowe operacje obliczania sumy, różnicy czy iloczynu zbiorów są niepotrzebne. Strukturę danych umożliwiającą wykonywanie tych trzech operacji podstawowych nazywamy **słownikiem** (ang. *dictionary*). Przykładami słownika są zbiory indeksowe w bazach danych lub komputerowy słownik języka angielskiego.

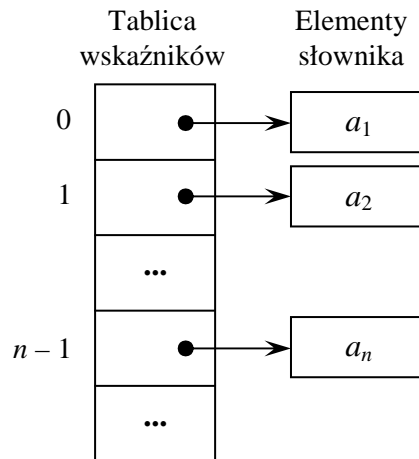
Zazwyczaj do wymienionego zestawu operacji dołącza się jeszcze operację konstrukcji pustego słownika. Dokładniej, przez słownik rozumiemy skończony zbiór danych  $S$ , na którym można wykonywać następujące operacje [3]:



- ♦  $construct(S)$  – inicjalizacja struktury słownikowej:  $S := \emptyset$ ,
- ♦  $insert(S, x)$  – wstawienie elementu do słownika:  $S := S \cup \{x\}$ ,
- ♦  $delete(S, x)$  – usunięcie elementu ze słownika:  $S := S - \{x\}$ ,
- ♦  $search(S, x)$  – sprawdzenie, czy  $x \in S$ , a jeśli tak, to wyznaczenie położenia elementu  $x$  w zbiorze  $S$ <sup>13</sup>.

Czas wykonywania operacji słownikowych jest funkcją mocy zbioru  $S$ , czyli liczby jego elementów. Implementacja tych operacji zależy od implementacji samego zbioru  $S$ . Zazwyczaj elementami zbioru  $S$  są rekordy o pewnej strukturze identyfikowane za pomocą klucza, a wtedy naturalną reprezentacją zbioru jest lista liniowa (uporządkowana lub nie), przechowywana w tablicy lub strukturze dowiązaniowej. Zaletą reprezentacji tablicowej jest prostota zaprogramowania operacji słownikowych, wadami powolne wstawianie i usuwanie elementów (konieczność przesuwania fragmentów tablicy), a w przypadku braku tablic dynamicznych nieefektywne wykorzystanie pamięci (deklarowanie rozmiaru tablic „na wyrost”).

Niewątpliwie interesującym rozwiązaniem jest reprezentacja listy polegająca na umieszczeniu wskaźników do elementów  $a_1, a_2, \dots, a_n$  zbioru  $S$  w tablicy dynamicznej (zob. rys. 3.16). Wygoda obsługi tablicy i możliwość zmiany jej rozmiarów bez utraty zawartości pozostających w niej elementów pozwala na elastyczne wykorzystanie pamięci. W Delphi taką organizację słownika umożliwiają m.in. omówione w tym rozdziale klasy *TList*, *TOrderedList* i *TStringList*.



**Rys. 3.16.** Implementacja słownika za pomocą tablicy wskaźników

<sup>13</sup> Innymi spotykanymi nazwami operacji *construct* i *search* są *mekenull* i *member*.

**Implementacja listowa nieuporządkowana.** Lista nieuporządkowana stanowi najprostszą implementację słownika. Elementy zbioru  $S$  są pamiętane w dowolnej kolejności na liście, która może być przechowywana w tablicy lub strukturze dwiżaniowej. Operacja *construct* polega na utworzeniu pustego słownika, więc wykonuje się w czasie  $O(1)$ . W pozostałych operacjach ma miejsce sekwencyjne przeszukiwanie listy, toteż ich złożoność czasowa wynosi  $O(n)$ . O ile w przypadku operacji *search* jest to oczywiste, to w przypadku operacji *insert* i *delete* przeszukiwanie listy jest wymagane z uwagi na potencjalne próby wstawienia elementu znajdującego się w słowniku lub usunięcia elementu nieistniejącego w słowniku. Nie można bowiem umieścić w słowniku kilku kopii tego samego elementu, ani usunąć elementu, który w nim nie występuje.

Założmy, że wskaźniki do elementów  $a_1, a_2, \dots, a_n$  słownika  $S$  są przechowywane w tablicy dynamicznej  $T[0..n-1]$  (por. rys. 3.16). Przyjmując, że  $n$  i  $T$  są atrybutami słownika  $S$ , możemy wykonywane na nim operacje zaimplementować w pseudojęzyku programowania w następujący sposób:

```

procedure CONSTRUCT( $S$ )
   $n := 0$ ;

function SEARCH( $S, x$ )
   $i := n$ 
  repeat
     $i := i - 1$ 
  until ( $i < 0$ ) or ( $x.key = T[i].key$ );
  return  $i$ 

procedure INSERT( $S, x$ )
  if SEARCH( $S, x$ )  $< 0$  then
     $i := n$ 
     $n := n + 1$ 
    new( $T[i]$ )
     $T[i].key := x$ 

procedure DELETE( $S, x$ )
   $i := SEARCH(S, x)$ 
  if  $i \geq 0$  then
    dispose( $T[i]$ )
    for  $j := i$  to  $n - 2$  do
       $T[j].key := T[j + 1].key$ 
     $n := n - 1$ 

```

Operacja *construct* jedynie zeruje liczbę  $n$  elementów słownika  $S$ . Wynikiem operacji *search* jest indeks wskaźnika do znalezionej elementu  $x$ , bądź  $-1$ , gdy szukany element  $x$  nie występuje w słowniku. Operacje *insert* i *delete* wykorzystują funkcję *search* w celu sprawdzenia, czy możliwe jest wstawienie lub usunięcie elementu. Jeśli tak, pierwsza tworzy nowy element (procedura *new*) o wartości  $x$

i dopisuje jego wskaźnik na końcu listy, a druga usuwa element (procedura *dispose*) wraz z wskaźnikiem i przesuwa dalsze wskaźniki o jedną pozycję w kierunku początku listy. W rzeczywistych implementacjach mogą być przyjęte nieco inne rozwiązania. Na przykład parametrem *search* i *delete* może być klucz elementu  $x$ , a przydział pamięci elementom słownika i zwalnianie jej (procedury *new* i *dispose*) może odbywać się poza operacjami *insert* i *delete*.

Czas działania sekwencyjnych operacji słownikowych jest długi, toteż implementację listową nieuporządkowaną stosuje się, gdy liczba elementów słownika nie jest duża, a także, gdy operacje słownikowe są szczególnie często wykonywane tylko w odniesieniu do niektórych elementów. Wówczas elementy, do których trzeba najczęściej się dostawać, warto umieszczać na końcu listy w reprezentacji tablicowej albo na początku listy w reprezentacji dowiązaniowej.

**Implementacja listowa uporządkowana.** W liście uporządkowanej można łatwo, poprzez porównanie kluczy niektórych elementów, wyeliminować te jej części, w których nie ma wyszukiwanego elementu. Jedną z najczęściej stosowanych metod przeszukiwania listy uporządkowanej jest omówione w podrozdziale 1.6 przeszukiwanie binarne, które polega na podziale obszaru poszukiwania na dwie równej długości części i ograniczeniu dalszego poszukiwania tylko do jednej z nich. Jak już wiadomo, algorytm wykonuje się w czasie  $O(\log n)$ .

W przypadku uporządkowanej rosnąco implementacji listowej słownika za pomocą tablicy wskaźników operacje *search* i *insert* można w pseudojęzyku programowania sprecyzować następująco:

```
function SEARCH(S, x)
  i := 0
  j := n-1
  while i ≤ j do
    k := (i+j) div 2
    if x.key < T[k].key then j := k-1 else
      if x.key > T[k].key then i := k+1 else return k
  return -1

procedure INSERT(S, x)
  i := 0
  j := n-1
  while i ≤ j do
    k := (i+j) div 2
    if x.key < T[k].key then j := k-1 else
      if x.key > T[k].key then i := k+1 else return
  n := n+1
  for k:=n-1 downto i do
    T[k+1] := T[k]
  new(T[i])
  T[i]^ := x
```

Nowy element jest wstawiany nie na końcu listy, lecz w jej odpowiednie miejsce, zgodnie z liniowym uporządkowaniem kluczy. Dlatego kod operacji *insert* wymagał zmodyfikowania. Algorytm wyszukiwania miejsca wstawienia jest taki sam, jak w funkcji *search*, ale nie można jej wykorzystać, ponieważ w przypadku braku wyszukiwanego elementu zwraca ona wartość  $-1$ , a nie indeks miejsca wstawienia nowego elementu. Pozostałe operacje, *construct* i *delete*, nie wymagają zmian. Oczywiście, druga wywołuje zmodyfikowaną funkcję *search*.

Niestety, pomimo że w operacjach *insert* i *delete* zastosowana jest strategia wyszukiwania binarnego, ich pesymistyczna złożoność czasowa nie poprawiła się i nadal wynosi  $O(n)$ . Jest tak dlatego, że wstawienie elementu do słownika lub jego usunięcie może wymagać przesunięcia wskaźników w tablicy  $T$ .

Istnieją inne, bardziej wymyślne, uporządkowane struktury danych, umożliwiające wykonywanie operacji słownikowych w czasie  $O(\log n)$ .<sup>14</sup> Cechuje je wyższy stopień skomplikowania, a ich efektywność jest zauważalna dopiero w przypadku słowników o dużych rozmiarach.

### 3.8. Mieszanie (haszowanie)

Kluczami w wielu słownikach są ciągi znaków. Oznacza to, że liczba wszystkich wartości klucza może być ogromna, podczas gdy w porównaniu z nią liczba przechowywanych w słowniku elementów mała. Niezwykle efektywną w takich przypadkach metodą składowania i wyszukiwania informacji wewnątrz komputera, prowadzącą do znacznych oszczędności pamięci i czasu, jest **mieszanie**, zwane również **rozpraszaniem**, **siekaniem** lub **haszowaniem** (ang. *hashing*)<sup>15</sup>. Metoda polega na obliczaniu indeksu elementu w tablicy wskaźników za pomocą funkcji, której parametrem jest klucz tego elementu. Nazywamy ją **funkcją mieszającą**, **rozpraszającą**, **siekającą** lub **haszującą** (ang. *hash function*).

Funkcja mieszająca ma na celu zmniejszenie dużej liczby wszystkich potrzebnych pozycji w tablicy słownika do rozsądnego rozmiaru  $m$ . Element o kluczu  $k$  zostaje umieszczony w tablicy na pozycji o indeksie  $h(k)$  obliczonym za pomocą funkcji mieszającej  $h$ , która odwzorowuje zbiór  $U$  (uniwersum) wszystkich możliwych kluczy w zbiór indeksów tablicy  $T[0..m-1]$ :

$$h : U \rightarrow \{0, 1, \dots, m-1\}$$

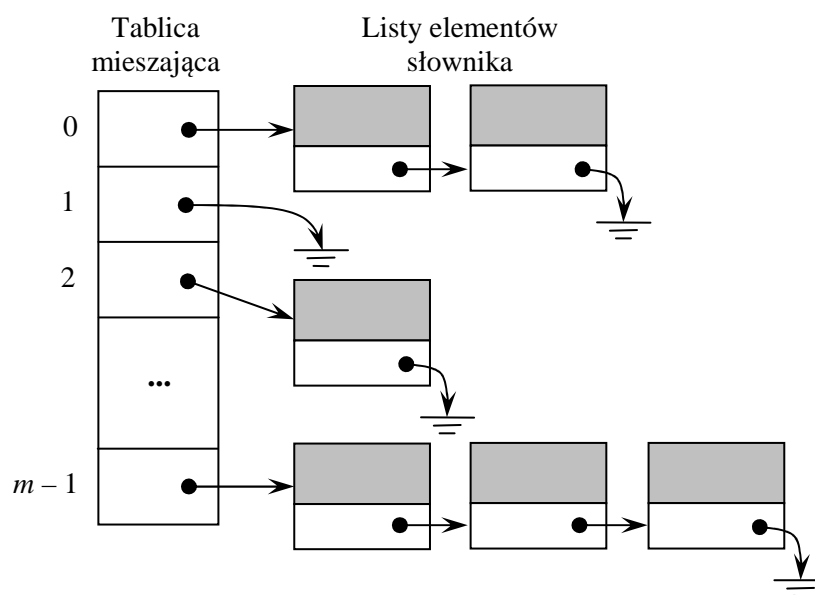
---

<sup>14</sup> Chodzi o struktury drzewiaste, m.in. drzewa poszukiwań binarnych – BST (skrót od ang. *binary search trees*) i drzewa wyważone – AVL (skrót złożony z pierwszych liter nazwisk wynalazców) [1, 3, 7, 8, 17, 22, 28].

<sup>15</sup> Idea haszowania zrodziła się i została urzeczywistniona w 1953 roku w IBM [17].

Obliczanie wartości funkcji mieszającej polega na wykonaniu pewnych operacji arytmetycznych na wartości klucza określonej w parametrze, bez porównywania jej z wartościami klucza innych elementów słownika, więc odbywa się w czasie stałym. Sugeruje to, że odnajdywanie elementów w słowniku z wykorzystaniem funkcji mieszającej, a także wstawianie elementów do słownika i usuwanie ich, wykonuje się w czasie  $O(1)$ . Niestety, funkcja mieszająca nie jest różnowartościowa, gdyż odwzorowuje ogromny zbiór  $U$  wszystkich kluczy w nieporównywalnie mniejszy zbiór indeksów tablicy  $T$ . Zatem może przypisać dwu lub większej liczbie różnych kluczy tę samą pozycję w tablicy  $T$ . Sytuację taką nazywamy **kolizją**.

Głównym problemem w implementacji słownika za pomocą tablicy mieszającej jest rozwiązywanie konfliktów wywołanych przez kolizje. Istnieje wiele efektywnych metod usuwania kolizji [1, 2, 3, 7, 8, 17, 28]. Jedną z najpopularniejszych opiera się na pomysłe wykorzystania wielu oddzielnych list. Wszystkie elementy o kluczach  $k$ , którym funkcja mieszająca  $h$  przypisuje ten sam indeks  $h(k)$  pomiędzy wartościami 0 i  $m - 1$ , są umieszczane na jednej liście, której początek określa element  $T[h(k)]$  (por. rys. 3.17).



**Rys. 3.17.** Implementacja słownika za pomocą tablicy mieszającej

Skoro nie można uniknąć kolizji, to przynajmniej należy zminimalizować ich liczbę wystąpień. Osiąga się to poprzez odpowiedni dobór funkcji mieszającej i rozmiaru  $m$  tablicy  $T$  w stosunku do możliwego rozmiaru  $n$  słownika  $S$ . Funkcja mieszająca powinna spełniać dwa warunki:

- ♦ obliczanie jej wartości powinno być bardzo szybkie,
- ♦ powinna równomiernie odwzorowywać wartości kluczy w zbiór indeksów.

Zaleca się, aby wartościami  $m$  były liczby pierwsze niezbyt bliskie potęgom 2. Jest oczywiste, że średnia liczba elementów przypadających na jedną listę wynosi  $n/m$ . Jeżeli  $m \geq n$ , przeciętna długość listy jest nie większa od 1. Przy wystarczająco dobrej funkcji mieszającej poszczególne listy są krótkie (niektóre są puste), przez co operacje słownikowe wykonują się szybko i niezależnie od tego, jak duże są wartości  $m$  i  $n$ . Zatem ich oczekiwana złożoność czasowa wynosi  $O(1)$ . Jeżeli  $m$  jest mniejsze od  $n$  niewiele razy, to i tak średnia długość list jest mała, np. dla  $n = 2000$  i  $m = 701$  wynosi 3.

Wybór funkcji mieszającej jest bardzo istotny. Gdy jest dobrana niewłaściwie, w skrajnym przypadku może powstać tylko jedna lista, która obejmuje wszystkie elementy słownika, a wtedy operacje słownikowe wykonują się w czasie  $O(n)$ . Dlatego warto sprawdzić, przynajmniej doświadczalnie, czy funkcja mieszająca równomiernie „rozrzuca” klucze w zbiór indeksów.

Rozważmy przykład słownika słów o długości do 20 liter alfabetu angielskiego. Załóżmy, że ma on zawierać około 500 pozycji:  $n = 500$ . Do zapamiętania słownika potrzebny jest więc obszar pamięci o rozmiarze  $21 \times 500 = 10500$  bajtów. Alfabet angielski liczy 26 liter, więc wszystkich możliwych słów jest  $26^{20}$ . Jest to liczba tak ogromna, że zapewne przez wiele lat zarezerwowanie tablicy na zapamiętanie ich wszystkich będzie niemożliwe w żadnych systemach i środowiskach programowania. Tymczasem implementację słownika można utworzyć już w Turbo Pascalu, przyjmując  $m = 503$  i wykorzystując poniższą funkcję mieszającą:

```
function h(var key: string): integer;
const
  m = 503;
var
  i, s: integer;
begin
  s := 0;
  for i:=1 to Length(key) do
    s := s + ord(UpCase(key[i]));
  h := s mod m;
end;
```

Wartością funkcji  $h$  jest reszta z dzielenia sumy kodów liter przez  $m$ , po zamianie wszystkich małych liter na duże. Nie jest to funkcja doskonała, ponieważ dwu słowom złożonym z tych samych liter, np. *kot* i *tok*, przypisuje jeden indeks, ale jest użyteczna w praktyce. Można ją łatwo poprawić, przemnażając w każdym kroku pętli **for** dotychczas obliczoną wartość  $s$  przez stały czynnik (por. ćw. 3.19). Stwierdzono doświadczalnie [13], że dobrymi czynnikami są liczby 31 i 37.

Implementacje słowników opartych na technice mieszania można tworzyć w Delphi, wyprowadzając klasy potomne z omówionych w niniejszym rozdziale klas *TList*, *TOrderedList* i *TStringList*. W Delphi 6 wprowadzono nowe klasy *TBucketList* i *TObjectBucketList* (lista dowolnych danych i lista obiektów), które obsługują wewnętrzną tablicę mieszającą złożoną z elementów, zwanych porcjami (ang. *buckets*), będących wskaźnikami do podlist. Elementami podlist są klucze i powiązane z nimi dane. Obie klasy zawierają m.in. metody *Add* (wstawienie klucza i danych do struktury), *Find* (wyszukanie danych o podanym kluczu), *Remove* (usuwanie klucza i danych), *Exists* (sprawdzenie, czy istnieje element o podanym kluczu) oraz *ForEach* (wykonanie danej procedury na każdym elemencie listy). Wbudowana w klasach funkcja mieszająca jest nienajlepsza [6], lecz można ją zastąpić własną, pokrywając funkcję wirtualną *BucketFor*.

## Ćwiczenia

**3.1.** Napisz program, który umożliwia wstawienie, usuwanie i przeszukiwanie uporządkowanej listy liczb całkowitych lub łańcuchów. Opracuj wersję tablicową i dowiązaniową.

**3.2.** Napisz program łączący dwie uporządkowane listy liczb całkowitych lub łańcuchów. Opracuj wersję tablicową i dowiązaniową.

**3.3.** Zaprojektuj tablicową strukturę danych reprezentującą licznik i umożliwiającą zerowanie licznika, zwiększanie go o jeden i przełączanie między systemem dwójkowym, dziesiętnym i szesnastkowym. Zbuduj aplikację w Delphi ilustrującą wykorzystanie licznika.

**3.4.** Implementacją wielomianu postaci:

$$P_n(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n$$

o współczynnikach rzeczywistych  $a_0, a_1, \dots, a_n$  może być lista dowiązaniowa tych współczynników. Napisz podprogram obliczający wartość reprezentowanego w ten sposób wielomianu.

**3.5.** Opracuj algorytmy różniczkowania, sumowania i mnożenia wielomianów implementowanych za pomocą listy dowiązaniowej (por. ćw. 3.4) i zapisz je w postaci programów komputerowych.

**3.6.** Zbuduj aplikację konsolową w Delphi, która odwraca kolejność wierszy pliku tekstowego, wykorzystując strukturę stosu.

**3.7.** Zmodyfikuj aplikację generowania skorowidza słów pliku tekstowego tak, by zamiast znajdowania numerów wierszy zliczała liczby wystąpień słów.

**3.8.** Opracuj cykliczną implementację kolejki pojedynczej, w której nie zapamiętuje się rozmiaru kolejki. Uwzględnij w niej operację wyznaczania rozmiaru kolejki i określ złożoność czasową tej operacji.

**3.9. Rozpraszanie** ciągu elementów  $a_1, a_2, \dots, a_n$ , zwane też **randomizacją**, różni się tym od sortowania przez selekcję, że w  $i$ -tym kroku ( $i = 1, 2, \dots, n - 1$ ) nie wybiera się elementu najmniejszego spośród  $a_i, a_{i+1}, \dots, a_n$ , lecz losowo element z całego ciągu [22]. Zbuduj aplikację w Delphi symulującą losowanie zakładów totolotka za pomocą struktury tablicowej rozpraszającej numery od 1 do 49.

**3.10.** Opracuj algorytm, który generuje nieuporządkowaną talię kart do gry w brydża. Zbuduj aplikację w Delphi, która wykorzystuje ten algorytm do symulacji rozdania kart pomiędzy czterech graczy.

**Wskazówka.** Użyj kolejki podwójnej i czterech kolejek priorytetowych.

**3.11.** Zbuduj aplikację w Delphi symulującą rozdanie kart do gry w brydża, wykorzystującą strukturę rozpraszającą do zaimplementowania nieuporządkowanej talii kart (por. ćw. 3.9).

**3.12.** Dany jest stos liczb całkowitych. Posługując się drugim stosem, wstępnie pustym, usuń z pierwszego stosu wszystkie liczby parzyste. Zbuduj aplikację w Delphi wykonującą tę operację.

**3.13.** Wyrażenie arytmetyczne zapisane w powszechnie stosowanej postaci nawiasowej, zwanej **notacją wrostkową** lub **infiksową**, daje się przedstawić bez użycia nawiasów w tzw. **odwrotnej notacji polskiej** (ONP)<sup>16</sup>, zwanej również **przyrostkową** lub **postfiksową** [1, 21, 24, 28, 30]. W notacji tej operatory zawsze występują po swoich argumentach, przez co nawiasy są zbędne. Na przykład wyrażenie:

---

<sup>16</sup> Odwrotną notację polską wynalazł polski logik Jan Łukasiewicz (1878 – 1956).



$$(3 * x - 5) / (2 * x)$$

zapisane w ONP ma postać:

$$3x*5-2x*/$$

Wyrażenia w ONP są mniej czytelne niż tradycyjne, ale są bardzo przydatne przy pisaniu kompilatorów i interpretatorów. Przy przekładzie wyrażenia złożonego ze stałych, zmiennych, nawiasów i operatorów dwuargumentowych +, -, \*, / na ONP należy uwzględnić następującą listę priorytetów<sup>17</sup>:

Ogranicznik	Priorytet
(	0
+, -, )	1
*, /	2

Algorytm konwersji, bazujący na strukturze stosu i analizujący wyrażenie od strony lewej do prawej, działa według następującego schematu:

- K1.** Pobierz kolejny element  $E$  źródłowego wyrażenia arytmetycznego.
- K2.** Jeżeli  $E$  jest stałą lub zmienną, przekaz ją na wyjście.
- K3.** Jeżeli  $E$  jest lewym nawiasem, wstaw go na stos.
- K4.** Jeżeli  $E$  jest operatorem, zdejmij ze stosu i przekaz na wyjście wszystkie operatory o priorytecie wyższym lub równym priorytetowi  $E$  (o ile istnieją), po czym wstaw  $E$  na stos.
- K5.** Jeżeli  $E$  jest prawym nawiasem, zdejmij ze stosu i przekaz na wyjście wszystkie operatory aż do napotkania lewego nawiasu, który zdejmij ze stosu bez przekazania na wyjście.
- K6.** Jeżeli wyrażenie źródłowe nie zostało wyczerpane, wróć do punktu K1.
- K7.** Zdejmij ze stosu pozostałe operatory i przekaz je na wyjście.

Zbuduj aplikację w Delphi przekształcającą wyrażenie arytmetyczne (łańcuch) z postaci nawiasowej na ONP. Dla uproszczenia przyjmij, że stałe są reprezentowane za pomocą cyfr i ew. kropki, zaś zmienne za pomocą pojedynczych liter. Uwzględnij kontrolę poprawności zapisu wyrażenia źródłowego.

<sup>17</sup> Dla uproszczenia przyjęliśmy, że wyrażenie nie zawiera jednoargumentowych operatorów + i -. Rozpoznanie ich wymaga wstępnej obróbki wyrażenia, podczas której pierwszy jest usuwany, a drugi zastępowany umownym ogranicznikiem o priorytecie 2 (grupa operatorów multiplikatywnych) [24]. Z kolei uwzględnienie potęgowania, które można oznaczyć np. symbolem ^, wymaga rozszerzenia listy ograniczników o operator potęgowania o priorytecie 3.

**3.14.** Dlaczego w omówionej w podrozdziale 3.5 implementacji kolejki priorytetowej za pomocą klasy *Theap* nie uwzględniono występującej w pseudojęzyku programowania sygnalizacji błędu?

**3.15.** Utwórz w pseudojęzyku programowania implementacje listowe operacji *insert*, *delete* i *member* na zbiorach oraz prostej operacji *makenull* tworzącej zbiór pusty, a następnie wykorzystaj je do zaimplementowania operacji sumy, różnicy, różnicy symetrycznej i iloczynu dwóch zbiorów.

**3.16.** Zmodyfikuj podane w podrozdziale 3.7 implementacje listowe słownika tak, by przydzielanie pamięci jego elementom i zwalnianie jej nie odbywało się w procedurach *insert* i *delete*.

**3.17.** Zmodyfikuj w implementacji listowej uporządkowanej słownika funkcję *search* tak, by w przypadku braku wyszukiwanego elementu wyznaczała miejsce, które by zajmował, gdyby znajdował się w słowniku. Następnie wykorzystaj tę funkcję w procedurze *insert*.

**3.18.** Zaprojektuj i zaimplementuj w Delphi strukturę słownikową umożliwiającą wyszukiwanie podstawowych informacji o różnych krajach świata (nazwa, kontynent, powierzchnia, liczba ludności, stolica, ustrój itp.). Użytkownik słownika powinien w trakcie wpisywania nazwy kraju uzyskiwać odpowiedź w postaci listy nazw krajów dostosowującej swoją zawartość do wpisywanego tekstu.

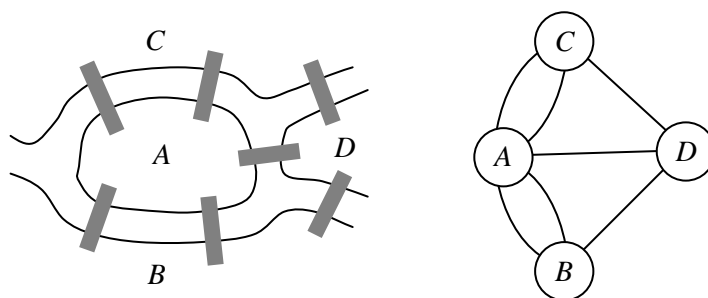
**3.19.** Jeden z najczęściej stosowanych algorytmów obliczania indeksu łańcucha w tablicy mieszającej polega na dodawaniu kodu kolejnego znaku do wielokrotności sumy dotychczas obliczonej. Napisz w języku Pascal funkcję mieszającą, która działa według tego algorytmu. Przyjmij, że rozmiar tablicy wynosi 701, a czynnikiem zwielokrotniającym sumę jest liczba 37.

**3.20.** Zaprojektuj i zaimplementuj w Delphi słownik haseł informatycznych. Opracuj wersje z wyszukiwaniem sekwencyjnym, binarnym i mieszającym.

## Rozdział 4.

### Algorytmy grafowe

Wiele problemów technicznych i naukowych jest rozwiązywanych za pomocą grafów. Grafom poświęcony jest osobny dział matematyki. Grafy mają zastosowanie w różnych działach informatyki, np. w złożoności obliczeniowej, grafice komputerowej, metodach translacji, sieciach komputerowych. Rozwój teorii grafów datuje się od roku 1736, kiedy to L. Euler rozwiązał słynny problem siedmiu mostów Królewca, rozstrzygając, że nie można przejść każdego z nich tylko raz i wrócić do punktu wyjścia (rys. 4.1). Szczególne rodzaje grafów, a właściwie drzew, pojawiły się już w tej książce w podrozdziałach 2.5 i 3.5. Były to kopce. Podobnie jak kopce, drzewa i grafy są strukturami składającymi się z wierzchołków połączonych krawędziami.



**Rys. 4.1.** Mosty w Królewcu i ich model grafowy

Niniejszy rozdział nie stanowi pełnego przeglądu najważniejszych problemów grafowych i algorytmów ich rozwiązywania. Po przedstawieniu podstawowych pojęć i sposobów implementowania grafów skoncentrujemy się na wybranych algorytmach grafowych. Takie wybiórcze ujęcie problematyki grafowej ma na celu zaprezentowanie minimalnej wiedzy dotyczącej grafów i algorytmicznej strony omawianych zagadnień. Skoncentrujemy się na następujących problemach przetwarzania grafów:

- ◆ przeszukiwanie grafu (systematyczne przechodzenie grafu wzdłuż jego krawędzi w celu odwiedzenia wszystkich wierzchołków),
- ◆ znajdowanie drzewa rozpinającego grafu,
- ◆ znajdowanie spójnych składowych grafu,
- ◆ znajdowanie najkrótszych ścieżek (dróg) w grafie.

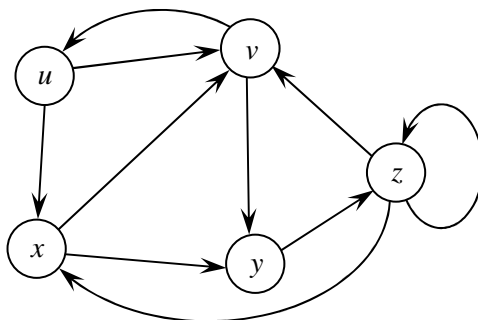
Czytelnik zainteresowany głębiej tematyką grafów może znaleźć wiele pozycji książkowych na rynku wydawniczym. Na uwagę zasługują pozycje o charakterze matematycznym [27], matematyczno-informatycznym [21] i informatycznym [7]. Cennym źródłem informacji są również pozycje [1, 2, 3, 8, 18, 19, 22, 30]. Poruszają one m.in. następujące zagadnienia pominięte w tej książce:

- ♦ znajdowanie drogi o minimalnym koszcie (problem komiwojażera),
- ♦ znajdowanie cykli Eulera (dróg zamkniętych, w których każda krawędź występuje tylko raz),
- ♦ kolorowanie grafów planarnych (graf jest planarny, gdy daje się przedstawić na płaszczyźnie bez przecinania krawędzi),
- ♦ problem maksymalnego przepływu w sieciach przepływowych.

## 4.1. Podstawowe pojęcia

**Grafem** nazywamy strukturę  $G = (V, E)$  złożoną z niepustego zbioru **wierzchołków**  $V$ , zwanych także **węzłami**, oraz zbioru **krawędzi**  $E$ , zwanych inaczej **łukami**. Rozróżnia się grafy **skierowane** (ang. *directed graph*), zwane też grafami **zorientowanymi** lub krócej **digrafami**, i grafy **nieskierowane** (ang. *undirected graph*), zwane również grafami **niezorientowanymi**.

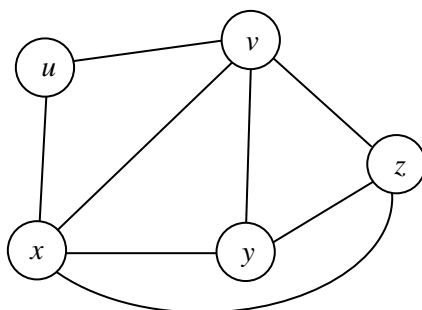
W grafie skierowanym krawędzie można opisać jako uporządkowane pary wierzchołków  $(u, v)$ . Wierzchołek  $u$  nazywamy **początkiem** krawędzi, a  $v$  **końcem**. Mówimy, że krawędź  $(u, v)$  biegnie od wierzchołka  $u$  do wierzchołka  $v$ , a także, że wierzchołki  $u$  i  $v$  są **sąsiednimi**, **sąsiadującymi** lub **incydentnymi**. Krawędź, która rozpoczyna się i kończy w tym samym wierzchołku, nazywamy **pętlą**. Krawędź  $(u, v)$  jest często zapisywana jako  $u \rightarrow v$  i rysowana w postaci zakończonego strzałką odcinka lub łuku łączącego oba wierzchołki (rys. 4.2).



Rys. 4.2. Przykładowy graf skierowany

**Ścieżką** lub **drogą** w grafie skierowanym nazywamy sekwencję krawędzi taką, że koniec jednej krawędzi jest początkiem następnej. **Długością ścieżki** nazywamy liczbę należących do niej krawędzi. Ciąg wierzchołków  $v_1, v_2, \dots, v_k$  grafu takich, że dla każdego  $i = 1, 2, \dots, k-1$  istnieje w tym grafie krawędź  $(v_i, v_{i+1})$ , określa ścieżkę o długości  $k-1$ . Ścieżka ta rozpoczyna się w wierzchołku  $v_1$ , przechodzi przez wierzchołki  $v_2, v_3, \dots, v_{k-1}$  i kończy w wierzchołku  $v_k$ . Ścieżkę nazywamy **prostą**, jeżeli jej wszystkie krawędzie są różne, tj. gdy każdy wierzchołek, z wyjątkiem być może pierwszego i ostatniego, jest różny od pozostałych. Ścieżkę nazywamy **zamkniętą**, jeżeli  $v_1 = v_k$ , tj. gdy rozpoczyna się i kończy w tym samym wierzchołku. **Cyklem** nazywamy ścieżkę zamkniętą o długości co najmniej 1. Cykl o długości 1 tworzy jedna krawędź, czyli pętla. Cykl nazywamy **prostym**, jeżeli jego wszystkie wierzchołki są parami różne, oprócz ostatniego, który jest równy pierwszemu. Graf nazywamy **acyklicznym**, jeżeli nie zawiera cykli. W grafie acyklicznym wszystkie ścieżki nie są zamknięte.

W grafie nieskierowanym kolejność wierzchołków połączonych krawędzią jest nieistotna. Krawędź łączącą wierzchołki  $u$  i  $v$  można oznaczać zarówno przez  $(u, v)$ , jak i przez  $(v, u)$ , gdyż  $(u, v) = (v, u)$ . Spotyka się również oznaczenia  $\{u, v\}$  i  $u - v$ . Graficznie krawędzie grafu nieskierowanego przedstawia się jako odcinki lub łuki łączące wierzchołki, ale bez strzałek (rys. 4.3).

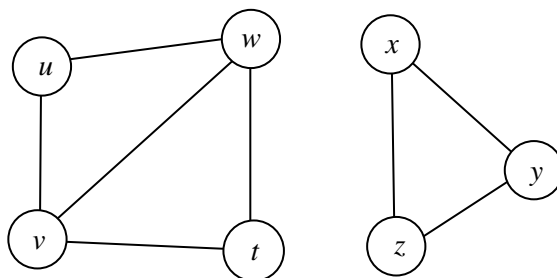


Rys. 4.3. Przykładowy graf nieskierowany

Większość pojęć zdefiniowanych dla grafów skierowanych przenosi się na grafy nieskierowane. I tak, wierzchołki  $u$  i  $v$  nazywamy **sąsiednimi**, gdy istnieje krawędź, która je łączy, a **ścieżkę** lub **drogę** definiujemy jako ciąg krawędzi, które łączą się ze sobą. Kolejne dwie krawędzie w ścieżce muszą mieć wspólny wierzchołek, a zatem ścieżkę wyznacza również ciąg wierzchołków. Podobnie określa się **długość ścieżki** – jako liczbę jej krawędzi. Ścieżkę łączącą ten sam wierzchołek nazywamy **zamkniętą**, a ścieżkę, w której wszystkie krawędzie są różne, **prostą**. Również **cyklem** w grafie nieskierowanym nazywamy ścieżkę zamkniętą o długo-

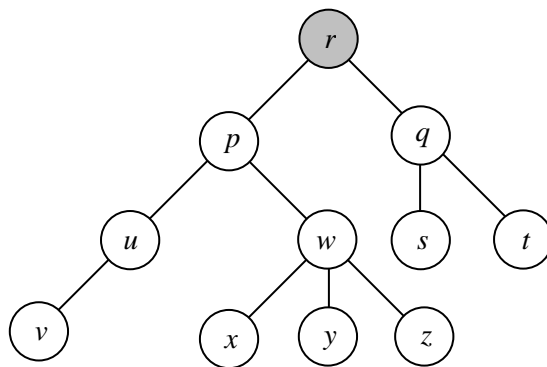
ści co najmniej 1, cyklem **prostym** ścieżkę prostą zamkniętą, a graf, w którym nie ma cykli, **acyklicznym**.

**Podgrafem** grafu  $G = (V, E)$  nazywamy dowolny graf  $G' = (V', E')$  taki, że  $V' \subseteq V$  i  $E' \subseteq E$ . Graf nieskierowany nazywamy **spójnym**, jeżeli dla każdej pary wierzchołków istnieje łącząca je ścieżka. Mówiąc bardziej obrazowo, graf taki składa się z jednego „kawałka”, a niespójny z wielu „kawałków” (rys. 4.4). **Spójną składową** grafu nieskierowanego nazywamy jego największy, w sensie zawierania zbiorów wierzchołków i krawędzi, spójny podgraf<sup>1</sup>.



Rys. 4.4. Przykład grafu niespójnego

**Drzewem** nazywamy graf nieskierowany, który jest spójny i acykliczny. Jeżeli do drzewa dodamy nową krawędź łączącą dwa jego wierzchołki, otrzymamy graf, który nie jest drzewem, gdyż będzie zawierał cykl [21]. **Drzewem z korzeniem** nazywamy drzewo, w którym jeden wierzchołek, zwany **korzeniem** (ang. *root*), jest wyróżniony (szary kolor na rys. 4.5).



Rys. 4.5. Drzewo z korzeniem

<sup>1</sup> Odpowiednikiem spójności w grafach skierowanych jest tzw. silna spójność [1, 3, 7, 8] i słaba spójność [8].

Pomimo że drzewo z korzeniem jest grafem nieskierowanym, można w nim określić następstwo wierzchołków. Mianowicie, jeżeli  $u$  i  $v$  są dowolnymi dwoma wierzchołkami drzewa, istnieje dokładnie jedna łącząca je ścieżka. Jeżeli sprowadza się ona do krawędzi  $\{u, v\}$  i  $r$  jest korzeniem drzewa, to wierzchołek  $u$  znajduje się na ścieżce od  $r$  do  $v$  (por. rys. 4.5), albo wierzchołek  $v$  znajduje się na ścieżce od  $r$  do  $u$ . W pierwszym przypadku mówimy, że  $u$  jest **poprzednikiem** lub **rodzicem**  $v$ , a  $v$  **następnikiem** lub **dzieckiem**  $u$ , zaś w drugim, że  $v$  jest poprzednikiem (rodzicem)  $u$ , a  $u$  następnikiem (dzieckiem)  $v$ .

Każdy wierzchołek, oprócz korzenia, ma jednego poprzednika. Poprzednik może mieć więcej niż jednego następnika. Wierzchołek, który ma następnika, nazywamy wierzchołkiem **wewnętrznym**, a taki, który nie ma następnika, nazywamy **liściem**. Mówimy też, że  $v$  jest **potomkiem** wierzchołka  $u$ , jeżeli  $v \neq u$  i  $u$  jest wierzchołkiem ścieżki łączącej korzeń  $r$  z wierzchołkiem  $v$ . Podobnie mówimy, że  $u$  jest **przodkiem** wierzchołka  $v$ , jeżeli  $u \neq v$  i  $u$  jest wierzchołkiem ścieżki łączącej korzeń  $r$  z wierzchołkiem  $v$ . Korzeń jest przodkiem wszystkim różnym od niego wierzchołków drzewa. **Wysokością wierzchołka** nazywamy maksymalną długość ścieżki od tego wierzchołka do liścia. **Wysokością drzewa** nazywamy wysokość jego korzenia. **Głębokością** lub **numerem poziomym** wierzchołka nazywamy długość ścieżki łączącej go z korzeniem. Korzeń ma więc głębokość zero.

Szczególnym przypadkiem drzewa z korzeniem jest **drzewo binarne**, w którym każdy wierzchołek ma co najwyżej dwóch następników. Zazwyczaj określa się ich jako **lewy** i **prawy**. Przypomnijmy, że drzewami binarnymi są kopce, z którymi mieliśmy do czynienia przy sortowaniu kopcowym i kolejkach priorytetowych.

Wypada nadmienić, że w teorii grafów istnieje jeszcze wiele innych definicji i pojęć, a ponadto spotyka się drobne różnice zarówno w definicjach, jak i terminologii. Na przykład niektórzy autorzy nazywają grafem taki graf, który nie ma pętli, określając graf z pętlami mianem **multigrafu**.

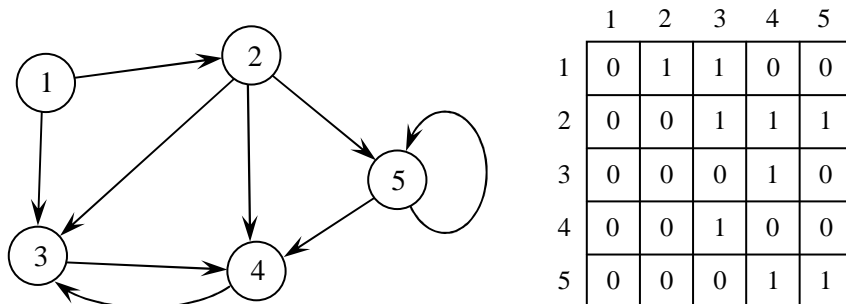
## 4.2. Sposoby reprezentowania grafów

Liczbę krawędzi grafu  $G = (V, E)$ , skierowanego lub nie, najczęściej oznacza się przez  $m$ , a liczbę wierzchołków przez  $n$ . Suma obu liczb stanowi tzw. **rozmiar grafu**  $G$ . Wierzchołki grafu najwygodniej jest ponumerować kolejnymi liczbami całkowitymi od 1 do  $n$ . Numeracja ta ma na celu jedynie łatwą identyfikację wierzchołków, a nie przypisywanie im jakiegokolwiek kolejności. Tak więc, bez szkody dla ogólności rozważań możemy założyć, że  $V = \{1, 2, \dots, n\}$ .

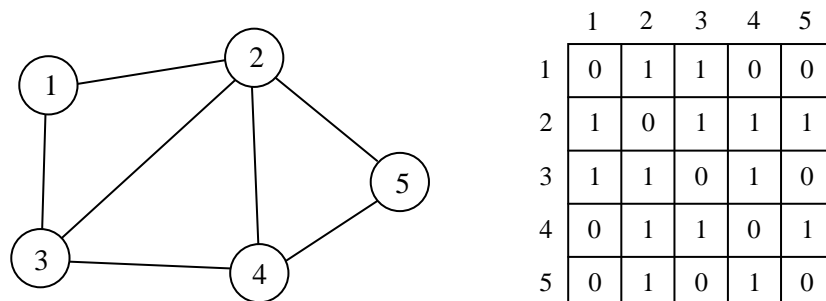
Istnieją dwa główne sposoby reprezentowania grafów w pamięci komputera:

- ◆ macierz sąsiedztwa,
- ◆ lista sąsiedztwa.

**Macierz sąsiedztwa** jest macierzą kwadratową  $A$  o rozmiarze  $n \times n$  złożoną z elementów typu *boolean* taką, że element  $A[u, v]$  ma wartość *true* wtedy, gdy w grafie istnieje krawędź prowadząca od wierzchołka  $u$  do wierzchołka  $v$ , a wartość *false*, gdy takiej krawędzi nie ma. Często wartości elementów macierzy  $A$  zapisuje się jako liczby 0 i 1, które odpowiadają wartościom logicznym *false* i *true* (rys. 4.6). Nietrudno zauważyć, że macierz sąsiedztwa grafu nieskierowanego jest symetryczna, co można wykorzystać do redukcji zajmowanej pamięci o połowę.



a) graf skierowany i jego macierz sąsiedztwa

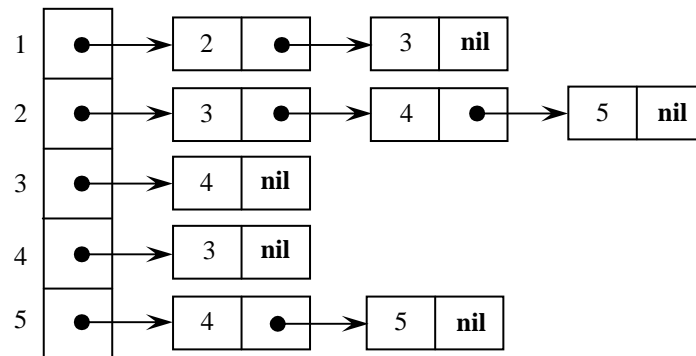


b) graf nieskierowany i jego macierz sąsiedztwa

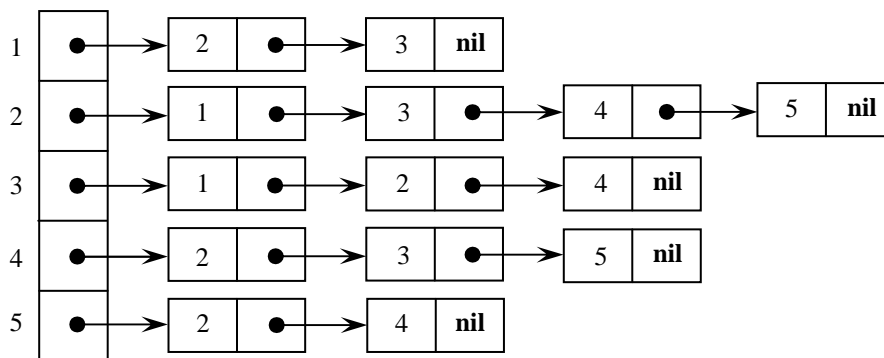
**Rys. 4.6.** Reprezentacja macierzowa grafu

**Lista sąsiedztwa** polega na przypisaniu każdemu wierzchołkowi grafu listy sąsiadujących z nim wierzchołków. Kolejność wierzchołków na liście związanej z danym wierzchołkiem może być dowolna. Cały zestaw list wygodnie jest zaimplementować w postaci tablicy wskaźników, której element o indeksie  $u$  wskazuje na początek listy  $L[u]$  wierzchołków sąsiadujących z wierzchołkiem  $u$  (por. rys. 4.7). Mówiąc dokładniej,  $v \in L[u]$  wtedy i tylko wtedy, gdy istnieje krawędź wiodąca od wierzchołka  $u$  do wierzchołka  $v$ .





a) lista sąsiedztwa grafu skierowanego z rys. 4.6a



b) lista sąsiedztwa grafu nieskierowanego z rys. 4.6b

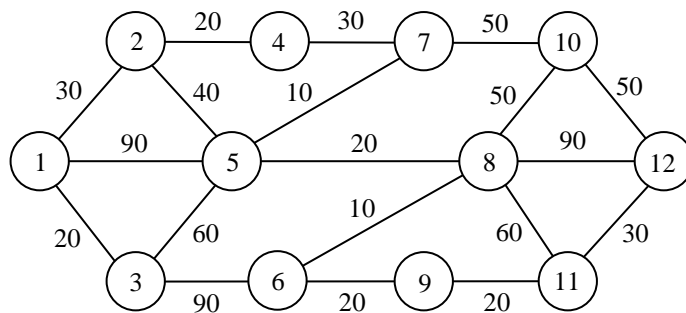
**Rys. 4.7.** Reprezentacja grafu w postaci listy sąsiedztwa

Reprezentacja macierzowa grafu jest wygodna w obliczeniach. Zwłaszcza łatwo jest sprawdzić, czy pomiędzy dwoma dowolnymi wierzchołkami grafu istnieje krawędź. Również czas dostępu do elementów macierzy jest stały, niezależny od liczby wierzchołków i krawędzi. Podstawową wadą macierzy sąsiedztwa jest wysoka złożoność pamięciowa oraz niedogodność reprezentowania grafu o zmiennej liczbie wierzchołków. Zapis grafu o  $n$  wierzchołkach wymaga  $n^2$  komórek pamięci i jest szczególnie niekorzystny w przypadku **grafów rzadkich**, w których liczba krawędzi  $m$  jest mała w porównaniu z wartością  $n^2$ . Jednak elementy macierzy można pamiętać po 8 w bajcie, co pozwala istotnie zredukować zapotrzebowanie na pamięć, ale nieco utrudnia dostęp do nich.

Zapotrzebowanie na pamięć list sąsiedztwa jest proporcjonalne do sumy liczby wierzchołków i liczby krawędzi, wynosi więc  $O(m+n)$ . Jednak sprawdzenie, czy istnieje krawędź  $(u, v)$ , wymaga przeglądania listy sąsiedztwa wierzchołka  $u$

w poszukiwaniu wierzchołka  $v$ , toteż jego pesymistyczna złożoność czasowa wynosi  $O(n)$ . W przypadku macierzy sąsiedztwa takie sprawdzenie wykonuje się natychmiastowo, tj. w czasie  $O(1)$ , gdyż wymaga jedynie dostępu do wartości  $A[u, v]$ .

Niekiedy krawędziom grafu przypisuje się pewne wartości, zwane **wagami** lub **etykietami**, a wtedy graf nazywamy **ważonym** lub **etykietowanym**. Rysunek 4.8 pokazuje przykładowy graf ważony nieskierowany<sup>2</sup>. Przykładem rzeczywistego grafu ważonego jest graf przedstawiający sieć połączeń drogowych lub lotniczych, w którym wierzchołkami są miejscowości, a wagami odległości między nimi, czas przelotu lub koszt podróży.



Rys. 4.8. Przykładowy graf ważony nieskierowany

Wartościami elementów macierzy sąsiedztwa grafu ważonego mogą być, zamiast *false* i *true* lub 0 i 1, wagi (etykiety). Listy sąsiedztwa można dostosować do grafów ważonych, dodając do elementów listy pole z wagą. Reprezentacja macierzowa jest pod tym względem korzystniejsza, gdyż wagi można przechowywać bezpośrednio w elementach macierzy.

### 4.3. Przeszukiwanie grafu w głąb

W wielu algorytmach grafowych wymagane jest systematyczne przechodzenie grafu wzdłuż jego krawędzi w celu odwiedzenia wszystkich wierzchołków. Jedną z najprostszych metod takiego przechodzenia jest **przeszukiwanie w głąb** (ang. *depth first search*). Metoda polega na przypisaniu każdemu wierzchołkowi statusu **nieodwiedzony**, wybraniu pewnego wierzchołka  $u \in V$  jako startowego, nadaniu mu statusu **odwiedzony**, a następnie na rekurencyjnym zastosowaniu tej metody do wszystkich następników wierzchołka  $u$ , czyli do tych wierzchołków  $v$ , dla których

<sup>2</sup> Na podstawie książki [27].

istnieje krawędź wiodąca od  $u$  do  $v$ . Jeżeli wszystkie wierzchołki grafu zostaną w ten sposób odwiedzone, przeszukiwanie zostaje zakończone, a jeżeli w grafie pozostaną wierzchołki nieodwiedzone, wybiera się którykolwiek z nich jako startowy i powtarza od niego przeszukiwanie. Postępowanie kontynuuje się dopóty, dopóki wszystkie wierzchołki grafu nie zostaną odwiedzone.

Do reprezentowania zbioru następników wierzchołka  $u \in V$  wygodnie jest użyć listy sąsiedztwa  $L[u]$ , a do określenia statusu wierzchołków – tablicy o elementach typu *boolean* (*false* – nieodwiedzony, *true* – odwiedzony), którą nazwiemy *visited*. Przy założeniu, że początkowo wszystkie elementy tablicy *visited* mają wartość *false*, czyli że wszystkie wierzchołki grafu są nieodwiedzone, przeszukiwanie od wierzchołka  $u$  możemy w pseudojęzyku programowania opisać w postaci następującej procedury rekurencyjnej:

```
procedure DFS(u)
    visited[u] := true
    for each v ∈ L[u] do
        if not visited[v] then DFS(v)
```

Procedurę *DFS* można stosować zarówno do grafów skierowanych, jak i nieskierowanych. W przypadku grafów skierowanych wybierane są w niej tylko te krawędzie, które są skierowane od  $u$  do nieodwiedzonych wierzchołków  $v$ .

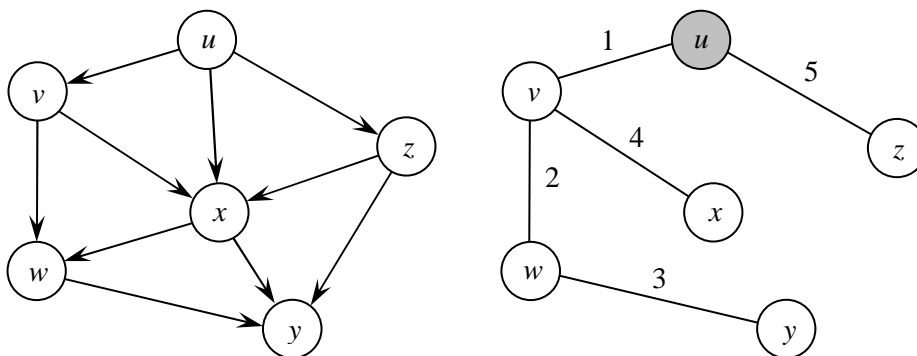
Aby odwiedzić wszystkie wierzchołki grafu, należy najpierw zaznaczyć je jako nieodwiedzone, a następnie wykonać procedurę *DFS* dla każdego nieodwiedzonego jeszcze wierzchołka:

```
for each u ∈ V do
    visited[u] := false
for each u ∈ V do
    if not visited[u] then DFS(u)
```

Nazwa algorytmu wywodzi się stąd, że przechodzi on wybraną ścieżką aż do jej całkowitego wyczerpania. W kolejnych wywołaniach rekurencyjnych procedury *DFS* przeszukiwanie zagłębia się coraz bardziej, jak tylko to jest możliwe. Ścieżka rozpoczyna się w wierzchołku  $u$ , biegnie wzdłuż krawędzi wychodzącej od  $u$  do nieodwiedzonego wierzchołka  $v$ , dalej wzdłuż krawędzi wiodącej od wierzchołka  $v$  do jego nieodwiedzonego następnika itd. Po dojściu do końca ścieżki i oznaczeniu nieodwiedzonych wierzchołków jako odwiedzonych następuje powrót i wybór kolejnej ścieżki prowadzącej od wcześniej odwiedzonych wierzchołków. Dokładniej, jeśli miał miejsce wybór krawędzi wiodącej od wierzchołka  $u$  do  $v$ , to w przypadku, gdy wierzchołek  $v$  jest nieodwiedzony, przeszukiwanie rozpoczyna się od nowa od  $v$ , a po wyczerpaniu wszystkich ścieżek wiodących od  $v$  następuje powrót do wierzchołka  $u$  i wybór kolejnej krawędzi wychodzącej od  $u$ . Natomiast w przypadku,

gdy wierzchołek  $v$  był już odwiedzony, przeszukiwanie wzdłuż ścieżki wiodącej od  $v$  jest pomijane, ponieważ odbyło się wcześniej. Dlatego od razu wybierana jest kolejna krawędź wychodząca od wierzchołka  $u$ .

Przykładowo, kolejność odwiedzania wierzchołków grafu przedstawionego na rysunku 4.9, przy uporządkowanych alfabetycznie listach sąsiedztwa i rozpoczęciu przeszukiwania grafu w głąb od wierzchołka  $u$ , jest następująca:  $u, v, w, y, x, z$ . Na rysunku pokazane jest również **drzewo przeszukiwań** zbudowane przez algorytm. Początkowo zawierało ono tylko korzeń  $u$ , a gdy podczas przechodzenia grafu napotkany został jego nieodwiedzony wierzchołek, zarówno ten wierzchołek, jak i prowadząca do niego krawędź, zostały dodane do drzewa. Numery obok krawędzi wskazują, w jakiej kolejności drzewo było rozbudowywane.



Rys. 4.9. Przykładowy graf i jego drzewo przeszukiwań w głąb

Jeżeli graf nieskierowany jest spójny, każdy wierzchołek zostanie odwiedzony podczas jednego wykonania procedury *DSF*. Jeżeli graf jest niespójny, procedura przeszukuje tylko jedną jego spójną składową. Zatem aby dokończyć przeszukiwanie całego grafu, należy wybrać jeden z nieodwiedzonych wierzchołków i wykonać nowe przeszukiwanie.

Zazwyczaj w procedurze *DFS* wraz z odwiedzaniem wierzchołków wykonuje się jakąś operację. Wówczas, oprócz prostej instrukcji przypisania zmieniającej status wierzchołka z określonego jako *nieodwiedzony* na *odwiedzony*, występują dodatkowe instrukcje, które precyzują tę operację. Modyfikując te instrukcje, można uzyskać różne algorytmy. W ten sposób algorytm przeszukiwania grafu w głąb określa pewien uniwersalny szablon postępowania, na którym zasada się wiele innych algorytmów grafowych. Ze względu na rolę, jaką algorytm przeszukiwania w głąb odgrywa w projektowaniu algorytmów grafowych, zapiszemy go w zwartej postaci, oznaczając komentarzem dodatkową operację odwiedzania wierzchołków. Oto pełny algorytm *DFS*:

```

procedure DEPTH-FIRST-SEARCH( $V, L$ )

  procedure DFS( $u$ )
    visited[ $u$ ] := true
    // Odwiedź wierzchołek  $u$ 
    for each  $v \in L[u]$  do
      if not visited[ $v$ ] then DFS( $v$ )

  for each  $u \in V$  do
    visited[ $u$ ] := false
  for each  $u \in V$  do
    if not visited[ $u$ ] then DFS( $u$ )

```

Przejdźmy teraz do oszacowania złożoności czasowej algorytmu. Zakładamy, że graf ma  $m$  krawędzi i  $n$  wierzchołków. Zauważmy, że obie pętle występujące w procedurze *DEPTH-FIRST-SEARCH* wymagają  $O(n)$  kroków, a jeśli pominiemy wywołania rekurencyjne procedury *DFS*, będzie ona wywołana jeden raz dla każdego wierzchołka. Z kolei analiza prostego kodu tej procedury prowadzi do wniosku, że dla żadnego z wierzchołków nie może być ona wywoływana więcej niż raz, ponieważ w momencie odwiedzenia wierzchołek otrzymuje status *odwiedzony*, co wyklucza ponowne odwiedzenie go. Zatem czas przeszukiwania wszystkich list sąsiedztwa jest proporcjonalny do łącznej długości tych list, czyli liczby krawędzi grafu, jest więc równy  $O(m)$ . Wynika stąd, że złożoność czasowa przeszukiwania w głąb jest proporcjonalna do rozmiaru grafu:

$$T(m, n) = O(m) + O(n) = O(m + n).$$

#### 4.4. Przeszukiwanie grafu wszerz

Inna prosta metoda przechodzenia grafu, nazywana **przeszukiwaniem wszerz** (ang. *breadth first search*), polega na odwiedzaniu od razu wszystkich nieodwiedzonych sąsiadów  $v \in L[u]$  rozpatrywanego wierzchołka  $u$ . Metoda używa kolejki pojedynczej do zapamiętania niewykorzystanych wierzchołków:

```

procedure BFS( $u$ )
   $Q := \emptyset$ 
  visited[ $u$ ] := true
  inject( $Q, u$ )
  while  $Q \neq \emptyset$  do
     $u := \text{pop}(Q)$ 
    for each  $v \in L[u]$  do
      if not visited[ $v$ ] then
        visited[ $v$ ] := true
        inject( $Q, v$ )

```

Podobnie jak w przypadku procedury *DFS* zakładamy, że początkowo wszystkie wierzchołki grafu są nieodwiedzone. Najpierw do wstępnie pustej kolejki  $Q$  wstawiamy, przekazany w parametrze przez wartość, wierzchołek  $u$ , od którego rozpoczyna się przeszukiwanie. Potem, dopóki kolejka  $Q$  nie jest pusta, pobieramy z niej wierzchołek  $u$ , a wstawiamy do niej wszystkie nieodwiedzone wierzchołki  $v \in L[u]$ . Umieszczanym w kolejce wierzchołkom nadajemy status *odwiedzony*. Różnica pomiędzy procedurą *DSF* a *BSF* jest taka, że w pierwszej każdy odwiedzany wierzchołek odkładamy na stos<sup>3</sup>, a w drugiej wstawiamy do kolejki. Konsekwencją użycia innej struktury do zapamiętania wierzchołków, które mają być wykorzystywane w dalszym przeszukiwaniu grafu, jest inna kolejność odwiedzania ich. Za każdym razem, gdy wierzchołek  $u$  jest pobierany z kolejki, następuje od razu odwiedzenie wszystkich jego nieodwiedzonych sąsiadów  $v \in L[u]$ , a dopiero potem przejście do sąsiadów wierzchołków  $v$ . Mówiąc bardziej obrazowo, po dotarciu do wierzchołka  $u$  poruszamy się wszerz grafu, aby odwiedzić wszystkich jego nieodwiedzonych sąsiadów. Stąd wywodzi się nazwa algorytmu.

Precyzując w pseudojęzyku algorytm przechodzenia grafu wszerz, umieszczamy treść procedury *BFS* bezpośrednio w zakresie pętli wybierającej nieodwiedzone wierzchołki do kolejnych przeszukiwań. Jednocześnie, aby uniknąć konfliktu nazw wierzchołków, zastępujemy nazwę  $u$  pobieranego z kolejki wierzchołka nazwą  $w$ . Ponadto, ponieważ przeszukiwanie wszerz odgrywa dużą rolę w projektowaniu algorytmów grafowych, podobnie jak przeszukiwanie w głąb, oznaczamy operację związaną z odwiedzaniem wierzchołków komentarzem. W rezultacie otrzymujemy następujący kod algorytmu przeszukiwania grafu wszerz:

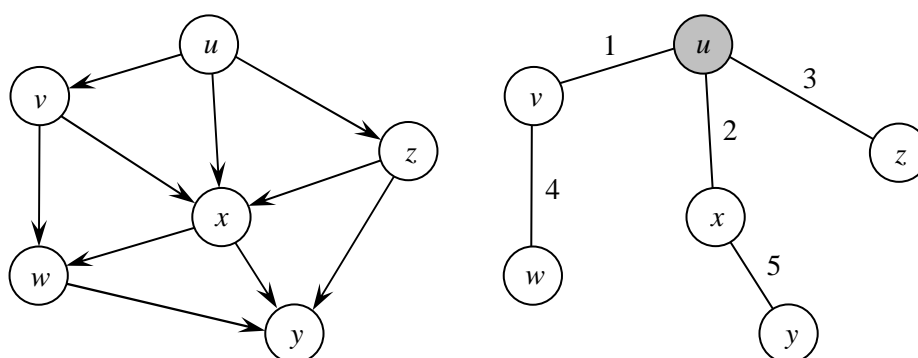
```

procedure BREADTH-FIRST-SEARCH( $V$ ,  $L$ )
  for each  $u \in V$  do
    visited[ $u$ ] := false
   $Q := \emptyset$ 
  for each  $u \in V$  do
    if not visited[ $u$ ] then
      visited[ $u$ ] := true
      inject( $Q$ ,  $u$ )
      while  $Q \neq \emptyset$  do
         $w := \text{pop}(Q)$ 
        // Odwiedź wierzchołek  $w$ 
        for each  $v \in L[w]$  do
          if not visited[ $v$ ] then
            visited[ $v$ ] := true
            inject( $Q$ ,  $v$ )

```

<sup>3</sup> Niejawne użycie stosu w procedurze *DSF* wynika z rekurencji, która w niej występuje. Iteracyjne wersje procedury *DSF*, wykorzystujące stos jawnie, można znaleźć m.in. w podręcznikach [3, 18].

Na rysunku 4.10 przedstawiony jest przytoczony wcześniej graf (por. rys. 4.9), który tym razem jest przeszukiwany wszerz od wierzchołka  $u$ . Uzyskana kolejność odwiedzania wierzchołków grafu jest inna niż poprzednio:  $u, v, x, z, w, y$ . Inne jest także drzewo przeszukiwań zbudowane przez algorytm.



Rys. 4.10. Przykładowy graf i jego drzewo przeszukiwań wszerz

Algorytm przeszukiwania wszerz można stosować zarówno dla grafów skierowanych, jak i nieskierowanych. W sposób analogiczny jak dla przeszukiwania w głąb można wykazać, że złożoność czasowa algorytmu wynosi  $O(m+n)$ .

## 4.5. Drzewo rozpinające grafu

Algorytmy przeszukiwania grafu  $G = (V, E)$  w głąb i wszerz można łatwo przystosować do znajdowania tzw. drzewa rozpinającego grafu. W obu przypadkach w trakcie przeszukiwania grafu wybierane są te jego krawędzie, które wiodą od wierzchołków odwiedzonych do nieodwiedzonych. Oznaczmy zbiór tych krawędzi przez  $T$ . Graf  $R = (V, T)$  złożony z wierzchołków  $V$  i krawędzi  $T \subset E$ , będący podgrafem grafu  $G$ , nazywamy **lasem rozpinającym** grafu  $G$  przy przeszukiwaniu w głąb lub wszerz, a gdy  $R$  jest drzewem, nazywamy go **drzewem rozpinającym** grafu  $G$ . Jeżeli graf  $G$  jest nieskierowany i spójny, tj. gdy dla każdej pary wierzchołków  $u, v \in V$  istnieje ścieżka prowadząca od wierzchołka  $u$  do wierzchołka  $v$ , las rozpinający grafu jest drzewem rozpinającym. Korzeniem tego drzewa jest wierzchołek, w którym rozpoczęto zostało przeszukiwanie.

Aby wygenerować las rozpinający grafu, wystarczy podczas przeszukiwania go rozszerzać pusty wstępnie zbiór krawędzi  $T$  o krawędzie wiodące od wierzchołków odwiedzonych do nieodwiedzonych. W przypadku przeszukiwania w głąb modyfikacja procedury *DEPTH-FIRST-SEARCH* prowadzi do następującego kodu:

```

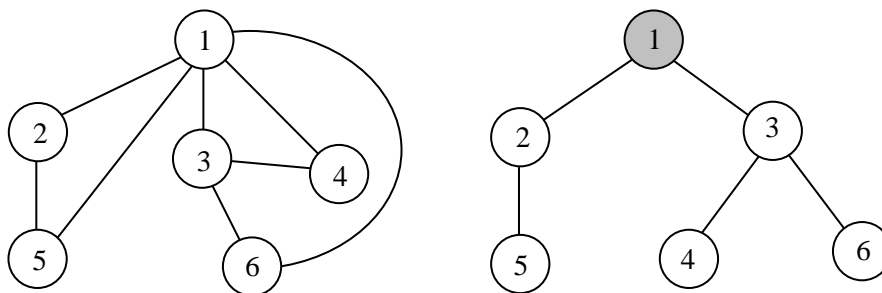
procedure DFS-SPANNING-TREE( $V, L, T$ )

  procedure DFS-TREE( $u$ )
    visited[ $u$ ] := true
    for each  $v \in L[u]$  do
      if not visited[ $v$ ] then
        dodaj krawędź ( $u, v$ ) do  $T$ 
        DFS-TREE( $v$ )

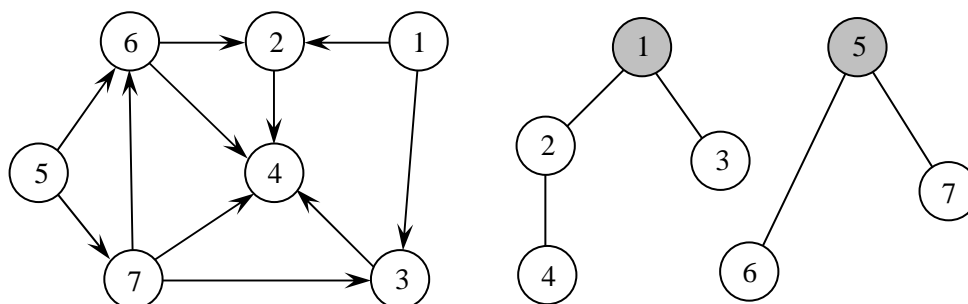
   $T := \emptyset$ 
  for each  $u \in V$  do
    visited[ $u$ ] := false
  for each  $u \in V$  do
    if not visited[ $u$ ] then DFS-TREE( $u$ )

```

Na rysunku 4.11a pokazano przykładowy graf nieskierowany i jego drzewo rozpinające, a na rysunku 4.11b graf skierowany i jego las rozpinający złożony z dwóch drzew. Wyniki uzyskano podczas przeszukiwania grafów w głąb przy uporządkowanej rosnąco liście wierzchołków i ich listach sąsiedztwa.



a) przykładowy graf nieskierowany i jego drzewo rozpinające

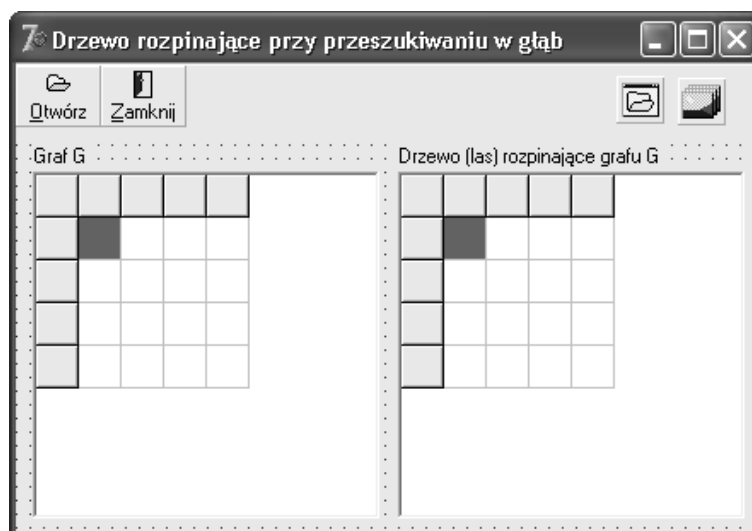


b) przykładowy graf skierowany i jego las rozpinający

**Rys. 4.11.** Drzewo i las rozpinający przy przeszukiwaniu w głąb



Zbudujemy teraz aplikację okienkową w Delphi, która znajduje drzewo rozpinające (las rozpinający) grafu przy przeszukiwaniu w głąb. Projekt formularza tej aplikacji przedstawia rysunek 4.12. Oprócz komponentów *ToolBar* i *ImageList*, umożliwiających stworzenie paska narzędziowego z dwoma przyciskami, oraz komponentu *OpenDialog*, pozwalającego na wygodny wybór pliku tekstowego zawierającego definicję grafu, na formularzu znajdują się dwie siatki tekstowe *StringGrid* opisane etykietami. W pierwszej siatce wyświetlona zostanie macierz sąsiedztwa przeszukiwanego grafu, a w drugiej macierz sąsiedztwa jego drzewa (lasu) rozpinającego. Szerokość kolumn obu siatek zmniejszamy do 24, przypisując tę wartość ich właściwości *DefaultColWidth*.



Rys. 4.12. Projekt formularza aplikacji znajdującej las rozpinający grafu

C całą pracę aplikacji będzie wykonywać procedura obsługi zdarzenia *OnClick* pierwszego przycisku *ToolButton*, bowiem rola drugiego przycisku sprowadza się jedynie do zamknięcia okna. Gdy użytkownik wybierze plik tekstowy, procedura ma wczytać z niego dane opisujące graf *G*, znaleźć drzewo *R* (lub las) rozpinające ten graf i wyświetlić macierze sąsiedztwa grafów *G* i *R*:

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
    if not OpenDialog1.Execute then Exit;
    // Wczytaj graf G z pliku
    // Znajdź drzewo R rozpinające graf G
    // Wyświetl grafy G i R
end;
```

Aby kod tej metody zbytnio nie rozrósł się, wydzielimy z niego dwa zadania, czytania grafu  $G$  z pliku i znajdowania drzewa rozpinającego  $R$ , precyzując je jako procedury w osobnym module *DFSUnit*. Pomimo że algorytm *DFS-SPANNING-TREE* został sformułowany dla list sąsiedztwa, w jego implementacji w Object Pascalu użyjemy reprezentacji macierzowej grafu, która w tym przypadku wydaje się wygodniejsza. Możemy mianowicie, korzystając z dwuwymiarowych tablic dynamicznych, zdefiniować nowy typ grafowy:

```
type
  TGraph = array of array of Boolean;
```

Tablica typu *TGraph* może być traktowana zarówno jako jednowymiarowa tablica wierszy, czyli tablic jednowymiarowych, jak i jako dwuwymiarowa tablica elementów typu *Boolean*. W pierwszym przypadku liczbę wierszy można określić za pomocą procedury *SetLength*, a następnie liczbę elementów każdego wiersza za pomocą odrębnego wywołania tej procedury. W ten sposób można np. utworzyć tablicę trójkątną [22]. W drugim przypadku, za pomocą tylko jednego wywołania procedury *SetLength* o zwiększonej o 1 liczbie parametrów, można określić liczby wierszy i kolumn całej tablicy. Na przykład instrukcja

```
SetLength(A, 10, 20);
```

przydziela pamięć tablicy  $A$  o 10 wierszach i 20 kolumnach. Pewną niedogodność może sprawiać indeksacja elementów tablic dynamicznych od zera wzwyż, co niestety nastąpi w rozpatrywanym przypadku, gdy zamiast naturalnej numeracji wierzchołków grafu  $G$  od 1 do  $n$  musimy użyć numeracji od 0 do  $n - 1$ .

Znaleźliśmy się w sytuacji, w której należy podjąć decyzję dotyczącą zawartości pliku tekstowego. Przyjmiemy, że pierwszy wiersz pliku zawiera dużą literę określającą rodzaj grafu  $G$  ( $S$  – skierowany,  $N$  – nieskierowany) i jego największy wierzchołek  $n$  (liczba całkowita dodatnia), a każdy następny dwa wierzchołki  $u$  i  $v$  (liczby całkowite od 1 do  $n$ ) określające krawędź grafu od  $u$  do  $v$ . Na przykład pliki grafów przedstawionych na rysunku 4.11 mogą mieć postać następującą:

N	6	S	7
1	2	1	2
1	5	1	3
1	3	2	4
1	4	3	4
1	6	5	6
2	5	5	7
3	4	6	2
3	6	6	4
		7	3
		7	4
		7	6

Jeżeli graf  $G$  jest skierowany, czytanie danych z pliku tekstowego do tablicy  $A$  typu  $TGraph$ , reprezentującej macierz sąsiedztwa wierzchołków grafu, możemy zaprogramować następująco:

```
ReadLn(Plik, n);
SetLength(A, n, n);
for u:=0 to n-1 do
  for v:=0 to n-1 do A[u,v] := false;
while not Eof(Plik) do
begin
  ReadLn(Plik, u, v);
  A[u-1,v-1] := true;
end;
```

Jak widać, po wczytaniu liczby  $n$  określającej największy wierzchołek grafu  $G$  przydzielamy pamięć tablicy  $A$  i wypełniamy jej wszystkie elementy wartością *false*. Uzyskany w ten sposób pusty zbiór krawędzi grafu rozszerzamy o nowe krawędzie, wczytując z pliku wyznaczające je pary wierzchołków  $u, v$  i przypisując stosownym elementom tablicy  $A$  wartość *true*. Jeżeli graf  $G$  jest nieskierowany, należy dodatkowo wypełnić elementy symetryczne tablicy  $A$ , ponieważ krawędź wiodąca od wierzchołka  $u$  do  $v$  jest również krawędzią od  $v$  do  $u$ .

Nietrudno w podobny sposób sprecyzować operacje grafowe występujące w procedurze *DFS-SPANNING-TREE*. Decydując się na najprostszą obsługę wyjątków, które mogą zdarzyć się podczas czytania pliku, i zakładając, że tablica dynamiczna  $T$  typu  $TGraph$  reprezentuje macierz sąsiedztwa grafu wynikowego  $R$ , możemy moduł *DSFUnit* sformułować w następującej postaci:

```
unit DSFUnit;

interface

type
  TGraph = array of array of Boolean;

procedure LoadGraph(FileName: string; var A: TGraph);

procedure DFS_Spanning_Tree(var A, T: TGraph);

implementation

procedure LoadGraph(FileName: string; var A: TGraph);
var
  Plik: TextFile;
  n, u, v: integer;
  c: char;
begin
  AssignFile(Plik, FileName);
```

```

Reset(Plik);
try
  ReadLn(Plik, c, n);
  SetLength(A, n, n);
  for u:=0 to n-1 do
    for v:=0 to n-1 do A[u,v] := false;
  while not Eof(Plik) do
  begin
    ReadLn(Plik, u, v);
    if (u > 0) and (u <= n) and (v > 0) and (v <= n) then
    begin
      A[u-1,v-1] := true;
      if c = 'N' then A[v-1,u-1] := true;
    end;
  end;
finally
  CloseFile(Plik);
end;
end;

procedure DFS_Spanning_Tree(var A, T: TGraph);

  var n, u, v: integer;
      visited: array of Boolean;

  procedure DFS_Tree(u: integer);
  var v: integer;
  begin
    visited[u] := true;
    for v:=0 to n-1 do
      if A[u,v] and not visited[v] then
      begin
        T[u,v] := true;
        DFS_Tree(v);
      end;
    end;
  end;

begin
  // DFS_Spanning_Tree
  n := Length(A);
  SetLength(T, n, n);
  SetLength(visited, n);
  for u:=0 to n-1 do
  begin
    for v:=0 to n-1 do T[u,v] := false;
    visited[u] := false;
  end;
  for u:=0 to n-1 do
    if not visited[u] then DFS_Tree(u);
  end;
end.

```

Rozbudowę modułu formularza aplikacji zaczynamy od rozszerzenia jego listy **uses** o nazwę *DFSUnit* przedstawionego wyżej modułu i umieszczenia w definicji klasy *TForm1* pól *A* i *T* typu *TGraph*, reprezentujących grafy *G* i *R*, oraz nagłówka metody *ShowGraph* wyświetlającej graf w siatce łańcuchów:

```
A, T: TGraph;
procedure ShowGraph(var A: TGraph; Grid: TStringGrid);
```

Możemy już sprecyzować metodę *ToolButton1Click* klasy *TForm1*, zastępując występujące w niej komentarze instrukcjami polecającymi wczytać graf *G* z pliku, znaleźć jego drzewo (las) rozpinające *R* i wyświetlić oba grafy:

```
LoadGraph(OpenDialog1.FileName, A);
DFS_Spanning_Tree(A, T);
ShowGraph(A, StringGrid1);
ShowGraph(T, StringGrid2);
```

Ostatnim etapem tworzenia aplikacji, której projekt zapisujemy np. w pliku *Rozwin.dpr*, jest uzupełnienie wygenerowanego przez Delphi pustego szkieletu metody *ShowGraph*. Zadaniem metody jest wyświetlenie grafu reprezentowanego przez macierz *A* w siatce *Grid*. Liczba wierszy i kolumn siatki jest o 1 większa od liczby wierzchołków grafu, ponieważ wiersz 0 i kolumna 0 są przeznaczone do prezentowania numerów wierzchołków. Pozostałe komórki siatki, na przecięciu wierszy *u* i kolumn *v*, wypełniamy znakiem  $\times$ , gdy istnieje krawędź wiodąca od wierzchołka *u* do *v*, albo pozostawiamy puste, gdy takiej krawędzi nie ma. Pełny kod metody jest zawarty w następującej wersji końcowej modułu formularza:

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Grids, ImgList, ComCtrls,
  ToolWin, DSFUnit;

type
  TForm1 = class(TForm)
    ToolBar1: TToolBar;
    ToolButton1: TToolButton;
    ToolButton2: TToolButton;
    ImageList1: TImageList;
    OpenDialog1: TOpenDialog;
    StringGrid1: TStringGrid;
    StringGrid2: TStringGrid;
    Label1: TLabel;
    Label2: TLabel;
```

```

    procedure ToolButton1Click(Sender: TObject);
    procedure ToolButton2Click(Sender: TObject);
private
    A, T: TGraph;
    procedure ShowGraph(var A: TGraph; Grid: TStringGrid);
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ShowGraph(var A: TGraph; Grid: TStringGrid);
var
    n, u, v: integer;
begin
    n := Length(A);
    Grid.ColCount := n+1;
    Grid.RowCount := n+1;
    for u:=1 to n do
        begin
            Grid.Cells[u,0] := IntToStr(u);
            Grid.Cells[0,u] := IntToStr(u);
            for v:=1 to n do
                if A[u-1,v-1] then Grid.Cells[v,u] := 'x'
                else Grid.Cells[v,u] := '';
            end;
        end;
    end;

procedure TForm1.ToolButton1Click(Sender: TObject);
begin
    if not OpenDialog1.Execute then Exit;
    LoadGraph(OpenDialog1.FileName, A);
    DFS_Spanning_Tree(A, T);
    ShowGraph(A, StringGrid1);
    ShowGraph(T, StringGrid2);
end;

procedure TForm1.ToolButton2Click(Sender: TObject);
begin
    Close;
end;

end.

```

Przykładowe wyniki wykonania aplikacji *Rozwin*, uzyskane dla grafu skierowanego przedstawionego na rysunku 4.11b, są przedstawione na rysunku 4.13. Zawartość pliku opisującego ten graf jest przytoczona na jednej z wcześniejszych stron niniejszego podrozdziału.

Graf G								Drzewo (las) rozpinające grafu G							
	1	2	3	4	5	6	7		1	2	3	4	5	6	7
1		x	x					1		x	x				
2				x				2				x			
3				x				3							
4								4							
5						x	x	5						x	x
6		x		x				6							
7			x	x		x		7							

Rys. 4.13. Przykładowe wyniki aplikacji znajdującej las rozpinający grafu

## 4.6. Spójne składowe grafu nieskierowanego

Jednym z podstawowych problemów grafowych jest znajdowanie spójnych składowych grafu nieskierowanego. Przypomnijmy, że graf nieskierowany nazywamy **spójnym**, gdy każde dwa jego wierzchołki można połączyć ścieżką, zaś **spójną składową** grafu nazywamy jego maksymalny, w sensie zawierania wierzchołków i krawędzi, spójny podgraf<sup>4</sup>. Specyfikacja problemu znajdowania spójnych składowych grafu nieskierowanego jest następująca:

**Dane:** Graf nieskierowany  $G = (V, E)$ , gdzie  $V = \{1, 2, \dots, n\}$ .

**Wynik:** Funkcja  $C$  przypisująca wierzchołkom  $w \in V$  liczbę naturalną  $C(w)$  taką, że dla wierzchołków  $u, v \in V$  równość  $C(u) = C(v)$  zachodzi wtedy i tylko wtedy, gdy należą one do tej samej spójnej składowej.

Wartość  $C(w)$  jest identyfikatorem spójnej składowej zawierającej wierzchołek  $w$ . W algorytmie, który zbudujemy w oparciu o strategię przechodzenia grafu w głąb, identyfikatorami składowych będą kolejne liczby całkowite od 1 wzwyż przypisywane składowym w miarę ich znajdowania<sup>5</sup>. Identyfikatory składowych

<sup>4</sup> Dla grafów skierowanych definiuje się silną i słabą spójność (por. ćw. 4.11 i 4.12).

<sup>5</sup> Inną numerację spójnych składowych otrzymamy, określając wartość  $C(w)$  jako równą najmniejszemu wierzchołkowi w spójnej składowej zawierającej wierzchołek  $w$  [3].

będą pamiętane w tablicy  $C[1..n]$ , a identyfikator wykrywanej właśnie składowej w pomocniczej zmiennej  $id$ . Modyfikacja procedury *DFS* polega na zastąpieniu występującego w niej komentarza instrukcją przypisania elementowi  $C[u]$  wartości  $id$ . Ze względu na numerację wierzchołków od 1 do  $n$  obie konstrukcje **for each ...** w procedurze *DEPTH-FIRST-SEARCH* zastępujemy pascelowymi pętlami **for**. Druga rozpoczyna przeszukiwanie spójnej składowej grafu, więc w niej nadajemy wstępnie wyzerowanej zmiennej  $id$  kolejną wartość całkowitą. W rezultacie otrzymujemy następujący algorytm wyznaczania spójnych składowych grafu:

```

procedure DFS-COMPONENT(L, C)

  procedure COMPONENT(u)
    visited[u] := true
    C[u] := id
    for each v ∈ L[u] do
      if not visited[v] then COMPONENT(v)

  for u:=1 to n do
    visited[u] := false
  id := 0
  for u:=1 to n do
    if not visited[u] then
      id := id+1
      COMPONENT(u)

```

## 4.7. Znajdowanie najkrótszych ścieżek w grafie

Jednym z najważniejszych problemów grafowych jest znajdowanie najkrótszych ścieżek (dróg) w grafach ważonych (skierowanych lub nie). Na przykład podróżni latający samolotami często szukają najkrótszego połączenia od jednego miasta do drugiego, gdy nie mają połączenia bezpośredniego. Problem znajdowania najkrótszych ścieżek w grafie formułujemy następująco:

**Dane:** Graf  $G = (V, E)$  o wierzchołkach  $V = \{1, 2, \dots, n\}$ , przyporządkowane wszystkim krawędziom  $(u, v) \in E$  liczby rzeczywiste  $C[u, v]$ , zwane **wagami**, i dwa ustalone wierzchołki  $p, q \in V$ .

**Wynik:** Długość najkrótszej ścieżki prowadzącej od wierzchołka  $p$  do  $q$ .

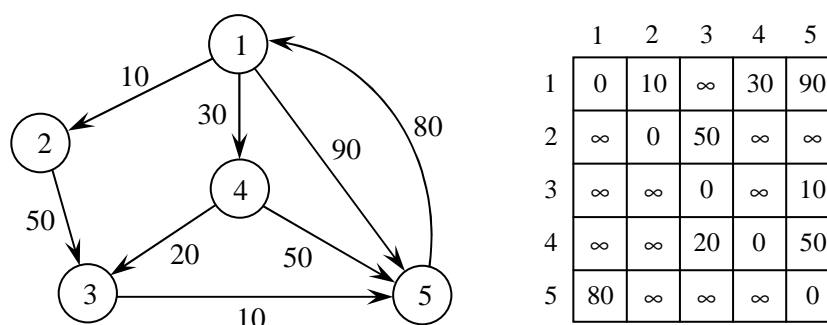
Macierz kwadratowa  $C$  o rozmiarze  $n \times n$  jest nazywana **macierzą kosztów** przejścia wzdłuż krawędzi grafu. Tak więc, element  $C[u, v]$  jest kosztem krawędzi wiodącej od wierzchołka  $u$  do  $v$ . Jeżeli w grafie  $G$  nie ma takiej krawędzi, to  $C[u, v] = \infty$ . Nieskończoność może być reprezentowana przez odpowiednio dużą wartość dodatnią, zbyt dużą, aby mogła być kosztem. Przyjmujemy, że  $C[u, u] = 0$ ,



czyli że graf  $G$  nie ma pętli. Długością lub kosztem ścieżki określonej przez ciąg wierzchołków  $v_1, v_2, \dots, v_k$  nazywamy sumę wag jej wszystkich krawędzi:

$$\sum_{i=1}^{k-1} C[v_i, v_{i+1}]$$

Przykładowy graf ważony wraz z macierzą kosztów jest pokazany na rysunku 4.14. W praktyce elementy macierzy kosztów często utożsamia się z odległościami między wierzchołkami, czasem lub kosztem przejazdu.



**Rys. 4.14.** Przykładowy graf ważony i jego macierz kosztów

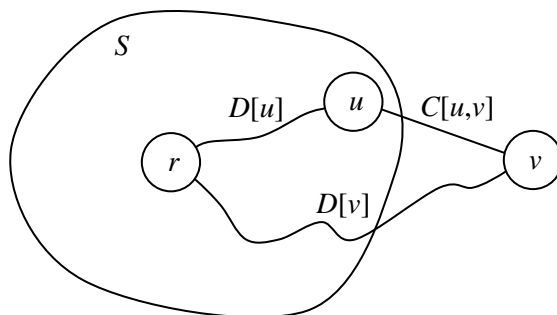
Najbardziej znanym algorytmem wyznaczania najkrótszych ścieżek w grafie ważonym jest **algorytm Dijkstry**. Pozwala on znaleźć najkrótsze ścieżki prowadzące od wyróżnionego wierzchołka  $r \in V$ , który nazwiemy **źródłem**, do każdego innego wierzchołka  $v \in V$ . W algorytmie prowadzony jest pomocniczy zbiór  $S$  zawierający te wierzchołki grafu, dla których długości najkrótszych ścieżek od źródła  $r$  zostały już obliczone, oraz tablica  $D$  długości znanych ścieżek od źródła do wszystkich wierzchołków. Dla  $v \in S$  wartości  $D[v]$  są długościami najkrótszych ścieżek prowadzących od  $r$  do  $v$ . Oto zapis algorytmu w pseudojęzyku:

```

S := [r]
D[r] := 0
for each v ∈ V-[r] do
    D[v] := C[r,v]
while S ≠ V do
    u := wierzchołek ∈ V-S taki, że dla v ∈ V-S
        D[u] = min D[v]
    S := S ∪ {u}
    for each v ∈ V-S do
        D[v] := min{D[v], D[u]+C[u,v]}

```

Na początku zbiór  $S$  zawiera tylko jeden wierzchołek – źródło  $r$ , zaś wartości  $D[v]$  są równe wartościom  $C[r, v]$ , czyli są wagami krawędzi wiodących od  $r$  do  $v$  bądź wynoszą  $\infty$ , gdy takich krawędzi w grafie nie ma. W kolejnych krokach zbiór  $S$  jest rozbudowywany poprzez dodawanie do niego tych wierzchołków  $u$ , których odległości  $D[u]$  od źródła  $r$  są możliwie jak najmniejsze. Jednocześnie aktualizowana jest tablica  $D$ . Mówiąc dokładniej, jeżeli ścieżka od źródła  $r$  poprzez wierzchołki ze zbioru  $S$  do nowego wierzchołka  $u \in S$ , a następnie do wierzchołka  $v \notin S$  ma mniejszą długość niż dotąd znana ścieżka od  $r$  do  $v$  poprzez wierzchołki zbioru  $S$  (por. rys. 4.15), to wartość  $D[v]$  jest modyfikowana. Na końcu zbiór  $S$  zawiera wszystkie wierzchołki grafu, a tablica  $D$  długości najkrótszych ścieżek prowadzących do tych wierzchołków od źródła  $r$ .



**Rys. 4.15.** Wybór krótszej ścieżki w algorytmie Dijkstry

Tabela 4.1 przedstawia stan zmiennych algorytmu Dijkstry po wykonaniu jego kolejnych iteracji dla grafu pokazanego na rysunku 4.14 i źródła  $r = 1$ .

**Tabela 4.1.** Działanie algorytmu Dijkstry dla grafu z rysunku 4.14 i źródła  $r = 1$

Iteracja	$S$	$u$	$D[1]$	$D[2]$	$D[3]$	$D[4]$	$D[5]$
Pocz.	[1]	–	0	10	$\infty$	30	90
1	[1, 2]	2	0	10	60	30	90
2	[1, 2, 4]	4	0	10	50	30	80
3	[1, 2, 4, 3]	3	0	10	50	30	60
4	[1, 2, 4, 3, 5]	5	0	10	50	30	60

Opisany algorytm wyznacza jedynie długości najkrótszych ścieżek, ale nie zapamiętuje ich postaci. Wypada go uzupełnić, by dostarczał również informacji, któredy wiodą ścieżki. Okazuje się, że modyfikacja jest bardzo łatwa.

Istotnie, wystarczy skorzystać z drugiej tablicy, nazwijmy ją  $P$ , której element o indeksie  $v$  będzie pamiętał wierzchołek poprzedzający wierzchołek  $v$  na ścieżce wiodącej od  $r$  do  $v$ . Początkowo wszystkie elementy tablicy  $P$  o indeksach  $v \neq r$ , dla których istnieje krawędź od  $r$  do  $v$ , mają wartość  $r$ , zaś pozostałe wartość zero, która oznacza, że stosowne wierzchołki nie mają poprzedników. Uaktualnienie tablicy  $P$  odbywa się wraz z uaktualnieniem tablicy  $D$ , w momencie znalezienia krótszej ścieżki wiodącej od źródła  $r$  poprzez wierzchołek  $u$  do wierzchołka  $v$ . Wówczas w elemencie  $P[v]$  zapamiętuje się wierzchołek  $u$  – poprzednik wierzchołka  $v$ . Zmodyfikowany w ten sposób algorytm ma postać następującą:

```

procedure Dijkstra( $V, C, r, D, P$ )
   $S := \{r\}$ 
   $D[r] := 0$ 
   $P[r] := 0$ 
  for each  $v \in V - \{r\}$  do
     $D[v] := C[r, v]$ 
    if istnieje krawędź  $(r, v)$ 
      then  $P[v] := r$  else  $P[v] := 0$ 
  while  $S \neq V$  do
     $u := \text{wierzchołek} \in V - S \text{ taki, że dla } v \in V - S$ 
       $D[u] = \min D[v]$ 
     $S := S \cup \{u\}$ 
    for each  $v \in V - S$  do
      if  $D[u] + C[u, v] < D[v]$  then
         $D[v] := D[u] + C[u, v]$ 
         $P[v] := u$ 

```

Nietrudno teraz rozszerzyć tabelę 4.1 o kolumny reprezentujące stan elementów tablicy  $P$  po wykonaniu poszczególnych iteracji w algorytmie Dijkstry dla grafu z rysunku 4.14 i źródła  $r = 1$ . Wyniki przedstawia tabela 4.2.

**Tabela 4.2.** Generowanie postaci ścieżek w algorytmie Dijkstry dla grafu z tabeli 4.1

Iteracja	$u$	$D[1]$	$D[2]$	$D[3]$	$D[4]$	$D[5]$	$P[1]$	$P[2]$	$P[3]$	$P[4]$	$P[5]$
Pocz.	–	0	10	$\infty$	30	90	0	1	0	1	1
1	2	0	10	60	30	90	0	1	2	1	1
2	4	0	10	50	30	80	0	1	4	1	4
3	3	0	10	50	30	60	0	1	4	1	3
4	5	0	10	50	30	60	0	1	4	1	3

Odtworzenie najkrótszej ścieżki wychodzącej od źródła  $r$  do dowolnego wierzchołka  $v$  jest proste. Wystarczy cofać się od  $v$  do  $r$ , przechodząc przez wierzchołki  $P[v]$ ,  $P[P[v]]$ ,  $P[P[P[v]]]$  itd., aż napotkana zostanie wartość zero. Przykładowo,

w ostatnim wierszu tabeli 4.2 odczytujemy, że poprzednikiem wierzchołka 5 jest wierzchołek 3, poprzednikiem wierzchołka 3 wierzchołek 4, a poprzednikiem wierzchołka 4 wierzchołek 1. Zatem najkrótsza ścieżka wiodąca od wierzchołka 1 do 5, której długość (koszt) wynosi 60, wiedzie przez wierzchołki 1, 4, 3, 5.

Kod algorytmu odtwarzającego postać ścieżki kończącej się wierzchołkiem  $v$ , zapisującego numery kolejnych jej wierzchołków w liście  $L$  o organizacji stosu, może wyglądać następująco:

```
L := ∅
repeat
  push(L, v)
  v := P[v]
until v = 0
```

Zdejmując numery wierzchołków ze stosu  $L$  za pomocą operacji *pop*, uzyskujemy dokładny przebieg ścieżki wiodącej od źródła  $r$  do wierzchołka  $v$ .

Algorytm Dijkstry cechuje **podejście zachłanne**. W każdej iteracji algorytm dodaje do zbioru  $S$  wierzchołek  $u \in V - S$ , do którego wiedzie najkrótsza ścieżka od źródła  $r$  poprzez wierzchołki należące do zbioru  $S$ . Zatem za każdym razem dokonuje zachłannego wyboru nowego wierzchołka. Zachłanność nie zawsze jest opłacalna. Algorytm Dijkstry prowadzi jednak do rozwiązania optymalnego, ale pod warunkiem, że przypisane krawędziom grafu **wagi są nieujemne**. W celu udowodnienia tego twierdzenia zapiszmy pierwszą wersję algorytmu w nieco innej postaci, bardziej zbliżonej do Pascala:

```
S := [r]
D[r] := 0
for v:=1 to n do
  if v ≠ r then D[v] := C[r,v]
for k:=2 to n do
  u := 0
  for v:=1 to n do
    if (v ∉ S) and ((u = 0) or (D[v] < D[u])) then u := v
  S := S ∪ {u}
  for v:=1 to n do
    if (v ∉ S) and (D[u]+C[u,v] < D[v])
      then D[v] := D[u]+C[u,v]
```

W dowodzie przez indukcję względem mocy  $k$  zbioru  $S$  wykażemy, że:

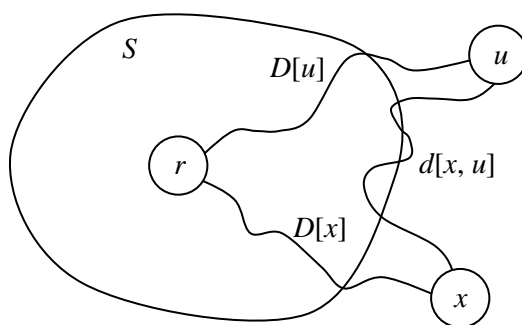
- 1) dla każdego  $u \in S$  najkrótsza ścieżka wiodąca od  $r$  do  $u$  poprzez wierzchołki należące do  $S$  ma długość  $D[u]$ ,
- 2) dla każdego  $v \in V - S$  najkrótsza ścieżka wiodąca od  $r$  do  $v$ , która przechodzi przez wierzchołki należące do  $S$ , z wyjątkiem samego  $v$ , ma długość  $D[v]$ .

Warunek początkowy indukcji matematycznej dla  $k = 1$  jest spełniony, bowiem zbiór  $S$  zawiera tylko jeden wierzchołek  $r$  i wiersze (\*) prawidłowo inicjalizują elementy tablicy  $D$ . Najkrótsza ścieżka wiodąca od  $r$  do  $r$  ma długość zero, a każda ścieżka od  $r$  do  $v$ , leżąca prócz  $v$  całkowicie w  $S$ , składa się z jednej krawędzi  $(r, v)$  o długości  $D[v] = C[r, v]$ .

Aby wykazać krok indukcyjny zakładamy, że warunki 1. i 2. są prawdziwe dla zbioru  $S$  zawierającego  $k$  wierzchołków. Przyjmujemy też, że w kolejnej iteracji w wierszach (\*\*) wybrany został wierzchołek  $u$ , który ma być włączony do  $S$ . Przypuśćmy, dla dowodu nie wprost, że  $D[u]$  nie jest długością najkrótszej ścieżki spośród wszystkich ścieżek wiodących od  $r$  do  $u$ . Zatem istnieje krótsza ścieżka  $P$  od  $r$  do  $u$ . Wobec indukcyjnego założenia o prawdziwości warunku 2. musi ona zawierać wierzchołek  $x \notin S$  różny od  $u$ , gdyż dla  $x = u$  miałaby długość  $D[u]$ . Niech  $x$  będzie pierwszym takim wierzchołkiem na ścieżce  $P$  (por. rys. 4.16). Na mocy warunku 2. ścieżka od  $r$  do  $x$  ma długość  $D[x]$ . Korzystając z założenia o nieujemności wag grafu, otrzymujemy:

$$D[x] \leq D[x] + d(x, u) < D[u],$$

gdzie  $d(x, u)$  jest długością części ścieżki  $P$  od  $x$  do  $u$ . Doszliśmy tu do sprzeczności, gdyż wierzchołek  $u$  został tak wybrany, by wartość  $D[u]$  była najmniejsza.



**Rys. 4.16.** Uzasadnienie algorytmu Dijkstry

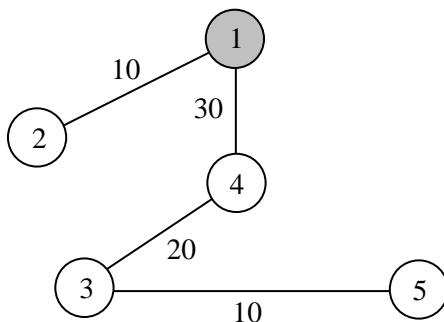
Możemy teraz bez naruszenia warunku 1. wstawić wierzchołek  $u$  do zbioru  $S$ , zwiększając liczbę jego elementów do  $k + 1$ . Po tej operacji niektóre najkrótsze dotąd ścieżki wiodące od  $r$  do  $v$ , które przechodziły przez wierzchołki zbioru  $S$  prócz wierzchołka  $v \notin S$ , mogą nie być najkrótszymi. Dzieje się tak, gdy dodanie nowego wierzchołka  $u$  do zbioru  $S$  spowoduje powstanie krótszej ścieżki od  $s$  do  $v$ , która prowadzi przez wierzchołki zbioru  $S$  do wierzchołka  $u$ , a następnie bezpośrednio do wierzchołka  $v$ . Długość takiej ścieżki wynosi  $D[u] + C[u, v]$ . Dlatego

w wierszach (\*\*\*), aby zapewnić spełnienie warunku 2., sprawdzamy długości nowych ścieżek wiodących od  $s$  do  $v$ , których przedostatnim wierzchołkiem jest  $u$ , i w razie potrzeby korygujemy wartości elementów  $D[v]$ .

Zatem po wykonaniu każdej iteracji warunki 1. i 2. są spełnione. Po ostatniej iteracji zbiór  $S$  zawiera wszystkie wierzchołki grafu, a zbiór  $V - S$  jest pusty, więc zgodnie z warunkiem 1., długości najkrótszych ścieżek wiodących od  $s$  do  $u$  są równe  $D[u]$ .

Przejdźmy teraz do oszacowania złożoności czasowej algorytmu Dijkstry. Jest oczywiste, że najwięcej czasu zajmuje druga pętla **for**, ponieważ w każdym jej wykonaniu odbywa się wybór wierzchołka  $u$  o najmniejszej wartości  $D[u]$  i przeglądanie pozostałych wartości  $D[v]$  w celu ich ewentualnego skorygowania. Wynika stąd, że algorytm wykonuje się w czasie  $O(n^2)$ . Przy bardziej odpowiednim doborze struktury danych można otrzymać wersję o złożoności  $O(m \log n)$  [1, 18].

Jeśli wykonamy algorytm Dijkstry dla grafu ważonego  $G(V, E)$  z nieujemnymi wagami i źródłem  $r$ , to w wyniku otrzymamy podgraf poprzedników, który jest **drzewem najkrótszych ścieżek** z korzeniem  $r$ . Przykład takiego drzewa dla grafu skierowanego z rysunku 4.14 i korzenia  $r = 1$  jest pokazany na rysunku 4.17.



**Rys. 4.17.** Drzewo najkrótszych ścieżek dla grafu z rys. 4.14

Bardziej ogólne zagadnienie polega na wyznaczaniu najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków grafu ważonego – APSP (skrót od ang. *all pairs shortest paths*). Specyfikacja problemu APSP jest następująca:

**Dane:** Graf  $G = (V, E)$  o wierzchołkach  $V = \{1, 2, \dots, n\}$  i przyporządkowane wszystkim krawędziom  $(u, v) \in E$  liczby rzeczywiste  $C[u, v]$  (wagi lub koszty krawędzi).

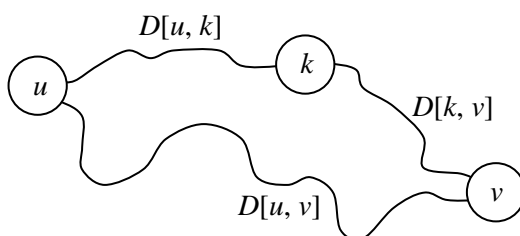
**Wynik:** Długości  $D[u, v]$  najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków  $u, v$ .

W przypadku nieujemnych wag rozwiązanie zagadnienia APSP można uzyskać za pomocą algorytmu Dijkstry, wykonując go dla każdego wierzchołka grafu traktowanego jako źródło. Otrzymany w ten sposób algorytm ma złożoność  $O(n^3)$ .

Istnieją bardziej bezpośrednie algorytmy APSP, a jednym z bardziej znanych jest **algorytm Floyda-Warshalla** o podobnej złożoności czasowej, szczególnie łatwy do zaprogramowania. Podobnie jak algorytm Dijkstry, wykorzystuje on macierz kosztów, ale jej elementami mogą być również liczby ujemne. Algorytm działa w oparciu o strategię **programowania dynamicznego**, wyznaczając długości najkrótszych ścieżek w tablicy dwuwymiarowej  $n \times n$ , powiedzmy o nazwie  $D$ . Początkowo wartości elementów  $D[u, v]$  są równe wagom  $C[u, v]$ , a w kolejnych  $n$  iteracjach są korygowane:

```
for u:=1 to n do
  for v:=1 to n do
    D[u,v] := C[u,v]
  D[u,u] := 0
  for k:=1 to n do
    for u:=1 to n do
      for v:=1 to n do
        D[u,v] := min(D[u,v], D[u,k]+D[k,v])
```

Jeżeli w  $k$ -tej iteracji najkrótsza znana ścieżka wiodąca od wierzchołka  $u$  do  $v$  jest dłuższa niż ścieżka od  $u$  do  $v$  przechodząca poprzez pośredni wierzchołek  $k$  (por. rys. 4.18), to jako nowa najkrótsza ścieżka od  $u$  do  $v$  jest wybierana ta, która wiedzie przez wierzchołek  $k$ . Mówiąc dokładniej, po pierwszej iteracji wartością  $D[u, v]$  jest długość najkrótszej ścieżki wiodącej bezpośrednio od wierzchołka  $u$  do  $v$  lub pośrednio przez wierzchołek 1, po drugiej iteracji – długością najkrótszej ścieżki od  $u$  do  $v$  przechodzącej co najwyżej przez wierzchołki pośrednie ze zbioru  $\{1, 2\}$  itd. Ogólnie, po  $k$ -tej iteracji  $D[u, v]$  jest długością najkrótszej ścieżki, która wiedzie od  $u$  do  $v$  i nie zawiera innych wierzchołków pośrednich niż  $\{1, 2, \dots, k\}$ . Po zakończeniu wszystkich iteracji wartościami elementów  $D[u, v]$  są długości najkrótszych ścieżek od  $u$  do  $v$  przebiegających przez wierzchołki zbioru  $V$ .



**Rys. 4.18.** Wybór krótszej ścieżki w algorytmie Floyda-Warshalla

Odtworzenie postaci najkrótszych ścieżek wymaga, podobnie jak w algorytmie Dijkstry, użycia dodatkowej tablicy  $P$ , jednak tym razem jest ona dwuwymiarowa, inne jest też znaczenie jej elementów. I tak, zerowa wartość  $P[u, v]$  oznacza, że najkrótsza ścieżka wiodąca od  $u$  do  $v$  jest krawędzią, a różna od zera jest numerem największego wierzchołka pośredniego na najkrótszej ścieżce od  $u$  do  $v$ . Oto pełna wersja algorytmu Floyda-Warshalla:

```

procedure FLOYD-WARSHALL( $C, D, P$ )
  for  $u:=1$  to  $n$  do
    for  $v:=1$  to  $n$  do
       $D[u, v] := C[u, v]$ 
       $P[u, v] := 0$ 
       $D[u, u] := 0$ 
    for  $k:=1$  to  $n$  do
      for  $u:=1$  to  $n$  do
        for  $v:=1$  to  $n$  do
          if  $D[u, k] + D[k, v] < D[u, v]$  then
             $D[u, v] := D[u, k] + D[k, v]$ 
             $P[u, v] := k$ 

```

Przykładowe wyniki wykonania algorytmu dla grafu pokazanego na rysunku 4.14 przedstawia tabela 4.3. Aby odtworzyć najkrótszą ścieżkę od wierzchołka 1 do 5, której długość wynosi 60, odczytujemy:  $P[1, 5] = 4$ ,  $P[1, 4] = 0$ ,  $P[4, 5] = 3$ ,  $P[4, 3] = 0$  i  $P[3, 5] = 0$ . Zatem ścieżka ta wiedzie przez wierzchołki 1, 4, 3 i 5.

**Tabela 4.3.** Wynik wykonania algorytmu Floyda-Warshalla dla grafu z rys.4.14

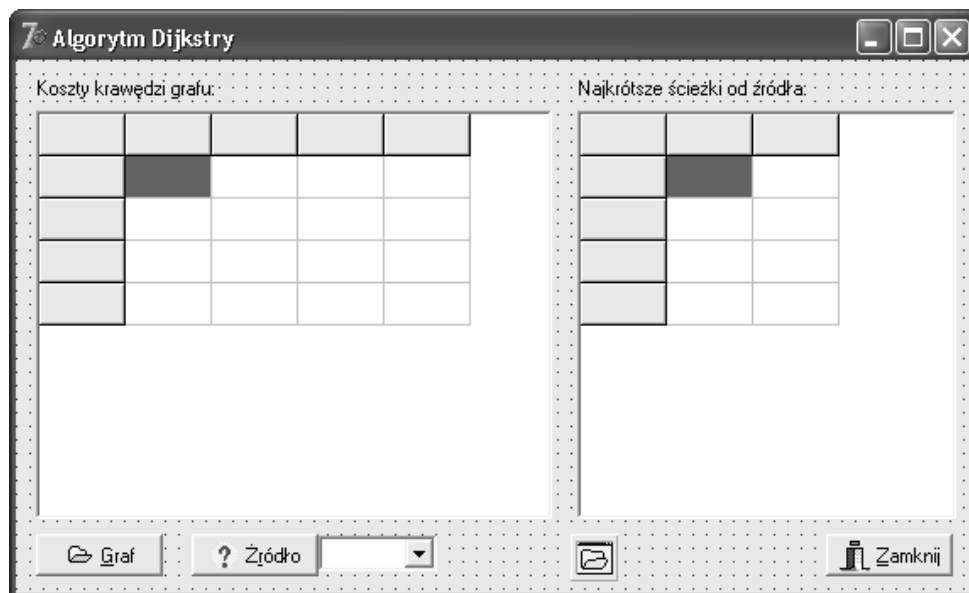
$u, v$	Macierz $D$					Macierz $P$				
	1	2	3	4	5	1	2	3	4	5
1	0	10	50	30	60	0	0	4	0	4
2	140	0	50	170	60	5	0	0	5	3
3	90	100	0	120	10	5	5	0	5	0
4	110	120	20	0	30	5	5	0	0	3
5	80	90	130	110	0	0	1	4	1	0

## 4.8. Implementacja algorytmu Dijkstry w Delphi

Omówiony w poprzednim podrozdziale algorytm Dijkstry wykorzystamy teraz w aplikacji okienkowej w Delphi. Jej zadaniem jest wczytanie z pliku tekstowego danych opisujących graf ważony  $G = (V, E)$  i wyświetlenie ich, a po każdorazowym wybraniu przez użytkownika źródła  $r$  znalezienie i wyświetlenie najkrótszych



ścieżek wiodących od  $r$  do pozostałych wierzchołków grafu. Rysunek 4.19 przedstawia projekt formularza tej aplikacji. Większą część formularza zajmują dwie siatki tekstowe *StringGrid* opisane etykietami informującymi o ich przeznaczeniu, a dolną część trzy przyciski *BitBtn* oraz komponenty *ComboBox* i *OpenDialog*. Aby użytkownik mógł w razie potrzeby zmieniać szerokość kolumn obu siatek, ich właściwości *goColSizing* (właściwość *Options*) nadajemy wartość *true*. Ponadto ustawiamy właściwość *Style* komponentu *ComboBox* na wartość *csDropDownList*, co pozwoli na wygodny wybór źródła  $r$  z wyświetlanej listy rozwijanej wszystkich wierzchołków grafu  $G$ , a właściwość *Kind* trzeciego przycisku na *bkClose*, dzięki czemu nie trzeba programować jego funkcji zamykania okna.



Rys. 4.19. Projekt formularza aplikacji znajdującej najkrótsze ścieżki w grafie

Rozbudowę kodu modułu formularza rozpoczynamy od zaprogramowania funkcji pozostałych dwóch przycisków *BitBtn* i obsługi zdarzenia *OnChange* listy rozwijanej *ComboBox*. Pierwszy przycisk ma posłużyć do wczytania grafu  $G$  (wag jego krawędzi) z pliku tekstowego i wyświetlenia grafu w siatce *StringGrid1*:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    if not OpenDialog1.Execute then Exit;
    // Wczytaj graf G z pliku tekstowego
    // Wyświetl graf G w siatce StringGrid1
end;
```

Rolą drugiego przycisku jest jedynie rozwinięcie listy *ComboBox1*, z której użytkownik może wybrać źródło *r*, polecając w ten sposób programowi wykonanie obliczeń i wyświetlenie wyników w siatce *StringGrid2*:

```

procedure TForm1.BitBtn2Click(Sender: TObject);
begin
    ComboBox1.DroppedDown := true;
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
begin
    if ComboBox1.ItemIndex < 0 then Exit;
    // Wykonaj algorytm Dijkstry dla wybranego źródła
    // Pokaż najkrótsze ścieżki w siatce StringGrid2
end;

```

Przed dalszą rozbudową aplikacji, którą nazwiemy *Dijkstra*, konieczne jest podjęcie decyzji dotyczących reprezentacji grafu w pamięci komputera i postaci pliku wejściowego. Ustalenia te precyzujemy w odrębnym module, w którym zamieścimy podprogramy czytania danych opisujących graf i wyznaczania najkrótszych ścieżek. Najpierw definiujemy w nim trzy typy tablicowe reprezentujące macierz kosztów krawędzi grafu, tablicę długości najkrótszych ścieżek i tablicę poprzedników wierzchołków, przez które wiodą te ścieżki:

```

type
    TGraph = array of array of real;
    TDists = array of real;
    TPaths = array of integer;

```

Przyjmujemy też, podobnie jak w przypadku zaprezentowanej w podrozdziale 4.5 aplikacji *Rozpin* znajdowania drzewa rozpinającego grafu, że w pierwszym wierszu pliku podany jest rodzaj grafu (litera *S* – skierowany, *N* – nieskierowany) i jego największy wierzchołek *n* (liczba całkowita), a w następnych wierszach po dwa wierzchołki *u*, *v* (liczby całkowite) określające krawędź grafu i dodatkowo jej wagę (nieujemna liczba rzeczywista). Przykładowo, plik opisujący graf przedstawiony na rysunku 4.14 zawiera następujące dane:

S	5	
1	2	10
1	4	30
1	5	90
2	3	50
3	5	10
4	3	20
4	5	50
5	1	80

Możemy teraz sprecyzować operację czytania danych z pliku tekstowego do tablicy dynamicznej  $C$  typu  $TGraph$  reprezentującej macierz kosztów krawędzi grafu. Jeżeli uwzględnimy indeksację elementów tablicy  $C$  od 0 do  $n - 1$  zamiast od 1 do  $n$  i przyjmiemy, że  $INFINITY$  jest stałą typu rzeczywistego zdefiniowaną jako  $1/0$  ( $\infty$ , plus nieskończoność)<sup>6</sup>, główną część podprogramu czytania danych z pliku możemy sformułować w następującej postaci:

```

ReadLn(Plik, z, n);
SetLength(C, n, n);
for u:=0 to n-1 do
  for v:=0 to n-1 do
    if u<>v then C[u,v] := INFINITY else C[u,v] := 0;
  while not Eof(Plik) do
    begin
      ReadLn(Plik, u, v, d);
      C[u-1,v-1] := d;
      if z = 'N' then C[v-1,u-1] := d;
    end;

```

Działanie tego kodu jest proste. Najpierw, po wczytaniu pierwszego wiersza pliku, tablicy  $C$  jest przydzielana pamięć, po czym jej elementy powyżej i poniżej głównej przekątnej są wypełniane wartością  $INFINITY$ , a elementy na przekątnej zerami. Po tej operacji tablica  $C$  reprezentuje graf ważony bez krawędzi. Następnie są wczytywane kolejne wiersze pliku określające krawędzie grafu i ich wagi, które są przypisywane odpowiednim elementom tablicy  $C$ . W rezultacie cała macierz kosztów krawędzi grafu zostanie określona na podstawie zawartości pliku.

Znowu znaleźliśmy się w sytuacji, w której powinniśmy podjąć decyzję, tym razem odnośnie reprezentacji zbioru  $S$  wierzchołków grafu w algorytmie Dijkstry. Narzucającym się rozwiązaniem jest wygodna implementacja pascalsowa **set of ...**, ale jej wadą jest ograniczenie liczby elementów zbioru do 256. Dlatego użyjemy bardziej uniwersalnej implementacji w postaci wektora charakterystycznego:

```

S: array of boolean;

```

Wyjaśnijmy, że wartość *true* elementu  $S[u]$  oznacza, że wierzchołek  $u$  należy do zbioru  $S$ , a wartość *false*, że  $u$  nie należy do  $S$ .

Przystępując do precyzowania algorytmu Dijkstry zakładamy, że dla macierzy kosztów podanej w tablicy  $C$  typu  $TGraph$  i źródła  $r$  typu *integer* podprogram ma obliczyć długości najkrótszych ścieżek w tablicy  $D$  typu  $TDists$  i zapamiętać ich postać w tablicy  $P$  typu  $TPaths$ . Jeżeli na razie pominiemy problem wyznaczania wartości elementów tablicy  $P$ , obliczenia te możemy sformułować następująco:

---

<sup>6</sup> Stała taka jest już właściwie zdefiniowana w module *Math*, ma nazwę *INF*.

```

SetLength(S, n);
SetLength(D, n);
for v:=0 to n-1 do
begin
  S[v] := false;
  D[v] := C[r,v];
end;
S[r] := true;
D[r] := 0;
for k:=2 to n do
begin
  u := -1;
  for v:=0 to n-1 do
    if not S[v] and ((u < 0) or (D[v] < D[u])) then u := v;
  S[u] := true;
  for v:=0 to n-1 do
    if not S[v] and (D[u]+C[u,v] < D[v]) then
      D[v] := D[u]+C[u,v];
  end;
end;

```

Zauważmy, że konsekwencją numeracji wierzchołków grafu od zera wzwyż jest inicjalizacja zmiennej  $u$  wartością  $-1$  przed pętlą przeglądającą wierzchołki  $v \notin S$  w poszukiwaniu wierzchołka  $u$  o minimalnej wartości  $D[u]$ . A oto pełny kod modułu obejmujący podprogram czytania grafu i algorytm Dijkstry:

```

unit DijkUnit;

interface

type
  TGraph = array of array of real;
  TDists = array of real;
  TPaths = array of integer;

const
  INFINITY = 1/0;

procedure LoadGraph(FileName: string; var C: TGraph);
procedure AlgDijkstry(var C: TGraph; r: integer;
                     var D: TDists; var P: TPaths);

implementation

procedure LoadGraph(FileName: string; var C: TGraph);
var Plik: TextFile;
    n, u, v: integer;
    d: real;
    z: char;
begin
  AssignFile(Plik, FileName);
  Reset(Plik);
  try

```

```

    ReadLn(Plik, z, n);
    SetLength(C, n, n);
    for u:=0 to n-1 do
        for v:=0 to n-1 do
            if u<>v then C[u,v] := INFINITY else C[u,v] := 0;
        while not Eof(Plik) do
            begin
                ReadLn(Plik, u, v, d);
                if (u > 0) and (u <= n) and (v > 0) and (v <= n) then
                    begin
                        C[u-1,v-1] := d;
                        if z = 'N' then C[v-1,u-1] := d;
                    end;
                end;
            finally
                CloseFile(Plik);
            end;
        end;
    end;

    procedure AlgDijkstry(var C: TGraph; r: integer;
                          var D: TDists; var P: TPaths);
    var S: array of Boolean;
        k, u, v: integer;
    begin
        SetLength(S, Length(C));
        SetLength(D, Length(C));
        SetLength(P, Length(C));
        for v:=0 to High(C) do
            begin
                S[v] := false;
                D[v] := C[r,v];
                if C[r,v] < INFINITY then P[v] := r else P[v] := -1;
            end;
        S[r] := true;
        D[r] := 0;
        P[r] := -1;
        for k:=2 to Length(C) do
            begin
                u := -1;
                for v:=0 to High(C) do
                    if not S[v] and ((u < 0) or (D[v] < D[u])) then u := v;
                S[u] := true;
                for v:=0 to High(C) do
                    if not S[v] and (D[u]+C[u,v] < D[v]) then
                        begin
                            D[v] := D[u]+C[u,v];
                            P[v] := u;
                        end;
                end;
            end;
        end;
    end.

```

Jak widać, podprogram *LoadGraph* zawiera prostą obsługę wyjątków, które mogą wystąpić podczas czytania pliku, a podprogram *AlgDijkstra*, oprócz wyznaczania długości najkrótszych ścieżek wiodących od wierzchołka *r* do wszystkich pozostałych wierzchołków grafu *G*, zapamiętuje postać tych ścieżek w tablicy *P*. Zauważmy też, że z przedstawionego wyżej powodu elementy tablicy *P* nie są inicjalizowane zerami, lecz wartością  $-1$ .

Możemy wreszcie powrócić do dalszej rozbudowy modułu formularza. I tak, listę **uses** modułu uzupełniamy o nazwę stworzonego modułu *DijkUnit*, a w klasie *TForm1* formularza umieszczamy trzy pola i dwie metody:

```
C: TGraph;
D: TDists;
P: TPaths;
procedure ShowGraph;
procedure ShowPaths;
```

Jest oczywiste, że nowe pola reprezentują odpowiednio: macierz kosztów krawędzi rozpatrywanego grafu *G*, długości najkrótszych ścieżek o wspólnym początku *r* i zbiory wierzchołków tworzących każdą z nich. Natomiast metoda *ShowGraph* ma wyświetlać macierz kosztów grafu *G* w siatce *StringGrid1*, a *ShowPaths* najkrótsze ścieżki w siatce *StringGrid2*. Wykorzystując atrapy obu tych metod i podprogramy zawarte w module *DijkUnit*, możemy już teraz zastąpić komentarze w procedurach *BitBtn1Click* i *ComboBox1Change* ostatecznym kodem. Mianowicie, procedura *BitBtn1Click* czyta graf z pliku i wyświetla go:

```
LoadGraph(OpenDialog1.FileName, C);
ShowGraph;
```

a *ComboBox1Change* znajduje najkrótsze ścieżki i wyświetla je:

```
AlgDijkstra(C, ComboBox1.ItemIndex, D, P);
ShowPaths;
```

Aby zakończyć program, należy uzupełnić wygenerowane przez Delphi puste metody *ShowGraph* i *ShowPaths*. W pierwszej dostosowujemy liczby kolumn i wierszy siatki *StringGrid1* oraz liczbę wierszy siatki *StringGrid2* do rozmiaru *n* grafu *G*, czyścimy też listę wierzchołków listy rozwijanej *ComboBox1*. Następnie wypełniamy pierwszą siatkę numerami wierzchołków i elementami macierzy *C*, a pierwszą kolumnę drugiej siatki i listę rozwijaną numerami wierzchołków:

```
StringGrid1.ColCount := n+1;
StringGrid1.RowCount := n+1;
StringGrid2.RowCount := n+1;
ComboBox1.Items.Clear;
```

```

for u:=1 to n do
begin
  StringGrid1.Rows[u].Clear;
  StringGrid1.Cells[u,0] := IntToStr(u);
  StringGrid1.Cells[0,u] := IntToStr(u);
  for v:=1 to Length(C) do
    if C[u-1,v-1] < INFINITY
      then StringGrid1.Cells[v,u] := FloatToStr(C[u-1,v-1])
      else StringGrid1.Cells[v,u] := '';
  StringGrid2.Rows[u].Clear;
  StringGrid2.Cells[0,u] := IntToStr(u);
  ComboBox1.Items.Add(IntToStr(u));
end;

```

Z kolei w metodzie *ShowPaths* drugą kolumnę siatki *StringGrid2* wypełniamy elementami tablicy *D* określającymi długości znalezionych najkrótszych ścieżek, a trzecią kolumnę listami wierzchołków, przez które te ścieżki wiodą. Przy odtwarzaniu postaci ścieżek wygodniej jest skorzystać z łańcucha zamiast stosu:

```

for u:=0 to n-1 do
begin
  if D[u] < INFINITY
    then StringGrid2.Cells[1,u+1] := FloatToStr(D[u])
    else StringGrid2.Cells[1,u+1] := '';
  v := u;
  s := IntToStr(v+1);
  repeat
    v := P[v];
    if v >= 0 then s := IntToStr(v+1) + ', ' + s;
  until v < 0;
  StringGrid2.Cells[2,u+1] := s;
end;

```

W ostatecznej wersji modułu definiujemy jeszcze procedurę obsługi zdarzenia *OnCreate* formularza, w której dokonujemy korekty szerokości niektórych kolumn siatek *StringGrid1* i *StringGrid2* oraz określamy zawartość pierwszego wiersza drugiej z nich. Oto kod tego modułu:

```

unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, Menus, StdCtrls, Grids, ComCtrls,
  Buttons, ExtCtrls, DijkUnit;

type
  TForm1 = class(TForm)

```

```

    Panel1: TPanel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    BitBtn3: TBitBtn;
    ComboBox1: TComboBox;
    StringGrid1: TStringGrid;
    StringGrid2: TStringGrid;
    Label1: TLabel;
    Label4: TLabel;
    OpenDialog1: TOpenDialog;
procedure FormCreate(Sender: TObject);
procedure ComboBox1Change(Sender: TObject);
procedure BitBtn1Click(Sender: TObject);
procedure BitBtn2Click(Sender: TObject);
private
    C: TGraph;
    D: TDists;
    P: TPaths;
    procedure ShowGraph;
    procedure ShowPaths;
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
    StringGrid1.ColWidths[0] := 24;
    StringGrid2.ColWidths[0] := 24;
    StringGrid2.ColWidths[2] :=
        StringGrid2.Width-32-StringGrid2.ColWidths[1];
    StringGrid2.Cells[1,0] := 'D';
    StringGrid2.Cells[2,0] := 'Ścieżka';
end;

procedure TForm1.ShowGraph;
var
    u, v: integer;
begin
    StringGrid1.ColCount := Length(C)+1;
    StringGrid1.RowCount := StringGrid1.ColCount;
    StringGrid2.RowCount := StringGrid1.RowCount;
    ComboBox1.Items.Clear;
    for u:=1 to Length(C) do
    begin
        StringGrid1.Rows[u].Clear;
        StringGrid1.Cells[u,0] := IntToStr(u);
    
```



```

StringGrid1.Cells[0,u] := IntToStr(u);
for v:=1 to Length(C) do
  if C[u-1,v-1] < INFINITY
    then StringGrid1.Cells[v,u] := FloatToStr(C[u-1,v-1])
    else StringGrid1.Cells[v,u] := '';
StringGrid2.Rows[u].Clear;
StringGrid2.Cells[0,u] := IntToStr(u);
ComboBox1.Items.Add(IntToStr(u));
end;
end;

procedure TForm1.ShowPaths;
var
  u, v: integer;
  s: string;
begin
  for u:=0 to High(C) do
    begin
      if D[u] < INFINITY
        then StringGrid2.Cells[1,u+1] := FloatToStr(D[u])
        else StringGrid2.Cells[1,u+1] := '';
      v := u;
      s := IntToStr(v+1);
      repeat
        v := P[v];
        if v >= 0 then s := IntToStr(v+1) + ', ' + s;
      until v < 0;
      StringGrid2.Cells[2,u+1] := s;
    end;
  end;

  procedure TForm1.ComboBox1Change(Sender: TObject);
  begin
    if ComboBox1.ItemIndex < 0 then Exit;
    AlgDijkstra(C, ComboBox1.ItemIndex, D, P);
    ShowPaths;
  end;

  procedure TForm1.BitBtn1Click(Sender: TObject);
  begin
    if not OpenFileDialog1.Execute then Exit;
    LoadGraph(OpenFileDialog1.FileName, C);
    ShowGraph;
  end;

  procedure TForm1.BitBtn2Click(Sender: TObject);
  begin
    ComboBox1.DroppedDown := true;
  end;

end.

```

Wynik wykonania aplikacji *Dijkstra* dla grafu przedstawionego na rysunku 4.14 i źródła  $r = 1$  można zobaczyć na rysunku 4.20. Przypomnijmy, że zawartość pliku opisującego ten graf została przytoczona w niniejszym podrozdziale.

# Algorytm Dijkstry

Koszty krawędzi grafu:

	1	2	3	4	5
1	0	10		30	90
2		0	50		
3			0		10
4			20	0	50
5					0

Najkrótsze ścieżki od źródła:

	D	Ścieżka
1	0	1
2	10	1, 2
3	50	1, 4, 3
4	30	1, 4
5	60	1, 4, 3, 5

Graf

Źródło

Zamknij

**Rys. 4.20.** Przykładowy wynik wykonania aplikacji znajdującej najkrótsze ścieżki w grafie za pomocą algorytmu Dijkstra

## Ćwiczenia

**4.1.** Zbuduj graf ważony reprezentujący sieć połączeń drogowych pomiędzy największymi miastami Polski, w którym wagami są odległości tych miast.

**4.2.** Opracuj w pseudojęzyku iteracyjną wersję procedury *DSF* wykorzystującą stos jawnie, a następnie iteracyjną wersję algorytmu przeszukiwania grafu w głąb.

**4.3.** Opracuj w pseudojęzyku algorytm generowania lasu rozpinającego grafu metodą przeszukiwania wszerz, a następnie utwórz jego implementację w Delphi.

**4.4.** Znajdź drzewo i las rozpinający grafów przedstawionych na rysunku 4.11 metodą przeszukiwania wszerz.

**4.5.** Zmodyfikuj przytoczoną w podrozdziale 4.5 aplikację *Rozpin* znajdowania drzewa (lasu) rozpinającego tak, by korzystała z reprezentacji grafu w postaci list sąsiedztwa zamiast macierzy sąsiedztwa.

**Wskazówka.** Użyj reprezentacji tablicowej listy, przydzielając dynamicznie każdemu elementowi jednowymiarowej tablicy wierszy jednowymiarową tablicę wierzchołków za pomocą odrębnych wywołań procedury *SetLength*.

**4.6. Stopniem wierzchołka  $u$**  grafu nazywamy liczbę krawędzi rozpoczynających się lub kończących w wierzchołku  $u$  (pętla jest uwzględniana dwukrotnie). Opracuj algorytm, który wyznacza stopnie wszystkich wierzchołków grafu i określ jego złożoność czasową.

**4.7. Stopniem wejściowym** wierzchołka  $u$  w grafie skierowanym nazywamy liczbę krawędzi, których końcem jest  $u$ , a **stopniem wyjściowym** wierzchołka  $u$  liczbę krawędzi, których początkiem jest  $u$ . Opracuj algorytm, który wyznacza stopnie wejściowe i wyjściowe wszystkich wierzchołków grafu skierowanego.

**4.8.** Napisz aplikację okienkową w Delphi znajdującą spójne składowe grafu nieskierowanego według algorytmu *DFS-COMPONENT*.

**4.9.** Zmodyfikuj algorytm *DFS-COMPONENT* znajdowania spójnych składowych grafu nieskierowanego tak, by numerami składowych były ich najmniejsze wierzchołki.

**4.10.** Opracuj algorytm znajdowania spójnych składowych grafu nieskierowanego, który działa w oparciu o strategię przechodzenia grafu wszerek, i zbuduj jego implementację w Delphi.

**4.11.** Graf skierowany  $G$  nazywamy **silnie spójnym**, jeżeli dla każdego jego dwóch wierzchołków  $u$  i  $v$  istnieją w  $G$  ścieżki wiodące od  $u$  do  $v$  i od  $v$  do  $u$ , zaś **silnie spójną składową** grafu skierowanego  $G$  nazywamy jego maksymalny, w sensie zawierania wierzchołków i krawędzi, silnie spójny podgraf. Podaj przykład grafu skierowanego, który ma kilka silnych spójnych składowych, chociaż jego rysunek sugeruje, że ma tylko jedną.

**4.12.** Graf skierowany  $G$  nazywamy **słabo spójnym**, jeżeli graf nieskierowany o takich samych wierzchołkach i krawędziach jest spójny, zaś **słabo spójną składową** grafu skierowanego  $G$  nazywamy jego maksymalny słabo spójny podgraf.

Zmodyfikuj algorytm *DFS-COMPONENT* tak, by znajdował słabo spójne składowe grafu skierowanego.

**4.13.** Zmodyfikuj algorytm *DFS-COMPONENT* tak, by znajdował silnie spójne składowe grafu nieskierowanego, i napisz jego implementację w Delphi.

**4.14.** Utwórz macierz kosztów grafu z rysunku 4.8 i znajdź w nim najkrótsze ścieżki wychodzące od różnych wierzchołków, posługując się aplikacją *Dijkstra*.

**4.15.** Zmodyfikuj algorytm Dijkstry w aplikacji *Dijkstra* tak, by implementacją zbioru *S* była zmienna typu zbiorowego (**set of ...**) w Pascalu.

**4.16.** Podaj przykład grafu ważonego, dla którego algorytm Dijkstry nie działa poprawnie.

**4.17.** Napisz aplikację okienkową w Delphi, która umożliwia znajdowanie najkrótszej drogi pomiędzy dwoma dowolnie wybranymi największymi miastami Polski za pomocą algorytmu Dijkstry (por. ćw. 4.1).

**4.18.** Zaimplementuj w Delphi algorytm Floyda-Warshalla wyznaczania najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków grafu ważonego.

**4.19.** Wykorzystując macierz *P* utworzoną przez algorytm Floyda-Warshalla, opracuj w pseudojęzyku programowania algorytm generowania listy wszystkich wierzchołków na najkrótszej ścieżce wiodącej między dwoma wskazanymi wierzchołkami grafu.

**4.20.** Napisz aplikację okienkową w Delphi, która umożliwia znajdowanie najkrótszej drogi pomiędzy dwoma dowolnie wybranymi największymi miastami Polski za pomocą algorytmu Floyda-Warshalla (por. ćw. 4.18 i 4.19).

## Rozdział 5.

### Algorytmy rekurencyjne

Rekurencja jest jedną z najważniejszych i najskuteczniejszych metod konstruowania algorytmów i programów. Algorytm lub podprogram (funkcję, procedurę) nazywamy **rekurencyjnym**, gdy odwołuje się do siebie bezpośrednio lub pośrednio. Jeżeli algorytm (lub podprogram)  $P$  zawiera bezpośrednie wywołanie samego siebie, nazywamy go **bezpośrednio rekurencyjnym**, a jeżeli wywołuje algorytm (lub podprogram)  $Q$ , który zawiera bezpośrednie lub pośrednie wywołanie  $P$ , nazywamy go **pośrednio rekurencyjnym**.

Zazwyczaj algorytmy rekurencyjne są bardziej przejrzyste i krótsze w zapisie niż ich odpowiedniki iteracyjne, jednak realizowane na komputerze w postaci podprogramów mają większą złożoność pamięciową i czasową. Każde wywołanie rekurencyjne wymaga bowiem przechowania na stosie części tzw. **środowiska wykonawczego**. Za każdym razem, gdy podprogram jest wywoływany, rezerwuje się dla niego dynamicznie obszar pamięci, zwany **ramką stosu** (ang. *stack frame*) lub **rekordem aktywacji**, w którym przechowuje się m.in. wartości parametrów, nowy zestaw zmiennych lokalnych i adres miejsca pamięci, do którego ma nastąpić powrót po wykonaniu podprogramu<sup>1</sup>. Gdy podprogram jest wywoływany, ramka stosu jest tworzona niezależnie od tego, czy podobna ramka istnieje w innym miejscu stosu dla wcześniejszego wywołania, czy nie, i jest zdejmowana ze stosu, gdy zakończy się wykonanie podprogramu dla tego wywołania i następuje powrót do miejsca wywołania. Tak więc, w przypadku wywołań rekurencyjnych powstaje odrębna ramka stosu dla każdego wywołania, przez co wzrasta zajętość pamięci, a z uwagi na konieczność zarządzania ramkami wzrasta też czas procesora.

Nie oznacza to bynajmniej, że zawsze należy unikać rekurencji. Za użyciem rekurencji przemawia czytelność i prostota algorytmu oraz rekurencyjne zdefiniowanie rozwiązywanego problemu, kiedy to wybór rekurencyjnej wersji algorytmu wydaje się szczególnie naturalny i uzasadniony. Algorytmy, w których „na siłę” unika się rekurencji, stają się najczęściej niezgrabne i nieczytelne.

Rekurencji należy unikać przede wszystkim wtedy, gdy istnieje oczywiste rozwiązanie iteracyjne, a także, gdy prowadzi ona do zbyt długiego ciągu wywołań rekurencyjnych, ponieważ grozi niebezpieczeństwem przepełnienia stosu, a jeśli nie, może znacznie spowolnić wykonywanie programu. Przykładem takiego nie-

---

<sup>1</sup> Bardziej szczegółowe informacje na temat zawartości ramki stosu można znaleźć np. w książce [8].

praktycznego podprogramu jest zaprezentowana w podrozdziale 1.8 funkcja *FIB*, która oblicza  $n$ -tą liczbę Fibonacciego. Wielokrotne powtarzanie tych samych wywołań prowadzi do zaskakująco długiego czasu obliczeń, nawet dla niewielkich wartości  $n$ , przez co jest ona praktycznie bezużyteczna. Przypomnijmy, że poprzez zastąpienie rekurencyjnej funkcji *FIB* iteracyjną funkcją *FIB2* udało się istotnie zredukować złożoność czasową obliczeń z wykładniczej do liniowej.

Rozpatrzmy inny przykład, którym jest wypisywanie słowne cyfr dziesiętnych liczby całkowitej  $n \geq 0$ . Problem można sprowadzić do wypisania cyfr wartości  $n \text{ div } 10$ , czyli wszystkich cyfr  $n$  prócz ostatniej, a potem do dopisania ostatniej cyfry liczby  $n$ . Prowadzi to do prostej procedury rekurencyjnej:

```
procedure Wypisz(n: integer);
const
  d: array[0..9] of string = ('zero', 'jeden', 'dwa', 'trzy',
    'cztery', 'pięć', 'sześć', 'siedem', 'osiem', 'dziewięć');
begin
  if n > 9 then Wypisz(n div 10);
  Write(' ', d[n mod 10]);
end;
```

Liczba wywołań rekurencyjnych w tej procedurze jest mała, a zamiana jej na postać iteracyjną wymaga wyznaczenia tablicy lub łańcucha cyfr występujących w zapisie dziesiętnym liczby albo wykorzystania konkatencji (dodawania) łańcuchów. Podprogram nie będzie jednak tak zręczny i czytelny.

W dalszej części niniejszego rozdziału przedstawimy przykłady programów rekurencyjnych, w których rekurencja jest w pełni uzasadniona. Aby przekonać się, jak skutecznym i potężnym jest ona narzędziem, wystarczy podjąć próbę opracowania odpowiedników iteracyjnych tych programów, nawet gdy wykorzystuje się znane metody derekursywacji [30].

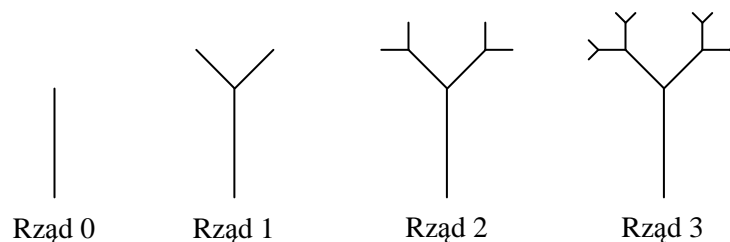
## 5.1. Drzewko

Rekurencyjność drzewek przedstawionych na rysunku 5.1 jest oczywista<sup>2</sup>. Drzewko rzędu 0 ma tylko pień (odcinek prostej). Drzewko rzędu 1 ma na górnym końcu pnia o wysokości  $h$  dwa drzewka rzędu 0 o pniach o wysokości  $h/2$  odchylonych w lewo i w prawo o kąt  $45^\circ$  (por. rys. 5.2). Z kolei drzewko rzędu 2 ma na końcu pnia dwa drzewka rzędu 1 o dwukrotnie mniejszych pniach odchylonych w lewo i w prawo o kąt  $45^\circ$ . Schemat rekurencji, na podstawie którego skonstru-

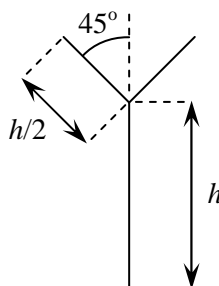
---

<sup>2</sup> Zadanie z Regionalnego Konkursu Informatycznego Politechniki Radomskiej dla uczniów szkół ponadpodstawowych w roku 1997/98.

ujemy program kreślący w Delphi, jest łatwy do odkrycia. Drzewko rzędu  $n$  ma pień o wysokości  $h$ , a na jego końcu dwa drzewka rzędu  $n - 1$  o pniach o wysokości  $h/2$  odchylonych w lewo i w prawo o kąt  $45^\circ$ .



**Rys. 5.1.** Drzewka rzędu 0, 1, 2 i 3



**Rys. 5.2.** Konstrukcja drzewka rzędu 1

Po utworzeniu nowego projektu aplikacji okienkowej budujemy prosty interfejs, wstawiając na dole formularza panel, na nim po lewej stronie pole edycyjne *SpinEdit*, a po prawej przycisk *BitBtn*. Pole edycyjne będzie służyło do określania rzędu drzewka, więc opisujemy go etykietą  $n$ , a jego właściwości *MaxValue* przypisujemy wartość 10, ograniczając tym samym (z uwagi na rozdzielczość ekranu) zakres  $n$  od 0 do 10. Przycisk ma kończyć program, toteż jego właściwość *Kind* ustawiamy na *bkClose*, a *Caption* na *Zamknij*. Pozostaje jeszcze drobna kosmetyka: usunięcie napisu na panelu i ustawienie koloru tła formularza na *clWindow*.

Sformułujemy teraz podprogram rekurencyjny kreślenia drzewka. Przyjmiemy, że jego parametrami są:  $n$  – rząd drzewka,  $x$  i  $y$  – współrzędne ekranowe podstawy pnia,  $h$  – wysokość pnia,  $k$  – kierunek kreślenia pnia (0 – północ, 1 – północny zachód, 2 – zachód itd. co  $45^\circ$ ). Podprogram ma mieć dostęp do obszaru roboczego okna i narzędzi do rysowania, więc deklarujemy go jako metodę klasy *TForm1*:

```
procedure Drzewko( $n$ ,  $x$ ,  $y$ ,  $h$ ,  $k$ : integer);
```

Kreślenie drzewka polega na ustawieniu pióra w punkcie o współrzędnych  $x$  i  $y$ , obliczeniu nowych wartości  $x$  i  $y$  (przesunięcie o  $h$  w kierunku  $k$ ), narysowaniu odcinka od aktualnej pozycji pióra do pozycji wskazanej przez  $x$  i  $y$  (pień drzewka rzędu  $n$ ), a w końcu na wykreśleniu dwóch drzewek rzędu  $n - 1$ , gdy  $n > 0$ . Uzyskujemy w ten sposób wstępne sformułowanie procedury:

```
procedure TForm1.Drzewko( $n, x, y, h, k$ : integer);
begin
    Canvas.MoveTo( $x, y$ );
    // Oblicz nowe wartości  $x$  i  $y$ 
    Canvas.LineTo( $x, y$ );
    if  $n \leq 0$  then Exit;
    Drzewko( $n-1, x, y, h \text{ div } 2, (k+1) \bmod 8$ );
    Drzewko( $n-1, x, y, h \text{ div } 2, (k+7) \bmod 8$ );
end;
```

Zwróćmy uwagę na wyrażenia określające kierunek kreślenia pni mniejszych drzewek w wywołaniach rekurencyjnych procedury *Drzewko*. Zwiększenie  $k$  o 1 oznacza obrót pnia drzewka o  $45^\circ$  w lewo, natomiast zwiększenie o 7 obrót o kąt  $7 \times 45^\circ = 315^\circ$  w lewo, czyli o kąt  $45^\circ$  w prawo. Użycie operatora **mod** ma na celu ograniczenie nowych wartości  $k$  do zakresu od 0 do 7.

Ostateczne sprecyzowanie podprogramu wymaga podania sposobu wyliczenia nowych wartości  $x$  i  $y$  w zależności od  $h$  i  $k$ . Nowe wartości  $x$  i  $y$  są przesunięte względem dotychczasowych o  $h$  w kierunku  $k$ . Dla  $k$  parzystych zmienia się tylko jedna z nich o  $h$ , a dla  $k$  nieparzystych zmieniają się obie o  $h/\sqrt{2}$ , tj. o długość boku kwadratu o przekątnej  $h$ . Najwygodniej jest rozpatrzeć każdy z ośmiu kierunków  $k$  osobno, wykorzystując instrukcję **case**. Z uwagi na naturę współrzędnych ekranowych wypada jeszcze wprowadzić małą korektę warunku kończącego rekurencję, gdyż wywołania rekurencyjne są w przypadku  $h \leq 1$  zbędne.

Metodę *Drzewko* wywołujemy w procedurze obsługi zdarzenia *OnPaint* formularza z parametrem  $n$  określonym w polu edycji *SpinEdit1*, parametrami  $x, y$  i  $h$  dopasowanymi do rozmiaru okna i parametrem  $k$  równym zero (drzewko ustawione pionowo). Pełne sformułowanie modułu formularza jest następujące:

```
unit MainUnit;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls, Spin, Buttons, ExtCtrls;

type
    TForm1 = class(TForm)
```



```

    Panel1: TPanel;
    SpinEdit1: TSpinEdit;
    Label1: TLabel;
    BitBtn1: TBitBtn;
    procedure FormPaint(Sender: TObject);
    procedure SpinEdit1Change(Sender: TObject);
public
    procedure Drzewko(n, x, y, h, k: integer);
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Drzewko(n, x, y, h, k: integer);
var d: integer;
begin
    Canvas.MoveTo(x, y);
    case k of
        0: Dec(y, h);
        1: begin
            d := Round(h/Sqrt(2)); Dec(x, d); Dec(y, d);
            end;
        2: Dec(x, h);
        3: begin
            d := Round(h/Sqrt(2)); Dec(x, d); Inc(y, d);
            end;
        4: Inc(y, h);
        5: begin
            d := Round(h/Sqrt(2)); Inc(x, d); Inc(y, d);
            end;
        6: Inc(x, h);
        7: begin
            d := Round(h/Sqrt(2)); Inc(x, d); Dec(y, d);
            end;
    end;
    Canvas.LineTo(x, y);
    if (n <= 0) or (h <= 1) then Exit;
    Drzewko(n-1, x, y, h div 2, (k+1) mod 8);
    Drzewko(n-1, x, y, h div 2, (k+7) mod 8);
end;

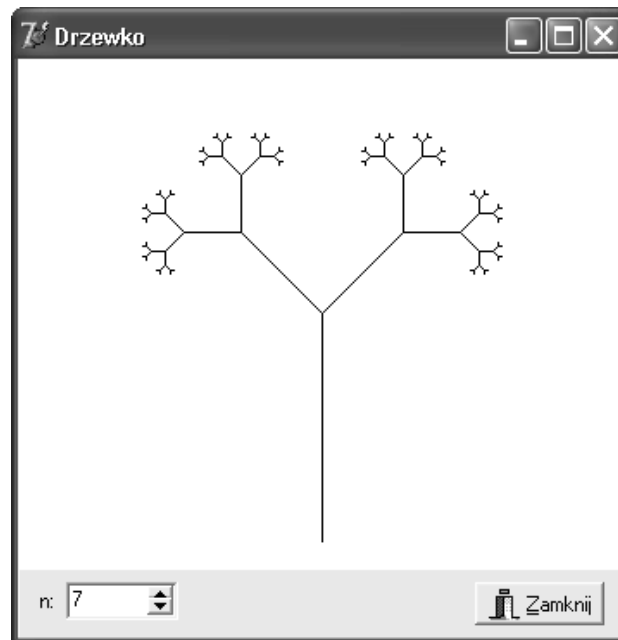
procedure TForm1.FormPaint(Sender: TObject);
var h, y: integer;
begin
    y := (ClientHeight - Panel1.Height) div 2;
    h := 9*y div 10;
    Drzewko(SpinEdit1.Value, ClientWidth div 2, y+h, h, 0);
end;

```

```
procedure TForm1.SpinEdit1Change(Sender: TObject);  
begin  
    Invalidate;  
end;  
  
end.
```

Jak widać, wymuszamy przermalowanie obszaru roboczego okna za pomocą metody *Invalidate*, gdy użytkownik zmieni w polu edycji wartość *n*. Nie możemy też przeoczyć zmiany rozmiaru okna. Po prostu obu zdarzeniom, *OnChange* pola edycji i *OnResize* formularza, przypisujemy tę samą procedurę *SpinEdit1Change*.

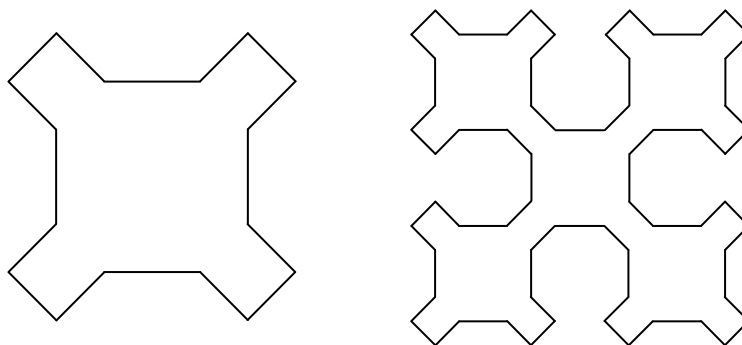
Przykładowy wynik wykonania aplikacji przedstawia rysunek 5.3.



Rys. 5.3. Drzewko rzędu 7 w oknie aplikacji

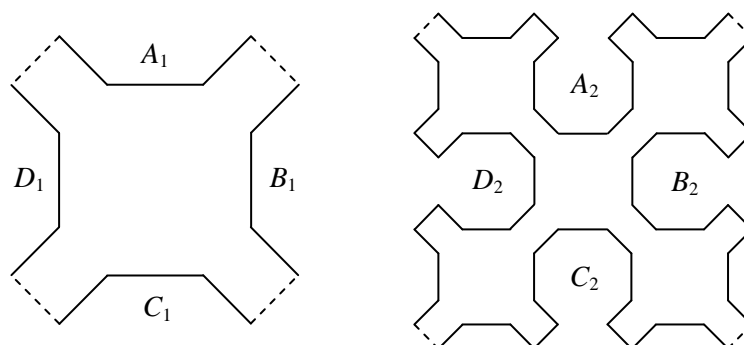
## 5.2. Krzywe Sierpińskiego

Inne atrakcyjnie wyglądające układy graficzne, które dają się wygenerować za pomocą algorytmu rekurencyjnego, przedstawiają krzywe Sierpińskiego (rys. 5.4). Przy rysowaniu tych krzywych nie korzysta się z opisujących je równań matematycznych, których być może nikt jeszcze nie odkrył, lecz ze ściśle określonego schematu rekurencyjnego sterującego narzędziem kreślącym komputera.



Rys. 5.4. Krzywe Sierpińskiego rzędu 1 i 2

Odkrycie schematu rekurencji jest istotnym elementem w konstruowaniu programu kreślącego. Rysunek 5.5 ukazuje, że krzywą Sierpińskiego daje się rozłożyć na cztery podobne krzywe otwarte i łączące je odcinki [28]. Krzywe te są oznaczone literami  $A$ ,  $B$ ,  $C$  i  $D$ , a odcinki są narysowane linią przerywaną. Zauważmy, że w przypadku pustych krzywych  $A$ ,  $B$ ,  $C$  i  $D$  pełna krzywa redukuje się do kwadratu złożonego z łączących te krzywe odcinków. Traktujemy ją jako krzywą Sierpińskiego rzędu 0.



Rys. 5.5. Schemat rekurencji kreślenia krzywych Sierpińskiego

Schemat rekurencji kreślenia krzywych  $A$ ,  $B$ ,  $C$  i  $D$  rzędu  $i$  można łatwo zdefiniować za pomocą wywołań rekurencyjnych kreślenia krzywych rzędu  $i - 1$ :

$$\begin{aligned}
 A_i &: A_{i-1} \searrow B_{i-1} \Rightarrow D_{i-1} \nearrow A_{i-1} \\
 B_i &: B_{i-1} \swarrow C_{i-1} \Downarrow A_{i-1} \searrow B_{i-1} \\
 C_i &: C_{i-1} \nwarrow D_{i-1} \Leftarrow B_{i-1} \swarrow C_{i-1} \\
 D_i &: D_{i-1} \nearrow A_{i-1} \Uparrow C_{i-1} \nwarrow D_{i-1}
 \end{aligned}$$

Strzałki wskazują kierunek ruchu pióra podczas kreślenia odcinków łączących krzywe. Jeśli oznaczymy długość jednostkową wzdłuż osi  $x$  i  $y$  przez  $h$ , to linie pojedyncze oznaczają przesunięcie pióra o  $h$  względem obu osi, czyli po przekątnej kwadratu o boku  $h$ , a linie podwójne – przesunięcie o długości  $2h$  wzdłuż osi  $x$  i  $y$ . Całą krzywą Sierpińskiego rzędu  $n$  można narysować za jednym pociągnięciem pióra według schematu:

$$A_n \searrow B_n \swarrow C_n \nwarrow D_n \nearrow$$

Jeżeli przyjmiemy, że zmienne  $x$  i  $y$  określają pozycję pióra, nietrudno jest sformułować przedstawiony wyżej schemat kreślenia krzywych  $A$ ,  $B$ ,  $C$  i  $D$  w postaci czterech procedur. I tak, procedurę kreślącą krzywą  $A$  rzędu  $i$ , odpowiadającą pierwszemu wierszowi schematu, definiujemy następująco:

```

procedure A(i: integer);
begin
  if i>0 then
    begin
      A(i-1);  x := x+h;  y := y-h;  LineTo(x, y);
      B(i-1);  x := x+2*h;          LineTo(x, y);
      D(i-1);  x := x+h;  y := y+h;  LineTo(x, y);
      A(i-1);
    end;
  end;

```

Podobnie wyglądają procedury  $B$ ,  $C$  i  $D$ . Wszystkie są bezpośrednio i pośrednio rekurencyjne. Na przykład procedura  $A$  jest bezpośrednio rekurencyjna, ponieważ wywołuje samą siebie, a pośrednio rekurencyjna, ponieważ wywołuje procedurę  $B$ , a ta z kolei wywołuje procedurę  $A$ <sup>3</sup>.

Nietrudno zauważyć, że krzywa Sierpińskiego rzędu 0 mieści się w kwadracie o boku  $2h$ , krzywa rzędu 1 –  $6h$ , a krzywa rzędu 2 –  $14h$ . Można łatwo wykazać, że krzywą rzędu  $n$  daje się zamknąć w kwadracie o boku:

$$a = h \cdot (2^{n+2} - 2) \quad (5.1)$$

Przejdźmy teraz do budowy aplikacji okienkowej w Delphi rysującej krzywe Sierpińskiego. Zaczynamy od wstawienia do projektu formularza okna głównego aplikacji komponentu *MainMenu* z dwoma pozycjami *Parametry* i *Koniec*, nadania właściwości *Color* formularza wartości *clWindow* i dodania do klasy *TForm1* pól

<sup>3</sup> Mamy tu sytuację patową, ponieważ definicja nazwy musi poprzedzać jej wykorzystanie. Wyjściem w Pascalu jest użycie deklaracji wyprzedzającej **forward** [14, 25], a w językach C i C++ skorzystanie z prototypów (nagłówek) funkcji [11]. Tutaj ominiemy problem, definiując podprogramy  $A$ ,  $B$ ,  $C$  i  $D$  jako metody klasy formularza.

$n$ ,  $h$ ,  $dx$  i  $dy$ . Pola  $dx$  i  $dy$  określają współrzędne lewego górnego rogu kwadratu obejmującego krzywą. Przyjmujemy, że krzywa może nie zmieścić się w obszarze roboczym okna, co wymaga obsługi **formularza wirtualnego** [5, 6], na którym jest rysowana cała krzywa. Rozmiar takiego formularza określa właściwość *Range* pasków przewijania *HorzScrollBar* i *VertScrollBar* (poziomego i pionowego). Jeżeli formularz wirtualny jest większy niż obszar roboczy okna, widzimy tylko jego część, a inną część możemy zobaczyć po przewinięciu okna. Aby okno było przewijane na bieżąco, gdy użytkownik będzie przeciągał suwaki pasków, ich właściwościom *Tracking* przypisujemy wartość *true*. Rozmiar formularza wirtualnego wyliczamy w nowej metodzie klasy *TForm1*, dostosowując go do wielkości krzywej na podstawie wzoru (5.1) i uwzględniając dodatkowe 16 pikseli na marginesy pomiędzy krzywą a brzegiem formularza:

```
procedure TForm1.SetParams;
var
  a: integer;
begin
  a := h*(1 shl (n+2) - 2);
  HorzScrollBar.Range := Max(ClientWidth, a+16);
  VertScrollBar.Range := Max(ClientHeight, a+16);
  dx := (HorzScrollBar.Range - a) div 2;
  dy := (VertScrollBar.Range - a) div 2;
end;
```

Przyjmujemy, że pierwszą krzywą, która ukaże się w oknie aplikacji, będzie krzywa rzędu 1. W tym celu definiujemy procedurę obsługi zdarzenia *OnCreate* formularza, w której nadajemy odpowiednie wartości początkowe polom  $n$  i  $h$ , a wartości pól  $dx$  i  $dy$  wyliczamy za pomocą metody *SetParams*:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  n := 1;
  h := Min(ClientWidth, ClientHeight) div 7;
  SetParams;
end;
```

Jest oczywiste, że gdy użytkownik zmieni rozmiar formularza, właściwości *Range* pasków przewijania ulegają zmianie. Zatem metoda *SetParams* jest również przydatna w procedurze obsługi zdarzenia *OnResize* formularza, której postać jest następująca:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  SetParams;
  Invalidate;
end;
```

W kolejnym kroku precyzowania programu zajmiemy się rysowaniem krzywej. Rozpoczynamy od przekształcenia omówionych wyżej procedur *A*, *B*, *C* i *D* schematu kreślenia na metody klasy *TForm1*. Zanim uzupełnimy wygenerowane przez Delphi puste szkielety tych metod wyjaśnijmy, że obiekt *Canvas* ma właściwość *PenPos* typu *TPoint* (rekord złożony z pól *x* i *y*), która określa aktualną pozycję pióra. W związku z tym nie musimy utrzymywać własnych zmiennych *x* i *y* do pamiętania pozycji pióra. Przypomnijmy również, że oś *y* jest na ekranie skierowana w dół. Konsekwencją tego jest zmiana znaku przyrostu współrzędnej *y* na przeciwny. Możemy teraz bez trudu sformułować metodę *A*:

```

procedure TForm1.A(i: integer);
begin
    if i>0 then with Canvas do
        begin
            A(i-1);  LineTo(PenPos.X+h, PenPos.Y+h);
            B(i-1);  LineTo(PenPos.X+2*h, PenPos.Y);
            D(i-1);  LineTo(PenPos.X+h, PenPos.Y-h);
            A(i-1);
        end;
    end;

```

Analogicznie tworzymy metody *B*, *C* i *D* klasy *TForm1*. Aby narysować całą krzywą, definiujemy procedurę obsługi zdarzenia *OnPaint* formularza, w której ustawiamy pióro w pozycji o współrzędnych  $x = dx + h$  i  $y = dy$ , a następnie kreślimy kolejno krzywe *A*, *B*, *C* i *D* rzędu *n* wraz z łączącymi je odcinkami:

```

procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
        begin
            MoveTo(dx+h, dy);
            A(n);  LineTo(PenPos.X+h, PenPos.Y+h);
            B(n);  LineTo(PenPos.X-h, PenPos.Y+h);
            C(n);  LineTo(PenPos.X-h, PenPos.Y-h);
            D(n);  LineTo(PenPos.X+h, PenPos.Y-h);
        end;
    end;

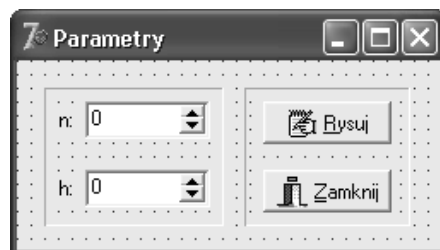
```

Gdy teraz uruchomimy aplikację, zobaczymy na ekranie krzywą Sierpińskiego rzędu 1, lecz gdy zmniejszymy okno i zaczniemy je przewijać, rysunek nie będzie wyświetlany poprawnie. Problem tkwi w przesunięciu lewego górnego rogu obszaru roboczego okna względem punktu (0, 0) formularza wirtualnego. Gdy przewiniemy okno, oba punkty zajmą różne pozycje, podczas gdy obszar roboczy jest przemasowywany tak, jakby znajdowały się w tym samym miejscu. Na szczęście przesunięcie to jest znane – określają je bieżące ustawienia właściwości *Position* pasków przewijania. Aby obraz dał się przewijać, wystarczy poinformować Win-

dows, jakie jest przesunięcie lewego górnego rogu obszaru roboczego okna względem punktu (0, 0) formularza wirtualnego. W tym celu na początku procedury *FormPaint* wywołujemy funkcję API (ang. *Application Programming Interface*) o nazwie *SetWindowOrgEx*:<sup>4</sup>

```
SetWindowOrgEx(Canvas.Handle,  
HorzScrollBar.Position, VertScrollBar.Position, nil);
```

Naszym zadaniem jest teraz umożliwienie użytkownikowi zmiany parametrów  $n$  i  $h$  krzywej Sierpińskiego. Posłużymy się podrzędnym formularzem, w którym wartości parametrów będzie można modyfikować. Po dodaniu nowego formularza do aplikacji wstawiamy do niego dwa pola edycji *SpinEdit*, opisując je etykietami  $n$  i  $h$ , oraz po dwa przyciski *BitBtn* i ramki *Bevel* (por. rys. 5.6). Z uwagi na rozdzielczość ekranu ograniczamy wartości  $n$  do zakresu od 0 do 8, nadając właściwości *MaxValue* pierwszego pola edycji wartość 8, natomiast właściwości *MinValue* drugiego pola edycji przypisujemy wartość 1. Na koniec ustawiamy właściwość *Kind* drugiego przycisku na *bkClose*.



Rys. 5.6. Projekt podrzędnego formularza parametrów

Zważywszy na wygodę użytkownika zakładamy, że gdy naciśnie on przycisk *Rysuj*, nastąpi przekazanie parametrów  $n$  i  $h$  z podrzędnego okna aplikacji do jej głównego okna, które natychmiast dostosuje się do nowych warunków, ukazując krzywą o żądanych parametrach. Wykorzystamy w tym celu funkcję API Windows o nazwie *SendMessage*, wysyłając za jej pomocą komunikat o numerze *wm\_User* i przekazując w nim wartości obu pól edycji określone przez użytkownika<sup>5</sup>. Pełny kod modułu podrzędnego formularza wygląda następująco:

<sup>4</sup> Pierwszym parametrem *SetWindowOrgEx* jest uchwyt urządzenia graficznego [6, 20], którym tu jest ekran, a dokładniej powierzchnia rysowania (*Canvas*) okna.

<sup>5</sup> Zamiast funkcji *SendMessage*, która omija kolejkę komunikatów Windows, wysyłając komunikat bezpośrednio do docelowego odbiorcy, można użyć funkcji *PostMessage*, która umieszcza komunikat w kolejce. Komunikat ten zostanie obsłużony dopiero po obsłużeniu wcześniejszych komunikatów [5, 6].

```

unit ParmUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, Spin, ExtCtrls;

type
  TForm2 = class(TForm)
    SpinEdit1: TSpinEdit;
    SpinEdit2: TSpinEdit;
    Label1: TLabel;
    Label2: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    Bevel1: TBevel;
    Bevel2: TBevel;
    procedure BitBtn1Click(Sender: TObject);
  end;

var
  Form2: TForm2;

implementation

{$R *.dfm}

procedure TForm2.BitBtn1Click(Sender: TObject);
begin
  SendMessage(Application.MainForm.Handle, wm_User,
    SpinEdit1.Value, SpinEdit2.Value);
end;

end.

```

Możemy już wrócić do modułu głównego formularza, w którym rozszerzamy listę **uses** o nazwę *ParmUnit* i definiujemy procedury obsługi zdarzeń *OnClick* dwóch poleceń menu. Jest oczywiste, że drugie polecenie ma jedynie wywołać metodę *Close*. Pierwsze natomiast ma ustawić odpowiednie właściwości pól edycji podrzędnego okna i wymusić jego ukazanie się, gdy jest niewidoczne:

```

procedure TForm1.Parametry1Click(Sender: TObject);
begin
  Form2.SpinEdit1.Value := n;
  Form2.SpinEdit2.MaxValue :=
    Max(h, Max(ClientWidth, ClientHeight) div 7);
  Form2.SpinEdit2.Value := h;
  Form2.Show;
end;

```



Ostatnim krokiem tworzenia aplikacji jest zdefiniowanie obsługi komunikatu *wm\_User* przychodzącego od podrzędnego okna. Procedury obsługi komunikatów Windows wymagają zadeklarowania w klasie formularza, mają dokładnie jeden parametr typu *TMessage* przekazywany przez zmienną, a ich deklaracje zawierają słowo kluczowe **message** i numer komunikatu [5, 14]. Zatem procedurę obsługi komunikatu *wm\_User* deklarujemy jako metodę klasy *TForm1* następująco:

```
procedure NewParams(var Msg: TMessage); message wm_User;
```

Jej zadaniem jest pobranie nowych wartości parametrów *n* i *h*, przekazanych przez komunikat w polach *WParam* i *LParam* rekordu *Msg*, dostosowanie pasków przewijania do rozmiaru formularza wirtualnego i narysowanie krzywej. Możemy więc pusty szkielet metody uzupełnić instrukcjami:

```
n := Msg.WParam;
h := Msg.LParam;
FormResize(Self);
```

Oto uzyskany w ten sposób pełny kod modułu głównego formularza:

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls, Menus, Math,
  ParmUnit;

type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    Parametry1: TMenuItem;
    Koniec1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure Parametry1Click(Sender: TObject);
    procedure Koniec1Click(Sender: TObject);
  private
    n, h, dx, dy: integer;
    procedure A(i: integer);
    procedure B(i: integer);
    procedure C(i: integer);
    procedure D(i: integer);
    procedure SetParams;
    procedure NewParams(var Msg: TMessage); message wm_User;
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.A(i: integer);
begin
  if i>0 then with Canvas do
    begin
      A(i-1); LineTo(PenPos.X+h, PenPos.Y+h);
      B(i-1); LineTo(PenPos.X+2*h, PenPos.Y);
      D(i-1); LineTo(PenPos.X+h, PenPos.Y-h);
      A(i-1);
    end;
  end;

procedure TForm1.B(i: integer);
begin
  if i>0 then with Canvas do
    begin
      B(i-1); LineTo(PenPos.X-h, PenPos.Y+h);
      C(i-1); LineTo(PenPos.X, PenPos.Y+2*h);
      A(i-1); LineTo(PenPos.X+h, PenPos.Y+h);
      B(i-1);
    end;
  end;

procedure TForm1.C(i: integer);
begin
  if i>0 then with Canvas do
    begin
      C(i-1); LineTo(PenPos.X-h, PenPos.Y-h);
      D(i-1); LineTo(PenPos.X-2*h, PenPos.Y);
      B(i-1); LineTo(PenPos.X-h, PenPos.Y+h);
      C(i-1);
    end;
  end;

procedure TForm1.D(i: integer);
begin
  if i>0 then with Canvas do
    begin
      D(i-1); LineTo(PenPos.X+h, PenPos.Y-h);
      A(i-1); LineTo(PenPos.X, PenPos.Y-2*h);
      C(i-1); LineTo(PenPos.X-h, PenPos.Y-h);
      D(i-1);
    end;
  end;
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    n := 1;
    h := Min(ClientWidth, ClientHeight) div 7;
    SetParams;
end;

procedure TForm1.SetParams;
var
    a: integer;
begin
    a := h*(1 shl (n+2) - 2);
    HorzScrollBar.Range := Max(ClientWidth, a+16);
    VertScrollBar.Range := Max(ClientHeight, a+16);
    dx := (HorzScrollBar.Range - a) div 2;
    dy := (VertScrollBar.Range - a) div 2;
end;

procedure TForm1.FormResize(Sender: TObject);
begin
    SetParams;
    Invalidate;
end;

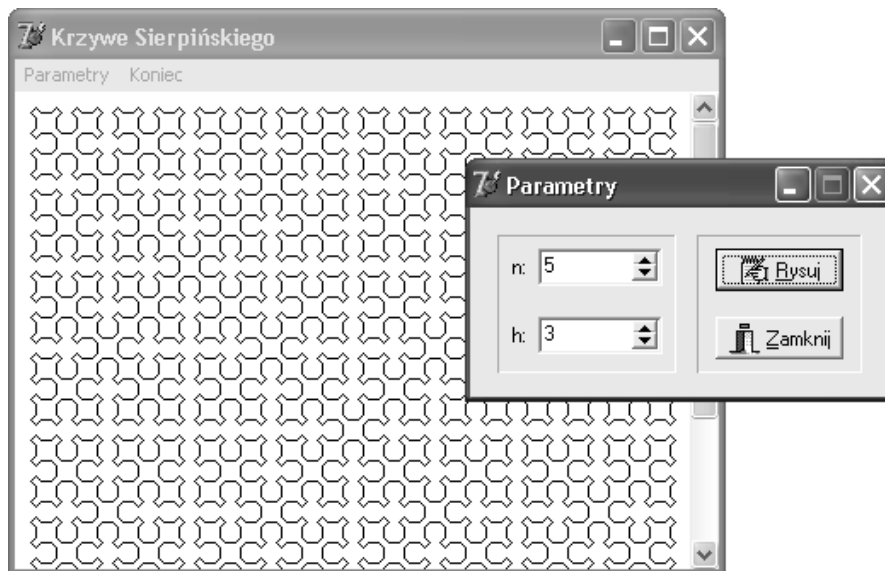
procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
    begin
        SetWindowOrgEx(Handle,
            HorzScrollBar.Position, VertScrollBar.Position, nil);
        MoveTo(dx+h, dy);
        A(n); LineTo(PenPos.X+h, PenPos.Y+h);
        B(n); LineTo(PenPos.X-h, PenPos.Y+h);
        C(n); LineTo(PenPos.X-h, PenPos.Y-h);
        D(n); LineTo(PenPos.X+h, PenPos.Y-h);
    end;
end;

procedure TForm1.Parametry1Click(Sender: TObject);
begin
    Form2.SpinEdit1.Value := n;
    Form2.SpinEdit2.MaxValue :=
        Max(h, Max(ClientWidth, ClientHeight) div 7);
    Form2.SpinEdit2.Value := h;
    Form2.Show;
end;

procedure TForm1.Koniec1Click(Sender: TObject);
begin
    Close;
end;
```

```
procedure TForm1.NewParams(var Msg: TMessage);  
begin  
    n := Msg.WParam;  
    h := Msg.LParam;  
    FormResize(Self);  
end;  
  
end.
```

Możemy wreszcie uruchomić aplikację i sprawdzić jej zachowanie dla różnych wartości  $n$  i  $h$ . Efekt jej wykonania dla  $n = 5$  i  $h = 3$  jest pokazany na rysunku 5.7.



Rys. 5.7. Okno główne z krzywą Sierpińskiego rzędu 5 i okno parametrów

### 5.3. Problem plecakowy

Niekiedy dla postawionego problemu trudno jest znaleźć algorytm, który prowadzi do rozwiązania za pomocą określonej reguły obliczeniowej. Zwykle najwygodniej jest wówczas szukać rozwiązania **metodą prób i błędów**. Algorytmy takie są nazywane algorytmami **z powrotami** lub **z nawrotami**. Charakteryzują się one tym, że kolejne kroki, które mogą prowadzić do rozwiązania, są zapisywane, a gdy później okaże się, że jakiś krok nie prowadzi do rozwiązania, następuje usunięcie jego zapisu i wycofanie się do stanu poprzedzającego błędną decyzję. Najbardziej naturalnym opisem algorytmów z powrotami jest rekurencja.

Technikę wykorzystywaną w algorytmach z powrotami zilustrujemy teraz przy rozwiązywaniu tzw. **problemu plecakowego**, a konkretnie na przykładzie Alibaby, który znalazł w Sezamie skarb i ma wynieść jego część o jak najwyższej wartości<sup>6</sup>. Skarb składa się z przedmiotów, z których każdy ma określoną wagę i wartość. Kłopot Alibaby polega na tym, że może odbyć drogę powrotną tylko raz, a plecak, który ma ze sobą, porwie się, jeśli łączna waga zabranych przedmiotów przekroczy pewien dopuszczalny ciężar.

Zauważmy, że zastosowanie algorytmu zachłannego, opartego na dokonywaniu w każdym kroku wyboru, który wydaje się być w danym momencie najlepszy, może nie doprowadzić do znalezienia optymalnego rozwiązania. Strategia działania zachłannego polega tu na wybieraniu przedmiotów najcenniejszych. Jeśli np. ładowność plecaka wynosiłaby 50, to w przypadku skarbu złożonego z trzech przedmiotów o wadze 25, 42, 20 i wartości 40, 68, 56 Alibaba wybrałaby jeden przedmiot o wadze 42 i wartości 68 zamiast dwóch przedmiotów o łącznym ciężarze 45 i wartości 96.

Właściwe rozwiązanie polega na systematycznym podejmowaniu kolejnych prób dobierania przedmiotów, aż wszystkie dopuszczalne kombinacje, w których każdy przedmiot może być pozostawiony w Sezamie albo zabrany przez Alibabę, zostaną wyczerpane. Jeżeli przedmioty ponumerujemy kolejnymi liczbami całkowitymi od 1 wzwyż, wstępną wersję algorytmu, próbującego wykonać kolejny krok dla przedmiotu  $k$ , możemy sformułować następująco:

```
procedure Próbuje(k)
  if jest następny przedmiot do zabrania then Próbuje(k+1)
  if można dołożyć do plecaka przedmiot k then
    zapisz, że Alibaba zabiera przedmiot k
    if wartość wynoszonej części skarbu jest wyższa then
      zapamiętaj zestaw i wartość wynoszonej części skarbu
    if jest następny przedmiot do zabrania then Próbuje(k+1)
    usuń zapis, że Alibaba zabiera przedmiot k
```

W algorytmie występują dwa bezpośrednie wywołania rekurencyjne. Pierwsze oznacza przejście do następnego przedmiotu bez zabierania przedmiotu  $k$ . Drugie zachodzi wtedy, gdy Alibaba może jeszcze do wynoszonej części skarbu dołożyć przedmiot  $k$ . Jeżeli polepsza to wartość wynoszonej części skarbu, nowy zestaw zgromadzonych przedmiotów i ich wartość zostaje zapamiętana.

W celu uściślenia algorytmu użyjemy następujących zmiennych, w których są przechowywane dane oraz wyniki pośrednie i końcowe:

---

<sup>6</sup> Zadanie z Regionalnego Konkursu Informatycznego Politechniki Radomskiej dla uczniów szkół ponadpodstawowych w roku 1996/97. Można spotkać inną wersję zadania, w której mówi się o włamującym się do jubilera złodzieju.

```

n: integer;           // Liczba przedmiotów
w,                   // Wagi przedmiotów
p: array[1..n] of real; // Wartości przedmiotów
cMax,                // Dopuszczalna ładowność plecaka

c,                   // Ciężar bieżącego zestawu
v: real;             // Wartość bieżącego zestawu
s: array[1..n] of Boolean; // Skład bieżącego zestawu
cOpt,                // Ciężar optymalnego zestawu
vOpt: real;          // Wartość optymalnego zestawu
sOpt: array[1..n] of Boolean; // Skład optymalnego zestawu

```

Uściślony algorytm podejmowania próby dla przedmiotu  $k$  możemy teraz zapisać w postaci procedury w języku Pascal:

```

procedure Probuje(k: integer);
begin
  if k < n then Probuje(k+1);
  c := c + w[k];
  if c <= cMax then
    begin
      v := v + p[k];
      s[k] := true;
      if v > vOpt then
        begin
          cOpt := c;
          vOpt := v;
          sOpt := s;
        end;
      if k < n then Probuje(k+1);
      s[k] := false;
      v := v - p[k];
    end;
  c := c - w[k];
end;

```

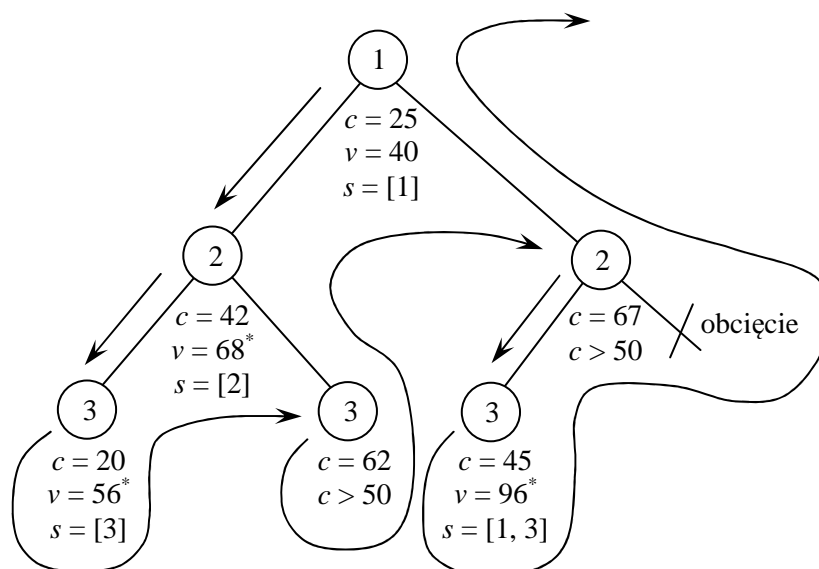
Jest oczywiste, że na początku należy wczytać dane do pierwszych czterech zmiennych. Następnie trzeba pozostałe zmienne zainicjalizować przyjmując, że zestawy zebranych przedmiotów (bieżący i optymalny) są puste, a ich ciężary i wartości zerowe, po czym wykonać procedurę *Probuje* dla przedmiotu 1:

```

for k:=1 to n do
  s[k] := false;
sOpt := s;
c := 0;
cOpt := 0;
v := 0;
vOpt := 0;
Probuje(1);

```

Można teraz prześledzić działanie algorytmu na przytoczonym wyżej przykładzie  $n = 3$ ,  $w = 25, 42, 20$ ,  $p = 40, 68, 56$  i  $cMax = 50$ . Prosta analiza prowadzi do przedstawionego na rysunku 5.8 binarnego drzewa przeszukiwań.



**Rys. 5.8.** Przykładowe drzewo przeszukiwań algorytmu

W wierzchołkach drzewa są wpisane wartości parametru  $k$  kolejnych wywołań procedury *Probuji*. Linie zakończone strzałkami wskazują kierunek wędrowki po wierzchołkach wzdłuż krawędzi drzewa. Lewe krawędzie odpowiadają przejściu do następnego przedmiotu bez zabierania przedmiotu określonego w wierzchołku, zaś prawe – przejściu po zabraniu tego przedmiotu. Obcięcie krawędzi oznacza, że wędrowka wzdłuż niej jest niemożliwa ze względu na przekroczenie dopuszczalnej ładowności plecaka. Etykiety pod wierzchołkami przedstawiają rozważane podczas dobierania nowego przedmiotu potencjalne rozwiązanie problemu (ciężar, wartość i zestaw zebranych przedmiotów). Gwiazdka oznacza, że jest ono dotąd najlepsze i jako takie zostaje zapamiętane w zmiennych  $cOpt$ ,  $vOpt$  i  $sOpt$ . Ostatnia napotkana na drodze gwiazdka określa rozwiązanie optymalne.

To, że algorytm rozwiązywania problemu plecakowego przeszukuje obcięte drzewo binarne w głąb, nie jest dziełem przypadku, ponieważ działanie algorytmów z powrotami polega na podziale zadania na podzadania i przeszukiwaniu drzewa podzadań w głąb. Chociaż drzewo przeszukiwań może być bardzo rozbudowane, najczęściej jest podczas kolejnych prób stopniowo obcinane, przez co liczba kroków redukuje się i algorytm daje się wykonać w rozsądnym czasie.

Na koniec przedstawmy jeszcze pełną aplikację konsolową w Delphi, która czyta dane dla problemu plecakowego z pliku tekstowego i wypisuje je wraz ze znalezionym rozwiązaniem. Ze względu na wykorzystanie tablic dynamicznych numerujemy w niej przedmioty od 0 do  $n - 1$  zamiast od 1 do  $n$ .

```

program Alibaba;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  n: integer;                // Liczba przedmiotów
  w, p: array of real;       // Wagi i wartości przedmiotów
  cMax,                        // Dopuszczalna ładowność plecaka
  c, v,                        // Ciężar i wartość bież. zestawu
  cOpt, vOpt: real;          // Ciężar i wartość opt. zestawu
  s, sOpt: array of Boolean; // Skład bież. i opt. zestawu

procedure Dane;
var
  Plik: TextFile;
  Nazwa: string;
  k: integer;
begin
  Write('Nazwa pliku: '); ReadLn(Nazwa);
  AssignFile(Plik, Nazwa); Reset(Plik);
  ReadLn(Plik, n);
  SetLength(w, n); SetLength(p, n);
  WriteLn('-----');
  WriteLn('Przedmiot  Waga      Wartość');
  WriteLn('-----');
  for k:=0 to n-1 do
  begin
    ReadLn(Plik, w[k], p[k]);
    WriteLn(k+1:5, w[k]:10:2, p[k]:10:2);
  end;
  ReadLn(Plik, cMax);
  WriteLn('-----');
  WriteLn('Ładowność plecaka: ', cMax:6:2);
  CloseFile(Plik);
end;

procedure Zanotuj;
begin
  cOpt := c;
  vOpt := v;
  sOpt := Copy(s, 0, n);
end;

```



```

procedure Start;
var
    k: integer;
begin
    SetLength(s, n);
    for k:=0 to n-1 do s[k]:=false;
    c := 0;
    v := 0;
    Zanotuj;
end;

procedure Wynik;
var
    k: integer;
begin
    WriteLn;
    WriteLn('Zestaw optymalny');
    WriteLn('-----');
    WriteLn('Przedmiot   Waga       Wartość');
    WriteLn('-----');
    for k:=0 to n-1 do
        if sOpt[k] then WriteLn(k+1:5, w[k]:10:2, p[k]:10:2);
    WriteLn('-----');
    WriteLn('Optimum:', cOpt:7:2, vOpt:10:2);
end;

procedure Probuje(k: integer);
begin
    if k < n-1 then Probuje(k+1);
    c := c + w[k];
    if c <= cMax then
        begin
            v := v + p[k];
            s[k] := true;
            if v > vOpt then Zanotuj;
            if k < n-1 then Probuje(k+1);
            s[k] := false;
            v := v - p[k];
        end;
    c := c - w[k];
end;

begin
    Dane;    Start;
    Probuje(0);
    Wynik;   ReadLn;
end.

```

Zadaniem procedury *Zanotuj* jest zapamiętanie napotkanego właśnie najlepszego zestawu przedmiotów na odbytej już części drogi poszukiwań po wierzchołkach drzewa, a procedury *Start* – inicjalizacja zmiennych reprezentujących zebrane

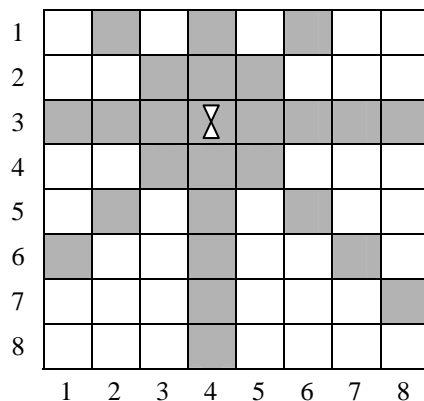
zestawy przedmiotów (bieżący i optymalny). Przyjeliśmy, że w pierwszym wierszu pliku podana jest liczba przedmiotów, w następnych wierszach po dwie liczby określające wagę i wartość kolejnego przedmiotu, a w ostatnim wierszu ładowność plecaka. Przykładowo, dla pliku postaci:

```
5
23 45
15 60
21 17
13 28
21 51
50
```

program znajduje rozwiązanie obejmujące trzy przedmioty o numerach 2, 4, 5, wagach 15, 13, 21 i wartościach 60, 28, 51. Łączny ciężar wynoszonej z Sezamu przez Alibabę części skarbu wynosi 49, a wartość 139.

## 5.4. Problem ośmiu hetmanów

Zgodnie z regułami szachowymi hetman szachuje inne figury ustawione na szachownicy w tym samym wierszu, kolumnie i na przekątnych, na których stoi (szare pola na rys. 5.9). Problem ośmiu hetmanów polega na ustawieniu na szachownicy ośmiu hetmanów tak, aby żaden nie szachował innego<sup>7</sup>.



**Rys. 5.9.** Pola szachowane przez hetmana

<sup>7</sup> Warto wspomnieć, że problemem ośmiu hetmanów zajmował się znakomity niemiecki matematyk Carl Friedrich Gauss (1777 – 1855), ale nie udało mu się znaleźć pełnego rozwiązania [28], niewątpliwie z powodu braku komputera.

Problem ośmiu hetmanów można łatwo rozwiązać na komputerze przy wykorzystaniu algorytmu z powrotami [8, 19, 28]. Jest oczywiste, że w jednej kolumnie na szachownicy możemy ustawić tylko jednego hetmana. Wynika stąd, że po ustawieniu  $k - 1$  hetmanów w kolumnach od 1 do  $k - 1$  należy podjąć próbę znalezienia w kolumnie  $k$  takiego wiersza  $w$ , żeby pole na skrzyżowaniu tej kolumny i wiersza nie było szachowane przez ustawionych już  $k - 1$  hetmanów. Rozumowanie to prowadzi do wstępnej wersji algorytmu:

```

procedure Próbuje( $k$ )
  for  $w:=1$  to 8 do
    if pole ( $w$ ,  $k$ ) nie jest szachowane then
      ustaw hetmana w polu ( $w$ ,  $k$ )
      if  $k < 8$  then Próbuje( $k+1$ )
      else zanotuj ustawienie wszystkich hetmanów
      usuń hetmana z pola ( $w$ ,  $k$ )

```

Znaleźliśmy się w sytuacji, w której powinniśmy podjąć decyzje dotyczące reprezentowania danych. Narzucającą się reprezentacją szachownicy jest tablica dwuwymiarowa zadeklarowana następująco:

```

var  $p[1..8, 1..8]$  of Boolean;

```

w której wartość *true* elementu  $p[w, k]$  oznacza, że na skrzyżowaniu kolumny  $k$  i wiersza  $w$  nie jest ustawiony hetman, a wartość *false*, że jest. W przypadku takiej reprezentacji danych sprawdzenie, czy pole, w którym ma być ustawiony hetman, nie jest szachowane, można sprecyzować w postaci podprogramu funkcyjnego. Zważywszy jednak na fakt, że będzie to najczęściej wykonywana operacja, warto pokusić się o lepszą reprezentację danych, która pozwalałaby na bardziej bezpośredni test, czy w danym polu wolno ustawić hetmana.

Interesującą reprezentację danych zaproponował Niklaus Wirth [28]. Wykorzystując fakt, że wszystkie pola leżące na przekątnej  $\swarrow$  szachownicy charakteryzują się stałą wartością  $k + w$ , a pola na przekątnej  $\searrow$  stałą wartością  $k - w$ , użył czterech tablic do określenia pozycji hetmana i stanu pól:

```

var  $x$ : array[1..8] of integer;
     $a$ : array[1..8] of Boolean;
     $b$ : array[2..16] of Boolean;
     $c$ : array[-7..7] of Boolean;

```

Znaczenie elementów tych tablic jest następujące:

- $x[k]$  oznacza pozycję (wiersz) hetmana w kolumnie  $k$ ,
- $a[w]$  oznacza brak hetmana w wierszu  $w$ ,
- $b[p]$  oznacza brak hetmana na przekątnej  $p$  o kierunku  $\swarrow$ ,
- $c[p]$  oznacza brak hetmana na przekątnej  $p$  o kierunku  $\searrow$ .

Nietrudno zauważyć, że test sprawdzający, czy pole na skrzyżowaniu kolumny  $k$  i wiersza  $w$  nie jest szachowane, sprowadza się teraz do obliczenia wartości wyrażenia logicznego:

$a[w] \text{ and } b[k+w] \text{ and } c[k-w]$

W prosty sposób może być również sprecyzowana operacja ustawienia w tym polu hetmana:

$x[k] := w;$   
 $a[w] := \text{false}; \quad b[k+w] := \text{false}; \quad c[k-w] := \text{false};$

oraz operacja usunięcia go:

$a[w] := \text{true}; \quad b[k+w] := \text{true}; \quad c[k-w] := \text{true};$

Ostateczna wersja procedury umożliwiającej znalezienie metodą prób i błędów wszystkich rozwiązań problemu ośmiu hetmanów może mieć postać następującą:

```

procedure Probuje( $k$ : integer);
var
   $w$ : integer;
begin
  for  $w:=1$  to 8 do
    if  $a[w] \text{ and } b[k+w] \text{ and } c[k-w]$  then
      begin
         $x[k] := w;$ 
         $a[w] := \text{false}; \quad b[k+w] := \text{false}; \quad c[k-w] := \text{false};$ 
        if  $k < 8$  then Probuje( $k+1$ )
        else Zanotuj;
         $a[w] := \text{true}; \quad b[k+w] := \text{true}; \quad c[k-w] := \text{true};$ 
      end;
  end;

```

Po znalezieniu kolejnego rozwiązania wywołuje ona procedurę *Zanotuj*, której zadaniem jest zapamiętanie rozwiązania. Implementacja procedury *Zanotuj* należy do użytkownika wykorzystującego algorytm. Na przykład może on wydrukować zawartość tablicy  $x$  lub przedstawić wynik graficznie na ekranie monitora.

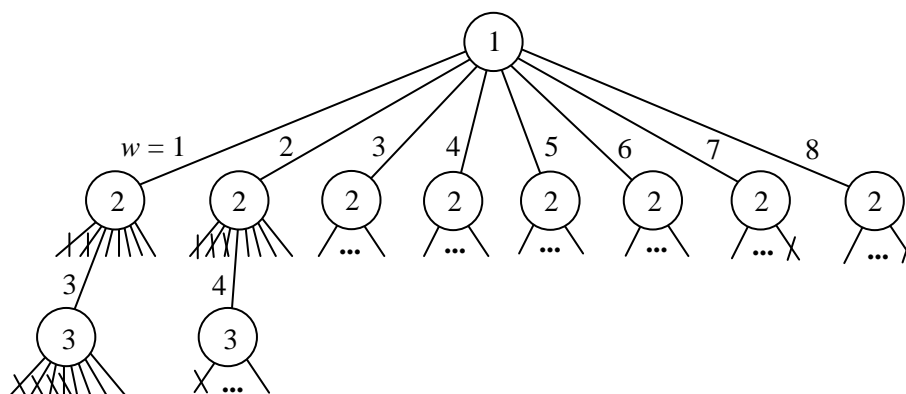
Jest oczywiste, że przed pierwszym wykorzystaniem procedury *Probuje*, dla parametru  $k = 1$ , trzeba zainicjalizować tablice  $a$ ,  $b$  i  $c$ . Ta część programu może wyglądać następująco:

```

for  $i:= 1$  to 8 do  $a[i] := \text{true};$ 
for  $i:= 2$  to 16 do  $b[i] := \text{true};$ 
for  $i:=-7$  to 7 do  $c[i] := \text{true};$ 
Probuje(1);

```

Przyjrzyjmy się drzewu przeszukiwań algorytmu, którego początkowy fragment jest pokazany na rysunku 5.10. Z każdego wierzchołka, którego numer jest wartością parametru  $k < 8$  wywołania procedury *Probuj*, wiedzie osiem krawędzi do wierzchołków o numerach  $k + 1$ . Krawędzie te odpowiadają wartościom zmiennej sterującej  $w = 1, 2, \dots, 8$  pętli **for**. Część z nich jest obcinana, przez co jest eliminowana z dalszych poszukiwań, ponieważ prowadzi do „ślepego zaułka”.



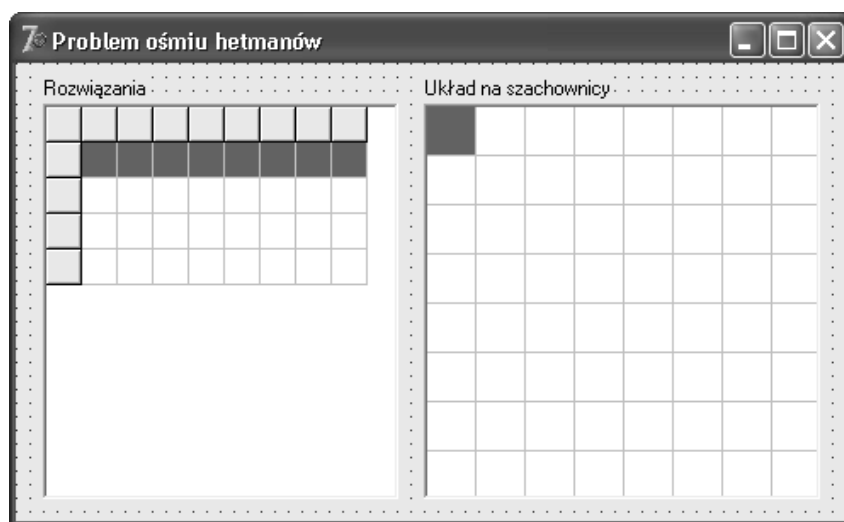
5.10. Początek obciętego drzewa przeszukiwań algorytmu

Pierwszego hetmana możemy ustawić w dowolnym z ośmiu wierszy pierwszej kolumny, dlatego z wierzchołka 1 wychodzi osiem krawędzi do wierzchołków 2. Gdy ustawimy go w pierwszym wierszu, drugiego hetmana możemy ustawić w wierszach od trzeciego do ósmego drugiej kolumny, dlatego dwie krawędzie wychodzące z wierzchołka 2 są obcięte. Z kolei trzeciego hetmana, przy ustawieniu pierwszego w wierszu pierwszym i drugiego w trzecim, możemy ustawić dopiero w piątym wierszu trzeciej kolumny, przez co cztery krawędzie wychodzące z wierzchołka 3 są obcięte. Generalnie obcinane są te krawędzie, a wraz z nimi całe poddrzewa, które opisują drogę nieprowadzącą do rozwiązania, gdyż powodują ustawienie kolejnego hetmana w polu szachowanym.

Zakładamy, że krawędzie wychodzące z wierzchołków o numerze 8 wiodą do wierzchołków (liści drzewa) odpowiadających wywołaniom procedury *Zanotuj*, która zapamiętuje znalezione rozwiązanie. Jest oczywiste, że z wierzchołka o numerze 8 może wychodzić co najwyżej jedna taka krawędź.

Aby przekonać się, jak szybki jest algorytm, zbudujemy aplikację okienkową w Delphi, której zadaniem będzie znalezienie wszystkich rozwiązań problemu ośmiu hetmanów. Na formularzu tej aplikacji umieszczamy dwie siatki, tekstową *StringGrid* i graficzną *DrawGrid*, oraz opisujące je etykiety (por. rys. 5.11). Siatkę tekstową wykorzystamy do wyświetlenia wszystkich znalezionych rozwiązań, po

jednym w wierszu, a graficzną do wizualizacji układu hetmanów na szachownicy dla dowolnie wybranego rozwiązania w siatce tekstowej. Liczbę kolumn pierwszej siatki określamy jako 9, a rozmiar komórek jako  $20 \times 20$ , natomiast liczbę kolumn i wierszy drugiej jako 8, a rozmiar komórek jako  $28 \times 28$ . Ponadto właściwościom *goRowSelect* i *goThumbTracking* siatki tekstowej przypisujemy wartość *true*, co umożliwi wybór jej całych wierszy i przewijanie na bieżąco zawartości za pomocą paska przewijania.



**Rys. 5.11.** Projekt formularza aplikacji rozwiązującej problem ośmiu hetmanów

Aby uatrakcyjnić program, tworzymy dwie mapy bitowe o rozmiarze  $24 \times 24$  pikseli przedstawiające hetmana na białym i szarym polu. W tym celu możemy posłużyć się prostym edytorem rysunków *Image Editor* [5], dostępnym z menu *Tools* Delphi, bądź jakimkolwiek innym narzędziem graficznym do edycji bitmap. Obrazki zapisujemy w plikach *h1.bmp* i *h2.bmp* w katalogu programu.

Rozbudowę modułu formularza rozpoczynamy od zadeklarowania w klasie *TForm1* tablic *x*, *a*, *b* i *c* wykorzystywanych przez algorytm. Deklarujemy również pole *n*, w którym liczona będzie liczba wszystkich rozwiązań problemu ośmiu hetmanów, dwuelementową tablicę *Hetman* reprezentującą obie mapy bitowe:

```
n: integer;
Hetman: array[Boolean] of TBitmap;
```

oraz procedury *Probuj* i *Zanotuj* występujące w algorytmie. Wygenerowany przez Delphi pusty szkielet metody *Probuj* uzupełniamy, przepisując bez żadnych zmian

przytoczony kod procedury o tej samej nazwie. Natomiast w metodzie *Zanotuj* zwiększamy liczbę  $n$  znalezionych rozwiązań o 1 i (w razie potrzeby) liczbę wierszy siatki *StringGrid1* oraz zapisujemy w jej wolnym wierszu pozycje hetmanów przechowywane tymczasowo w tablicy  $x$ :

```
procedure TForm1.Zanotuj;
var
  k: integer;
begin
  Inc(n);
  if n >= StringGrid1.RowCount then
    StringGrid1.RowCount := n+1;
  StringGrid1.Cells[0, n] := IntToStr(n);
  for k:=1 to 8 do
    StringGrid1.Cells[k, n] := IntToStr(x[k]);
end;
```

Następnym krokiem jest zdefiniowanie procedur obsługi zdarzeń *OnCreate* i *OnDestroy* formularza. Pierwsza tworzy dwa obiekty klasy *TBitmap* i ładuje ich zawartości z plików *h1.bmp* i *h2.bmp*, inicjalizuje tytułowy wiersz siatki tekstowej i pola  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $n$  oraz wywołuje metodę *Probuj* dla parametru  $k = 1$ :

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  Hetman[false] := TBitmap.Create;
  Hetman[false].LoadFromFile('h1.bmp');
  Hetman[true] := TBitmap.Create;
  Hetman[true].LoadFromFile('h2.bmp');
  for i:= 1 to 8 do
    begin
      a[i] := true;
      StringGrid1.Cells[i, 0] := IntToStr(i);
    end;
  for i:= 2 to 16 do b[i] := true;
  for i:=-7 to 7 do c[i] := true;
  n := 0;
  Probuj(1);
end;
```

Druga natomiast niszczy oba obiekty *TBitmap*, gdy aplikacja kończy działanie:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Hetman[false].Free;
  Hetman[true].Free;
end;
```

Jak widać, cała praca algorytmu jest wykonywana w momencie, gdy formularz jest tworzony. Aby się przekonać, że algorytm działa szybko, wystarczy uruchomić niedokończoną jeszcze aplikację, a lista wszystkich rozwiązań problemu ośmiu hetmanów natychmiast ukazuje się na ekranie. Jest ich 92, ale z uwagi na symetrię szachownicy istotnie różnych jest znacznie mniej.

Aby zakończyć proces tworzenia aplikacji, precyzujemy operację malowania komórek siatki graficznej i wymuszenia jej przemalowania, gdy w siatce tekstowej wybrane zostanie rozwiązanie. W tym celu definiujemy dwie procedury obsługi zdarzeń – *OnDrawCell* siatki graficznej i *OnClick* tekstowej. W drugiej metodzie wystarczy po prostu wywołać metodę *Invalidate* obiektu *DrawGrid1*. Kod pierwszej jest nieco skomplikowany. Metoda ma kilka parametrów, z których interesują nas *ACol*, *ARow* i *Rect*. Pierwsze dwa określają kolumnę i wiersz komórki, którą malujemy, a trzeci prostokąt, który ona zajmuje. Najpierw dowiadujemy się, jaki kolor ma pole szachownicy reprezentowane przez komórkę. Przyjmujemy, że określa go lewy górny piksel bitmapy *Hetman[false]*, gdy wartość *ACol + ARow* jest parzysta, lub bitmapy *Hetman[true]*, gdy jest nieparzysta. Kolorem tym wypełniamy cały prostokąt komórki. Następnie sprawdzamy, czy wybrane w siatce *StringGrid1* rozwiązanie opisuje układ, w którym na pozycji określonej przez parametry *ACol* i *ARow* jest ustawiony hetman. Jeśli tak, pośrodku komórki malujemy bitmapę, która reprezentuje ustawionego w polu hetmana. Oto kod tej metody:

```
procedure TForm1.DrawGrid1DrawCell(Sender: TObject;
  ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
var
  w, x, y: integer;
  h      : TBitmap;
begin
  h := Hetman[Odd(ACol+ARow)];
  DrawGrid1.Canvas.Brush.Color := h.Canvas.Pixels[0, 0];
  DrawGrid1.Canvas.FillRect(Rect);
  w := StrToInt(StringGrid1.Cells[ACol+1, StringGrid1.Row]);
  if ARow+1 = w then
    begin
      x := (Rect.Right - Rect.Left - h.Width) div 2;
      y := (Rect.Bottom - Rect.Top - h.Height) div 2;
      DrawGrid1.Canvas.Draw(Rect.Left+x, Rect.Top+y, h);
    end;
  end;
```

Możemy wreszcie przedstawić pełny kod modułu formularza aplikacji znajdującej wszystkie rozwiązania problemu ośmiu hetmanów:

```
unit MainUnit;

interface
```



```

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids, ExtCtrls;

type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    DrawGrid1: TDrawGrid;
    Label1: TLabel;
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DrawGrid1DrawCell(Sender: TObject;
      ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
    procedure StringGrid1Click(Sender: TObject);
  private
    x: array[1..8] of integer;
    a: array[1..8] of Boolean;
    b: array[2..16] of Boolean;
    c: array[-7..7] of Boolean;
    n: integer;
    Hetman: array[Boolean] of TBitmap;
    procedure Probujs(k: integer);
    procedure Zanotuj;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  Hetman[false] := TBitmap.Create;
  Hetman[false].LoadFromFile('h1.bmp');
  Hetman[true] := TBitmap.Create;
  Hetman[true].LoadFromFile('h2.bmp');
  for i:= 1 to 8 do
    begin
      a[i] := true;
      StringGrid1.Cells[i, 0] := IntToStr(i);
    end;
  for i:= 2 to 16 do b[i] := true;
  for i:=-7 to 7 do c[i] := true;
  n := 0;
  Probujs(1);
end;

```

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
    Hetman[false].Free;
    Hetman[true].Free;
end;

procedure TForm1.Probuj(k: integer);
var
    w: integer;
begin
    for w:=1 to 8 do
        if a[w] and b[k+w] and c[k-w] then
            begin
                x[k] := w;
                a[w] := false; b[k+w] := false; c[k-w] := false;
                if k<8 then Probuj(k+1)
                else Zanutuj;
                a[w] := true; b[k+w] := true; c[k-w] := true;
            end;
        end;
    end;
end;

procedure TForm1.Zanutuj;
var
    k: integer;
begin
    Inc(n);
    if n >= StringGrid1.RowCount then
        StringGrid1.RowCount := n+1;
    StringGrid1.Cells[0, n] := IntToStr(n);
    for k:=1 to 8 do
        StringGrid1.Cells[k, n] := IntToStr(x[k]);
    end;
end;

procedure TForm1.DrawGrid1DrawCell(Sender: TObject;
    ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);
var
    w, x, y: integer;
    h      : TBitmap;
begin
    h := Hetman[Odd(ACol+ARow)];
    DrawGrid1.Canvas.Brush.Color := h.Canvas.Pixels[0, 0];
    DrawGrid1.Canvas.FillRect(Rect);
    w := StrToInt(StringGrid1.Cells[ACol+1, StringGrid1.Row]);
    if ARow+1 = w then
        begin
            x := (Rect.Right - Rect.Left - h.Width) div 2;
            y := (Rect.Bottom - Rect.Top - h.Height) div 2;
            DrawGrid1.Canvas.Draw(Rect.Left+x, Rect.Top+y, h);
        end;
    end;
end;
```

```

procedure TForm1.StringGrid1Click(Sender: TObject);
begin
    DrawGrid1.Invalidate;
end;

end.

```

Efekt końcowy wykonania aplikacji można zobaczyć na rysunku 5.12, który pokazuje początek listy rozwiązań wygenerowanych przez algorytm i pierwsze znalezione rozwiązanie graficznie.



Rys. 5.12. Początek listy rozwiązań problemu ośmiu hetmanów

## 5.5. Wypłacalność kwoty

W ostatnim przykładzie algorytmu z powrotami liczba prób podejmowanych w kolejnych krokach nie będzie stała. Przykład dotyczy często spotykanego problemu polegającego na określeniu, czy podana kwota może być dokładnie wypłacona z portmonetki, a jeśli tak, to z wykorzystaniem jakich nominałów i jakiej ich liczby. Przyjmujemy, że zawartość portmonetki jest podana w postaci listy nominałów polskich banknotów i monet wraz z liczbami ich wystąpień.

Wszystkie obecnie używane nominały możemy zapisać w postaci tablicy liczb całkowitych wyrażających ich wartości w przeliczeniu na grosze:

```

const Nom: array[1..14] of integer = (
    20000,10000,5000,2000,1000,500,200,100,50,20,10,5,2,1);

```

Do reprezentowania danych w programie, który tym razem będzie aplikacją konsolową w Delphi, użyjemy zmiennych globalnych:

```
var Por, Ile: array[1..14] of integer;
    kgr: integer;
    Stop: Boolean;
```

Ich znaczenie jest następujące:

- Por* – wczytane z pliku liczby wystąpień poszczególnych banknotów i monet, określające zawartość portmonetki,
- Ile* – liczby banknotów i monet dobierane metodą prób i błędów przez algorytm w celu złożenia żądanej kwoty,
- kgr* – wyrażona w groszach kwota, która ma być wypłacona z portmonetki,
- Stop* – wartość logiczna służąca do zanotowania faktu, że kwota została złożona i należy zakończyć dalsze poszukiwania.

Algorytm polega na podejmowaniu kolejnych prób dobierania banknotów i monet z portmonetki aż do zebrania żądanej kwoty lub wyczerpania wszystkich możliwych kombinacji. Aby kwota została wyrażona za pomocą jak najmniejszej liczby banknotów i monet, należy kierować się zachłannością. Po pierwsze, próby dobierania nominałów powinny być podejmowane w kolejności od najwyższego do najniższego. Po drugie, poszukiwanie liczby wystąpień nominału powinno rozpoczynać się od liczby możliwie jak największej. Rozumowanie to prowadzi do wstępnego sformułowania algorytmu:

```
procedure Próbuje(k, kwota)
  Ile[k] := maksymalna możliwa liczba wystąpień
           nominału Nom[k] w kwocie
  if istnieje następny nominał then
    repeat
      Próbuje(k+1, kwota - Ile[k]*Nom[k])
      if not Stop then zmniejsz Ile[k]
    until Stop or (Ile[k] < 0)
  else
    if kwota = Ile[k]*Nom[k] then Stop := true
```

Parametry procedury *Próbuje* oznaczają, że po próbach dobierania nominałów wyższych, wyszczególnionych w tablicy *Nom* na pozycjach od 1 do  $k-1$ , pozostała jeszcze do wypłacenia część kwoty, którą należy złożyć z nominałów niższych. Dlatego po dobraniu jak największej liczby niewykorzystanego nominału *Nom[k]*, procedura jest wywoływana dla następnych nominałów i kwoty pomniejszonej o wartość dobranej liczby nominału *Nom[k]*. Jeżeli postępowanie to nie doprowadzi do złożenia kwoty, jest kontynuowane dla mniejszych liczb wystąpień nomina-

tu  $Nom[k]$  i niższych nominałów, aż w końcu cała kwota zostanie zebrana bądź wszystkie próby zostaną podjęte. Jest oczywiste, że na początku zmiennej *Stop* należy nadać wartość *false*, a procedurę *Próbuj* wywołać z parametrem  $k = 1$ .

Poniżej przedstawiono program rozwiązujący problem wypłacalności kwoty, w którym zawarta jest pełna wersja tej procedury. Program czyta plik tekstowy opisujący zawartość portmonetki. Każdy wiersz pliku zawiera liczbę rzeczywistą i całkowitą. Pierwsza określa wysokość nominału, a druga podaje, ile banknotów lub monet o tym nominale znajduje się w portmonetce. Procedura *Czytaj* wywołuje dla każdej pary pobranych z pliku liczb pomocniczą procedurę *Wstaw*, która przeszukuje tablicę nominałów *Nom* w celu znalezienia miejsca, w którym występuje wczytana wartość nominalna przeliczona na grosze, a następnie wstawia liczbę wystąpień tego nominału do znalezionego miejsca tablicy *Por*. Kwota, którą należy złożyć, jest liczbą rzeczywistą wczytywaną z klawiatury.

```
program Kwota;  
  
{ $APPTYPE CONSOLE }  
  
uses  
    SysUtils, Math;  
  
const  
    Nom: array[1..14] of integer =  
        (20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1);  
var  
    Por, Ile: array[1..14] of integer;  
    kgr: integer;  
    Stop: Boolean;  
  
procedure Wstaw(x: real; k: integer);  
var  
    v, i: integer;  
begin  
    v := Round(100*x);  
    i := 1;  
    while (i <= 14) and (Nom[i] <> v) do Inc(i);  
    if i <= 14 then Por[i] := k;  
end;  
  
procedure Czytaj;  
var  
    Plik: TextFile;  
    k: integer;  
    x: real;  
    Nazwa: string;  
begin  
    for k:=1 to 14 do Por[k] := 0;  
    Write('Nazwa pliku: ');
```

```

    ReadLn(Nazwa);
    AssignFile(Plik, Nazwa);
    Reset(Plik);
    while not Eof(Plik) do
    begin
        ReadLn(Plik, x, k);
        Wstaw(x, k);
    end;
    Close(Plik);
    Write('Kwota: ');
    ReadLn(x);
    kgr := Round(100*x);
end;

procedure Drukuj(var a: array of integer; s: string);
var
    i: integer;
begin
    WriteLn(s);
    WriteLn('-----');
    for i:=0 to High(a) do
        if a[i]>0 then
            WriteLn(Nom[i+1]/100:6:2, ' ', a[i]);
    end;

procedure ProbuJ(k, kwota: integer);
begin
    Ile[k] := Min(kwota div Nom[k], Por[k]);
    if k<14 then
        repeat
            ProbuJ(k+1, kwota - Ile[k]*Nom[k]);
            if not Stop then Dec(Ile[k]);
        until Stop or (Ile[k] < 0)
    else
        if kwota = Ile[k]*Nom[k] then Stop := true;
    end;

begin
    Czytaj;
    Drukuj(Por, 'Zawartość portmonetki:');
    Stop := false;
    ProbuJ(1, kgr);
    if Stop
    then Drukuj(Ile, 'Wypłacone nominały:')
    else WriteLn('Kwota nie do wypłacenia');
    ReadLn;
end.

```

Program wypisuje za pomocą procedury *Drukuj* zawartość portmonetki, a także znalezione rozwiązanie, gdy istnieje, lub komunikat informujący, że podanej kwoty nie daje się złożyć z zawartości portmonetki. Przykładowo, dla pliku postaci:

200	1
100	3
50	2
10	2
5	3
2	4
0.50	1
0.20	2
0.05	1
0.02	3

i kwoty 156,44 zł program wypisuje rozwiązanie:

100.00	1
50.00	1
2.00	3
0.20	2
0.02	2

Natomiast dla tego samego pliku i kwoty 128,73 zł informuje, że nie daje się jej złożyć z zawartości portmonetki.

## 5.6. Interpretator wyrażeń arytmetycznych

Na koniec niniejszego rozdziału zajmiemy się pokrótce interesującym zagadnieniem tworzenia programów analizujących struktury językowe – **translatorów**, w którym szczególnie uwidacznia się moc rozwiązań rekurencyjnych. Zazwyczaj translatory dzielimy na **kompilatory** i **interpretatory**, chociaż w praktyce buduje się również translatory wykazujące jednocześnie cechy obu kategorii. Czytelnik zapewne wie, że w przypadku kompilatora cały kod źródłowy programu zostaje przełożony na postać „zrozumiałą dla komputera”, która następnie może zostać wykonana, zaś w przypadku interpretatora przekłady poszczególnych jednostek składniowych, np. instrukcje, mogą być wykonywane bez czekania na zakończenie translacji całego programu. Wybór jednego z tych dwu rozwiązań zależy od wielu czynników, którymi tu nie będziemy się zajmować.

Tak czy owak, budowa translatora nie należy do zadań łatwych. Podstawowym celem każdego translatora jest **analiza składni** (syntaktyki) języka, którą stanowi zbiór reguł pozwalający decydować, czy określone ciągi symboli są rozpoznawalne jako poprawne w tym języku, czy jako niepoprawne. Jeżeli w trakcie analizy translator produkuje kod wynikowy dla docelowego komputera (procesora), jest kompilatorem, a jeżeli wykonuje (interpretuje) instrukcje odpowiadające poprawnym jednostkom składniowym kodu źródłowego, jest interpretatorem. Program, który dokonuje jedynie samej analizy składni kodu źródłowego w danym języku, jest tylko **analizatorem składniowym**.

Postaramy się teraz opracować w Delphi interpretator bardzo prostego języka programowania. Instrukcjami w tym języku będą tylko instrukcje przypisania, a prezentowany program będzie interpretatorem, gdyż oprócz sprawdzania, czy instrukcje przypisania są poprawne, będzie je wykonywał. Składnia języka, sformułowana za pomocą **diagramów syntaktycznych** pokazanych na rysunku 5.13, umożliwia łatwe sprawdzenie, czy dany ciąg symboli jest poprawny. Linie oznaczone strzałkami wskazują ścieżki, po których można się poruszać. Wychodząc od diagramu oznaczonego napisem *program* i idąc wybraną ścieżką, generujemy możliwe ciągi symboli. Jeżeli napotkany element graficzny ma odrębny diagram, przechodzimy do niego, a jeżeli nie ma, dopisujemy znajdujący się w nim symbol do generowanego tekstu. Na przykład z diagramu oznaczonego napisem *stała* wynika, że stała jest niepustym ciągiem cyfr lub dwoma takimi ciągami rozdzielonymi przecinkiem<sup>8</sup>, a z diagramu oznaczonego napisem *zmienna*, że zmienną (właściwie nazwą) jest dowolny ciąg liter i cyfr rozpoczynający się od litery.

Zauważmy, że w naszym języku instrukcje przypisania są pisane „w duchu języka C”, gdyż są konstrukcjami złożonymi ze zmiennej, operatora przypisania =, wyrażenia i średnika. Wyrażenia składają się z argumentów i operatorów arytmetycznych. Argumentami mogą być stałe, zmienne i inne wyrażenia ujęte w okrągłe nawiasy. Dla uproszczenia programu zakładamy, że wartościami argumentów są liczby rzeczywiste. Podobnie jak w większości języków programowania, operatory mogą być jednoargumentowe (+, -) oraz dwuargumentowe multiplikatywne (\*, /) i addytywne (+, -). Kolejność wykonywania operacji, określona przez diagramy oznaczone napisami *czynnik*, *składnik* i *wyrażenie*, jest zgodna z przyjętą zwyczajowo notacją matematyczną. Jak widać na rysunku 5.13, diagramy te opisują rekurencję pośrednią (wyrażenie może być składnikiem, składnik czynnikiem, zaś czynnik zamkniętym w nawiasach wyrażeniem).

Interpretator ma sekwencyjnie sprawdzać poprawność instrukcji przypisania, wykonywać je i pokazywać wartości przypisane zmiennym. Na przykład wynikiem jego działania w przypadku programu

```
a1 = 3,6;  
a2 = 5,4;  
H = 2*a1*a2/(a1 + a2);
```

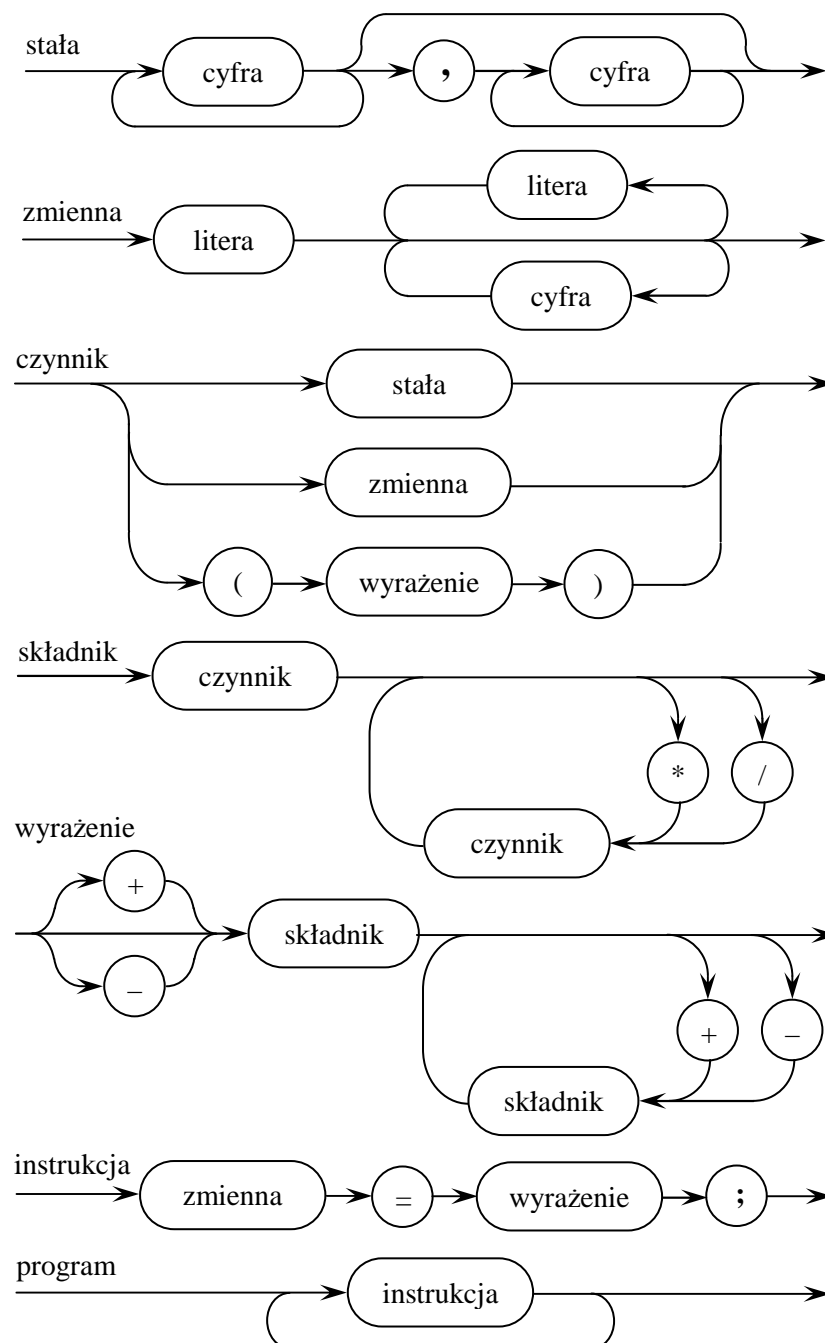
powinno być

```
a1 = 3,6  
a2 = 5,4  
H = 4,32
```

---

<sup>8</sup> Do oddzielenia części całkowitej od ułamkowej liczby będziemy zamiast kropki używać przecinka (takie jest domyślne ustawienie w polskiej wersji Windows).





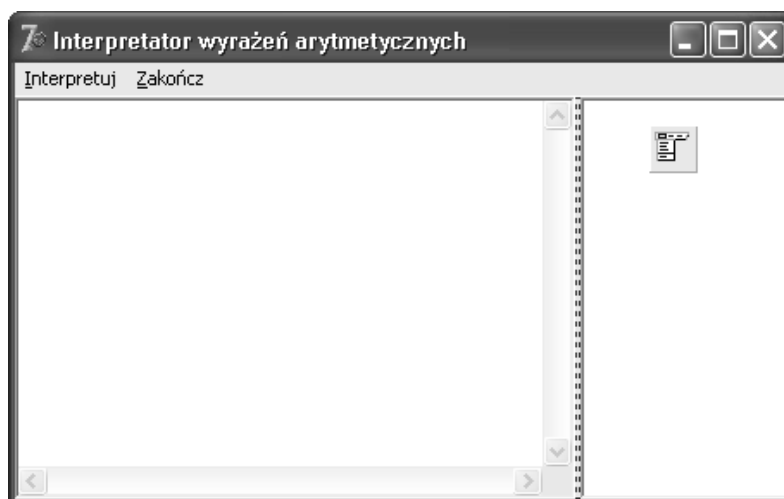
Rys. 5.13. Diagramy syntaktyczne prostego języka programowania

Interpretator powinien sensownie reagować na błędy formalne. Żaden tekst źródłowy nie powinien doprowadzić do przerwania jego działania, a każda napotkana nieprawidłowa konstrukcja językowa powinna być przez niego właściwie zdiagnozowana. Ponadto interpreter powinien wykrywać błędy wykonania, które mogą się przydarzyć podczas obliczania wartości wyrażeń. Pospolitymi błędami tego rodzaju są użycie zmiennej o nieokreślonej wartości i dzielenie przez zero.

Przejdźmy wreszcie do zaimplementowania interpretatora wyrażeń w Delphi. Praktycznie wszystkie funkcje umożliwiające edycję tekstu udostępnia komponent *Memo*, dlatego budowę aplikacji rozpoczynamy od umieszczenia go na formularzu oraz ustawienia jego właściwości *Align* na *alClient* i *ScrollBars* na *ssBoth*. Drugim komponentem, który wstawiamy do formularza, jest *ListBox*. Jego właściwość *Align* ustawiamy na *alRight*, przez co będzie on pełnił rolę panelu wyświetlającego wyniki obliczeń po prawej stronie okna. Komponent *ListBox* ma właściwość *Items*, która obsługuje łańcuchy postaci:

*nazwa=wartość*

Dzięki temu nie musimy korzystać z dodatkowej tablicy do przechowywania wartości zmiennych. Aby zakończyć projektowanie okna interpretatora, definiujemy jeszcze proste menu z dwoma elementami o etykietach *Interpretuj* i *Zakończ* oraz wstawiamy komponent *Splitter* (pasek podziału), którego właściwość *Align* ustawiamy na *alRight*. Pasek dzieli obszar roboczy okna na dwie części (por. rys. 5.14), których szerokości można zmieniać za pomocą myszki.



Rys. 5.14. Projekt okna interpretatora wyrażeń arytmetycznych

Interpretator ma pobierać do analizy kolejne znaki tekstu źródłowego stanowiącego wartość właściwości *Text* komponentu *Memo1*. Zakładamy, że pobierany znak i jego pozycja w tekście będą przechowywane w polach klasy *TForm1*:

```
Znak: char;  
Poz: integer;
```

Przejdźcie do następnego znaku wyrazimy za pomocą dwóch metod tej klasy:

```
procedure PobierzZnak;  
procedure PobierzSymbol;
```

Pierwsza ma pobrać kolejny znak tekstu do pola *Znak* i zapamiętać jego pozycję w polu *Poz*. Druga ma dodatkowo pomijać „białe” znaki, tj. spacje i symbole końca wierszy (znaki CR i LF o kodach 13 i 10). Metody te definiujemy następująco:

```
procedure TForm1.PobierzZnak;  
begin  
  if Poz < Length(Memo1.Text) then  
  begin  
    Inc(Poz);  
    Znak := Memo1.Text[Poz];  
  end else  
    Znak := #0;  
  end;  
  
procedure TForm1.PobierzSymbol;  
begin  
  repeat  
    PobierzZnak;  
  until not (Znak in [#13, #10, ' ']);  
end;
```

Przyjęliśmy umownie, że znak o kodzie 0 jest symbolem końca tekstu. Znak ten nie występuje w tekście źródłowym, jest generowany przez metodę *PobierzZnak*, gdy cały tekst zostanie zinterpretowany.

Dalsza rozbudowa interpretatora wymaga przejścia od przedstawionych na rysunku 5.13 diagramów składni do kodu programu. Podstawową regułą, którą należy się kierować, jest zastąpienie każdego diagramu odpowiednim podprogramem [8, 28], co w naszym przypadku prowadzi do nowych metod klasy *TForm1*:

```
function Stala: real;  
function Zmienna: string;  
function Czynn timer;  
function Skladnik: real;  
function Wyrazenie: real;  
procedure Instrukcja;
```

Uważny Czytelnik spostrzeże, że powyższa lista podprogramów nie obejmuje diagramu oznaczonego napisem *program*. Odpowiednikiem tego diagramu jest pętla **repeat ... until** w poniższej procedurze obsługi zdarzenia *OnClick* polecenia *Interpretuj*, której zadaniem jest analiza całego tekstu źródłowego.

```
procedure TForm1.Interpretuj1Click(Sender: TObject);
begin
    ListBox1.Clear;
    Poz := 0;
    PobierzSymbol;
    repeat
        Instrukcja;
    until Znak = #0;
end;
```

Zauważmy, że zanim metoda *Instrukcja* zostanie po raz pierwszy wywołana, w polu *Znak* „czeka” pierwszy symbol (znak różny od spacji, CR i LF). Wypada podkreślić, że nie jest to przypadkowe. Analiza składni odbywa się przy „czytaniu tekstu z wyprzedzeniem” o jeden symbol [28]. Każdy podprogram odpowiadający diagramowi składni zastaje w polu *Znak* symbol, który stara się rozpoznać i zinterpretować, a po zinterpretowaniu go i symboli następnych, pobieranych do pola *Znak* za pomocą procedury *PobierzZnak* bezpośrednio lub pośrednio, pozostawia w tym polu kolejny symbol do dalszej analizy. Jeżeli zastany w polu *Znak* symbol nie jest rozpoznawalny, traktowany jest jako nieprawidłowy.

Przykładowo, funkcja *Zmienna* oczekuje w polu *Znak* litery. Jeśli jest to litera, pobiera kolejne znaki i zlicza je, aż do napotkania znaku innego niż litera i cyfra. Na koniec pomija ewentualne „białe” znaki, a z uprzednio rozpoznanych liter i cyfr tworzy nazwę zmiennej. Oto pełny kod tej metody:

```
function TForm1.Zmienna: string;
var
    p, k: integer;
begin
    if Znak in ['A'..'Z', 'a'..'z'] then
        begin
            p := Poz;
            k := 0;
            repeat
                Inc(k);
                PobierzZnak;
            until not (Znak in ['A'..'Z', 'a'..'z', '0'..'9']);
            if Znak in [#13, #10, ' '] then PobierzSymbol;
            Result := Copy(Mem1.Text, p, k);
        end else
            raise Exception.Create('Oczekiwana litera');
    end;
```

Wystąpienie zmiennej w wyrażeniu (diagram *czynnik*) powoduje przejście listy łańcuchów postaci *nazwa=wartość* przechowywanych we właściwości *Items* klasy *TStrings* komponentu *ListBox1*. Celem tej operacji jest pobranie wartości zmiennej. Brak odpowiedniego łańcucha należy traktować jako błąd, ponieważ oznacza użycie zmiennej o nieokreślonej wartości. Indeks elementu listy o podanej nazwie udostępnia metoda *IndexOfName*, a wartość znalezionej metody *ValueFromIndex*. Rozważania te prowadzą do następującej definicji zapowiedzianej metody *Czynnik*:

```
function TForm1.Czynnik: real;
var
  k: integer;
begin
  if Znak in ['0'..'9'] then Result := Stala else
  if Znak in ['A'..'Z', 'a'..'z'] then
  begin
    k := ListBox1.Items.IndexOfName(Zmienna);
    if k >= 0 then
      Result := StrToFloat(ListBox1.Items.ValueFromIndex[k])
    else
      raise Exception.Create('Nieokreślona wartość zmiennej');
  end else if Znak = '(' then
  begin
    PobierzSymbol;
    Result := Wyrażenie;
    if Znak = ')' then PobierzSymbol else
      raise Exception.Create('Oczekiwany nawias ')''');
  end else
    raise Exception.Create('Oczekiwana cyfra, litera lub "("');
end;
```

Zmienna występuje również po lewej stronie operatora przypisania = (diagram *instrukcja*), lecz tym razem brak odpowiedniego łańcucha *nazwa=wartość* na liście *Items* nie jest błędem, ponieważ łańcuch taki należy utworzyć, uwzględniając przypisywaną zmiennej wartość. Nowa pozycja wymaga usunięcia poprzedniej o tej samej nazwie. Definicja metody *Instrukcja* może wyglądać następująco:

```
procedure TForm1.Instrukcja;
var
  s: string;
  v: real;
  k: integer;
begin
  s := Zmienna;
  if Znak = '=' then
  begin
    PobierzSymbol;
    v := Wyrażenie;
```

```

    if Znak = ';' then PobierzSymbol else
        raise Exception.Create('Oczekiwany znak ";"');
    k := ListBox1.Items.IndexOfName(s);
    if k >= 0 then ListBox1.Items.Delete(k);
    ListBox1.Items.Add(s + '=' + FloatToStr(v));
end else
    raise Exception.Create('Oczekiwany znak "="');
end;

```

Postępując podobnie, definiujemy pozostałe trzy zapowiedziane metody: *Stala*, *Skladnik* i *Wyrazenie*. Czytelnik może je łatwo odnaleźć w przedstawionym niżej kodzie modułu formularza. Metoda *InterpretujClick* została nieco rozbudowana, by w przypadku błędu wskazywała jego lokalizację.

```

unit MainUnit;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls, ExtCtrls, ComCtrls, Menus;

type
    TForm1 = class(TForm)
        Memo1: TMemo;
        ListBox1: TListBox;
        MainMenu1: TMainMenu;
        Interpretuj1: TMenuItem;
        Zakocz1: TMenuItem;
        Splitter1: TSplitter;
        procedure InterpretujClick(Sender: TObject);
        procedure Zakocz1Click(Sender: TObject);
    private
        Znak: char; // Ostatni pobrany znak
        Poz: integer; // Pozycja pobranego znaku
    public
        procedure PobierzZnak; // Pobiera kolejny znak
        procedure PobierzSymbol; // jw., pomija białe znaki
        function Stala: real; // Wartość stałej
        function Zmienna: string; // Nazwa zmiennej
        function Czynn timer: real; // Wartość czynn timer
        function Skladnik: real; // Wartość składnika
        function Wyrazenie: real; // Wartość wyrażenia
        procedure Instrukcja; // Realizacja instrukcji
    end;

var
    Form1: TForm1;

implementation

```

```
{ $R *.dfm }

procedure TForm1.PobierzZnak;
begin
    if Poz < Length(Mem1.Text) then
        begin
            Inc(Poz);
            Znak := Mem1.Text[Poz];
        end else
            Znak := #0;
    end;

procedure TForm1.PobierzSymbol;
begin
    repeat
        PobierzZnak;
    until not (Znak in [#13, #10, ' ']);
end;

procedure TForm1.Interpretuj1Click(Sender: TObject);
begin
    ListBox1.Clear;
    Poz := 0;
    PobierzSymbol;
    try
        repeat
            Instrukcja;
        until Znak = #0;
    except
        Mem1.SelStart := Poz;
        raise;
    end;
end;

function TForm1.Zmienna: string;
var
    p, k: integer;
begin
    if Znak in ['A'..'Z', 'a'..'z'] then
        begin
            p := Poz;
            k := 0;
            repeat
                Inc(k);
                PobierzZnak;
            until not (Znak in ['A'..'Z', 'a'..'z'], '0'..'9');
            if Znak in [#13, #10, ' '] then PobierzSymbol;
            Result := Copy(Mem1.Text, p, k);
        end else
            raise Exception.Create('Oczekiwana litera');
    end;
```

```

function TForm1.Stala: real;

  var
    p, k: integer;

  procedure CiagCyfr;
  begin
    if Znak in ['0'..'9'] then
      repeat
        Inc(k);
        PobierzZnak;
      until not (Znak in ['0'..'9'])
    else
      raise Exception.Create('Oczekiwana cyfra');
    end;

  begin { Stala }
    p := Poz;
    k := 0;
    CiagCyfr;
    if Znak = ',' then
      begin
        Inc(k);
        PobierzZnak;
        CiagCyfr;
      end;
    if Znak in [#13, #10, ' '] then PobierzSymbol;
    Result := StrToFloat(Copy(Mem1.Text, p, k));
  end;

function TForm1.Czynnik: real;
var
  k: integer;
begin
  if Znak in ['0'..'9'] then Result := Stala else
  if Znak in ['A'..'Z', 'a'..'z'] then
    begin
      k := ListBox1.Items.IndexOfName(Zmienna);
      if k >= 0 then
        Result := StrToFloat(ListBox1.Items.ValueFromIndex[k])
      else
        raise Exception.Create('Nieokreślona wartość zmiennej');
    end else if Znak = '(' then
      begin
        PobierzSymbol;
        Result := Wyrazenie;
        if Znak = ')' then PobierzSymbol else
          raise Exception.Create('Oczekiwany nawias ')''');
        end else
          raise Exception.Create('Oczekiwana cyfra, litera lub "("');
    end;

```



```
function TForm1.Skladnik: real;
begin
    Result := Czynn timer;
    while Znak in ['*', '/'] do
        if Znak = '*' then
            begin
                PobierzSymbol;
                Result := Result * Czynn timer;
            end else
            begin
                PobierzSymbol;
                Result := Result / Czynn timer;
            end;
    end;
end;

function TForm1.Wyrazenie: real;
begin
    if Znak = '-' then
        begin
            PobierzSymbol;
            Result := -Skladnik;
        end else
        begin
            if Znak = '+' then PobierzSymbol;
            Result := Skladnik;
        end;
    while Znak in ['+', '-'] do
        if Znak = '+' then
            begin
                PobierzSymbol;
                Result := Result + Skladnik;
            end else
            begin
                PobierzSymbol;
                Result := Result - Skladnik;
            end;
    end;
end;

procedure TForm1.Instrukcja;
var
    s: string;
    v: real;
    k: integer;
begin
    s := Zmienna;
    if Znak = '=' then
        begin
            PobierzSymbol;
            v := Wyrazenie;
            if Znak = ';' then PobierzSymbol else
                raise Exception.Create('Oczekiwany znak ";");
        end;
```

```

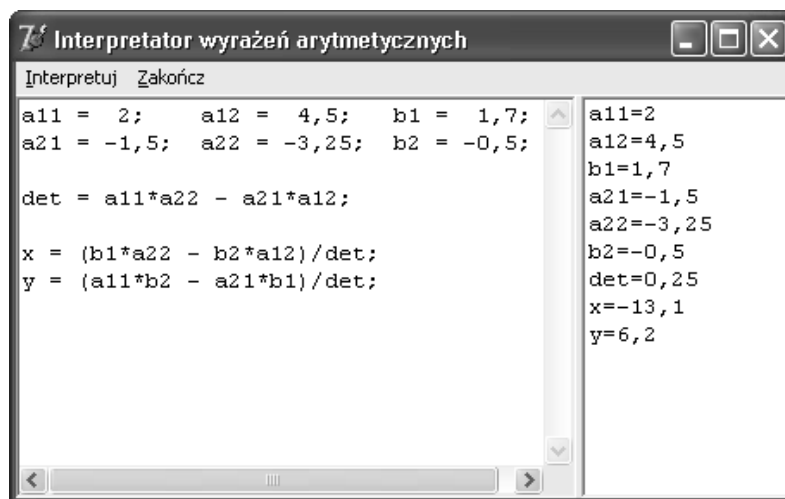
k := ListBox1.Items.IndexOfName(s);
if k>=0 then ListBox1.Items.Delete(k);
ListBox1.Items.Add(s + '=' + FloatToStr(v));
end else
raise Exception.Create('Oczekiwany znak "="');
end;

procedure TForm1.Zakocz1Click(Sender: TObject);
begin
Close;
end;

end.

```

Rysunek 5.15 pokazuje przykład użycia interpretera do rozwiązania układu dwóch równań liniowych o dwóch niewiadomych.



Rys. 5.15. Interpreter wyrażań arytmetycznych w działaniu

## Ćwiczenia

**5.1.** Napisz aplikację konsolową w Delphi, która wypisuje reprezentację dwójkową lub szesnastkową nieujemnej liczby całkowitej bez użycia tablicy, wykorzystując rekurencję.

**5.2.** Funkcja Ackermanna jest zdefiniowana dla dowolnych nieujemnych liczb całkowitych  $m$  i  $n$  następująco [28]:

$$\begin{aligned}
 A(0, n) &= n + 1 & (n \geq 0) \\
 A(m, 0) &= A(m - 1, 1) & (m > 0) \\
 A(m, n) &= A(m - 1, A(m, n - 1)) & (m > 0, n > 0)
 \end{aligned}$$

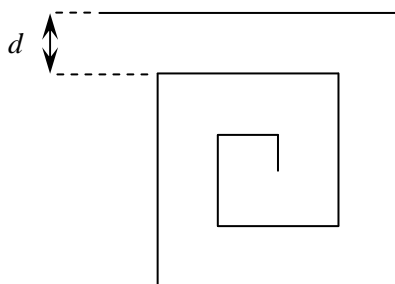
Opracuj dwa algorytmy obliczania wartości tej funkcji: rekurencyjny i iteracyjny. Napisz ich implementacje komputerowe i sprawdź ich zachowanie w praktyce.

**5.3.** Niech  $a_0, a_1, \dots, a_n$  będzie ciągiem  $n$  liczb rzeczywistych. Podaj zależności rekurencyjne określające sposób obliczania wartości poniższych funkcji:

$$P_n(x) = a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n$$

$$F_n(x) = a_n + \frac{x}{a_{n-1} + \frac{x}{a_{n-2} + \frac{x}{a_{n-3} + \dots \frac{x}{a_0}}}}$$

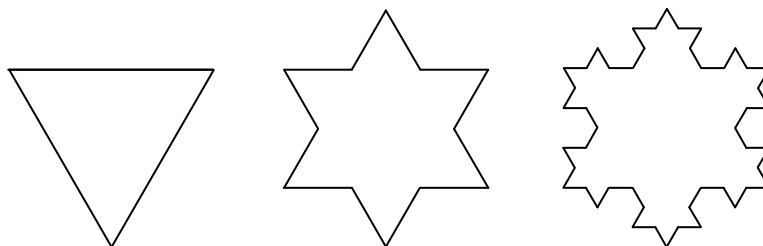
**5.4.** Zbuduj aplikację w Delphi rysującą rekurencyjnie spiralę złożoną z odcinków prostych, w której odstęp pomiędzy sąsiednimi równoległymi odcinkami jest stały (rys. 5.16).



**Rys. 5.16.** Spirala złożona z odcinków prostych

**5.5.** Krzywa Kocha rzędu 0 ma kształt trójkąta równobocznego, krzywa rzędu 1 powstaje z krzywej rzędu 0 w wyniku podziału każdego boku na trzy równe odcinki i wstawienia w miejscu środkowego odcinka dwóch takich odcinków, połączonych i obróconych o kąty  $60^\circ$  i  $-60^\circ$ . Kontynuując to postępowanie, można otrzy-

mywać krzywe wyższych rzędów (rys. 5.17). Znajdź schemat rekurencji dla krzywych Kocha i napisz aplikację okienkową w Delphi, która je rysuje<sup>9</sup>.



**Rys. 5.17.** Krzywe Kocha rzędu 0, 1 i 2

**5.6.** Rysunek 5.18 przedstawia krzywe Hilberta rzędu 1, 2 i 3. Schemat rekurencyjny ich kreślenia opiera się na podziale krzywej na cztery części połączone trzema odcinkami o jednostkowej długości:

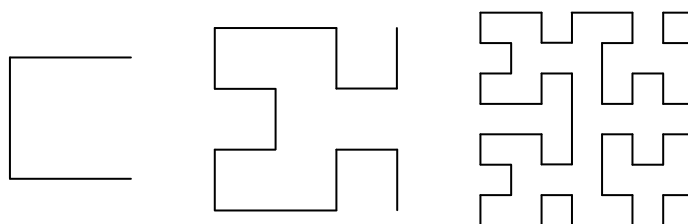
$$A_i: D_{i-1} \leftarrow A_{i-1} \downarrow A_{i-1} \rightarrow B_{i-1}$$

$$B_i: C_{i-1} \uparrow B_{i-1} \rightarrow B_{i-1} \downarrow A_{i-1}$$

$$C_i: B_{i-1} \rightarrow C_{i-1} \uparrow C_{i-1} \leftarrow D_{i-1}$$

$$D_i: A_{i-1} \downarrow D_{i-1} \leftarrow D_{i-1} \uparrow C_{i-1}$$

Znajdź na rysunku te części i łącząc je odcinki. Jak wygląda krzywa Hilberta rzędu 0? Zbuduj aplikację w Delphi, która na podstawie powyższego schematu rysuje krzywe Hilberta<sup>10</sup>.



**Rys. 5.18.** Krzywe Hilberta rzędu 1, 2 i 3

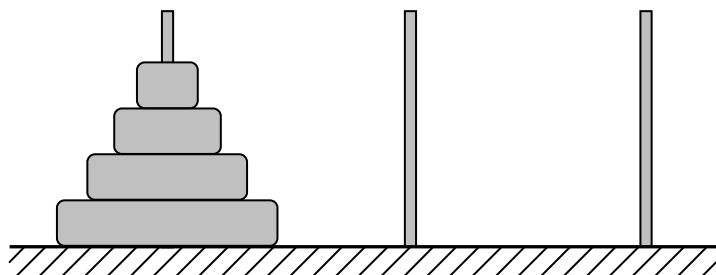
<sup>9</sup> Pełny program w języku Borland C++ w wersji dla DOS można znaleźć w książce [8].

<sup>10</sup> Program w języku Pascal można znaleźć w książce [28].

**5.7.** W zadaniu o wieżach Hanoi mamy trzy pręty i  $n$  krążków o różnych średnicach. Na początku wszystkie krążki są nałożone na pierwszy pręt w kolejności malejących średnic (por. rys. 5.19). Opracuj algorytm przeniesienia wszystkich krążków z pierwszego pręta na trzeci, zgodnie z następującymi zasadami:

- ♦ w każdym kroku przenosi się dokładnie jeden krążek z jednego pręta na inny,
- ♦ krążek nie może być nigdy nałożony na krążek o mniejszej średnicy,
- ♦ drugi pręt może służyć jako magazyn pomocniczy.

**Wskazówka.** Zauważ, że wieżę złożoną z  $n$  krążków można potraktować jako mniejszą, złożoną z  $n - 1$  krążków, nałożoną na krążek  $n^{11}$ .



Rys. 5.19. Wieże Hanoi

**5.8.** Ciąg wszystkich permutacji zbioru  $\{1, 2, 3\}$  w porządku antyleksykograficznym [18] ma postać:

```
1  2  3
2  1  3
1  3  2
3  1  2
2  3  1
3  2  1
```

Przykład ten sugeruje, że ciąg wszystkich  $n!$  permutacji zbioru  $\{1, 2, \dots, n\}$  w tym porządku ma następujące własności:

- 1) ostatnia permutacja jest odwróceniem pierwszej,
- 2) ciąg można podzielić na  $n$  podciągów o długości  $(n - 1)!$  odpowiadających malejącym wartościom elementu na ostatniej pozycji,

<sup>11</sup> Historię zadania i oszacowanie złożoności czasowej jego rozwiązania można znaleźć w książkach [10, 23], a pełne programy w Turbo Pascalu i Borland C++ w wersji dla DOS w książce [11].

- 3) w podciągu zawierającym element  $m$  na ostatniej pozycji początkowe  $n - 1$  elementów każdej permutacji określa ciąg permutacji zbioru  $\{1, 2, \dots, n\} - \{m\}$  w porządku antyleksykograficznym.

Spostrzeżenia te prowadzą do sformułowania rekurencyjnego algorytmu generowania wszystkich  $n!$  permutacji elementów  $a_1, a_2, \dots, a_n$  *in situ*, tj. bez użycia tablicy dodatkowej. Mianowicie, zadanie znalezienia wszystkich permutacji elementów  $a_1, a_2, \dots, a_m$  można potraktować jako  $m$  podzadań znajdujących wszystkie permutacje początkowych  $m - 1$  elementów z ciągu  $a_1, a_2, \dots, a_m$ , przy czym na koniec  $k$ -tego podzadania należy przestawić elementy  $a_k$  i  $a_m$  oraz odwrócić kolejność elementów ciągu  $a_1, a_2, \dots, a_{m-1}$ <sup>12</sup>. Opracuj implementację komputerową tego algorytmu. Jaka jest jego złożoność obliczeniowa?

**5.9.** Załóżmy, że Alibaba może zabierać z Sezamu nie tylko całe przedmioty, ale również ich dowolne ułamkowe części. Możemy sobie np. wyobrazić przedmioty złote lub srebrne jako woreczki ze złotym lub srebrnym pyłem. Jak wtedy należy rozwiązać problem plecakowy Alibaby?

**5.10.** Kiedy w algorytmie plecakowym żadne krawędzie drzewa przeszukiwań nie będą obcinane? Jaka jest wtedy złożoność czasowa algorytmu mierzona liczbą podejmowanych prób?

**5.11.** Opracuj wersję programu rozwiązującego problem ośmiu hetmanów, w którym szachownicę reprezentuje tablica o  $8 \times 8$  elementach typu logicznego. Porównaj jego czas wykonania z programem omówionym w podrozdziale 5.4.

**5.12.** Zmodyfikuj algorytm rozwiązywania problemu ośmiu hetmanów tak, by wyznaczał liczbę podejmowanych prób (wywołań procedury *Probu*). Jaka byłaby liczba podejmowanych prób, gdyby nie obcinać drzewa przeszukiwań?

**5.13.** Zaprezentowany algorytm rozwiązywania problemu ośmiu hetmanów nie rozpoznaje rozwiązań symetrycznych. Zmodyfikuj ten algorytm tak, by podawał jedynie rozwiązania istotnie różne.

**5.14.** Dana jest macierz o  $m$  wierszach i  $n$  kolumnach, której elementami są liczby całkowite. Opracuj algorytm przechodzenia od lewego górnego rogu tej

---

<sup>12</sup> Ten i dwa inne algorytmy generowania permutacji można znaleźć w książce [18].

macierzy do jej prawego dolnego rogu tak, by suma wartości elementów, przez które wiedzie droga, była maksymalna. Obowiązują następujące zasady przechodzenia:

- ♦ w każdym kroku wolno przejść do elementu sąsiedniego na lewo, na prawo, w górę lub w dół,
- ♦ nie wolno przechodzić przez ten sam element więcej niż raz.

**5.15.** Na prostokątnej planszy o wymiarach  $m \times n$  utworzony został labirynt poprzez zabudowanie jej niektórych pól. Na jednym z wolnych (niezabudowanych) pól znajduje się małpa, która chce wydostać się z labiryntu. Może ona poruszać się tylko po wolnych polach, przeskakując na pola sąsiednie w kierunkach: w lewo, prawo, górę i dół. Opracuj algorytm znajdujący najkrótszą drogę wyjścia małpy z labiryntu. Przyjmij, że polami planszy są znaki spacji, oznaczające pola wolne, i znaki X, oznaczające pola zabudowane.

**5.16.** Problem **kolorowania grafu** nieskierowanego  $G = (V, E)$  polega na przypisaniu każdemu wierzchołkowi  $u \in V$  koloru (dodatniej liczby całkowitej) tak, by sąsiednie wierzchołki miały różne kolory i liczba wykorzystanych kolorów była jak najmniejsza. Opracuj algorytm, który metodą prób i błędów koloruje wierzchołki danego grafu reprezentowanego przez macierz sąsiedztwa. Jaka jest złożoność obliczeniowa tego algorytmu?

**5.17.** **Cyklem** lub **drogą Hamiltona** w grafie  $G = (V, E)$  nazywamy ścieżkę zamkniętą przechodzącą dokładnie raz przez każdy wierzchołek  $u \in V$ . Opracuj algorytm, który metodą prób i błędów znajduje w danym grafie nieskierowanym  $G$  cykl Hamiltona lub wykazuje, że graf  $G$  nie ma takiego cyklu. Jaka jest złożoność obliczeniowa tego algorytmu?

**5.18.** **Kliką** w grafie nieskierowanym  $G = (V, E)$  nazywamy taki jego podgraf  $G' = (V', E')$ , w którym dla każdej pary wierzchołków  $u, v \in V'$  istnieje łącząca je krawędź  $(u, v) \in E'$ . Opracuj algorytm, który metodą prób i błędów znajduje w danym grafie nieskierowanym największą, w sensie zawierania zbiorów wierzchołków i krawędzi, klikę. Jaka jest złożoność obliczeniowa tego algorytmu?

**5.19.** Wzorując się na omówionym w podrozdziale 5.6 interpretatorze wyrażeń arytmetycznych, opracuj interpretator wyrażeń logicznych. Powinien on umożliwić wykonywanie co najmniej trzech podstawowych operacji logicznych (suma, iloczyn, negacja) na wartościach logicznych 0, 1 (fałsz, prawda).

**5.20.** Istnieje kilka prostych możliwości udoskonalenia interpretatora wyrażeń arytmetycznych:

- 1) wprowadzenie podstawowych funkcji matematycznych (kwadrat, pierwiastek kwadratowy, sinus, cosinus),
- 2) uwzględnienie potęgowania (można go oznaczyć symbolem  $^$ , ma ono priorytet wyższy niż inne działania, jest prawostronnie łączne),
- 3) wprowadzenie instrukcji wypisywania wartości wyrażenia (zamiast wypisywania wartości wszystkich zmiennych),
- 4) rozróżnianie wartości typu całkowitego i rzeczywistego.

Opracuj wersję interpretatora uwzględniającą co najmniej jedno z wymienionych rozszerzeń. Jaki problem rodzi wprowadzenie funkcji dwuargumentowych?



## Dodatek.

### Programowanie wizualne w Delphi

Delphi jest nowoczesnym narzędziem szybkiego tworzenia aplikacji – RAD (ang. *Rapid Application Development*) opartym na języku Pascal. **Zintegrowane środowisko programistyczne** – IDE (ang. *Integrated Development Environment*) Delphi jest mocno rozbudowane. Zawiera szereg narzędzi pozwalających na wizualne projektowanie, uruchamianie i testowanie aplikacji. Duża liczba okien i zawartych w nich elementów sterujących może na początku wprowadzić użytkownika w stan dezorientacji i zagubienia. Nie musi się on jednak obawiać, ponieważ IDE Delphi jest intuicyjne i łatwe do zrozumienia.

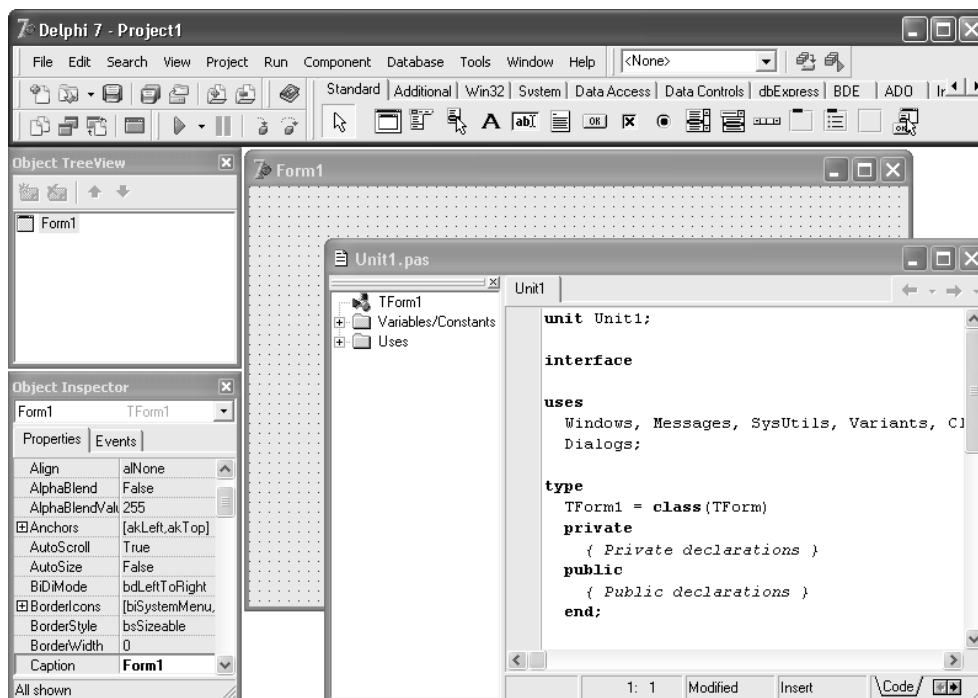
Elementami sterującymi IDE są: menu, przyciski, pola opcji, pola wyboru, pola listy, pola tekstowe, paski przewijania i inne obiekty stanowiące tzw. **graficzny interfejs użytkownika** – GUI (ang. *Graphical User Interface*). Często są one zgrupowane na paskach narzędziowych i zakładkach. Za ich pomocą można wykonywać różne zadania, jak np. otwieranie i zamykanie plików, budowanie, kompilowanie i uruchamianie aplikacji, dostosowywanie środowiska do własnych potrzeb i upodobań. Oprócz menu głównego można korzystać z menu podręcznego, przywoływanego prawym przyciskiem myszy. Wiele poleceń jest dostępnych za pomocą skrótów klawiaturowych. Paski narzędziowe, podobnie jak w programach popularnego pakietu Microsoft Office, dają się łatwo przemieszczać i konfigurować, a funkcje zamieszczonych na nich elementów sterujących są objaśniane za pomocą krótkich podpowiedzi pojawiających się na chwilę, gdy kursor myszy zostanie nad nimi zatrzymany.

#### D.1. Elementy i konfiguracja środowiska

Wygląd IDE zależy od wersji Delphi i ustawień dokonanych przez użytkownika, a nawet od wykorzystywanej wersji systemu Windows [25]. Zasadnicze elementy różnych odmian środowiska są jednak niemal identyczne. W przypadku Delphi 7 standardowo jest wyświetlanych pięć okien (rys. D.1):

- ♦ **okno główne**, zatytułowane wstępnie *Delphi 7 – Projekt1*, obejmujące menu, paski narzędziowe i paletę komponentów,
- ♦ **okno drzewa komponentów**, zatytułowane *Object TreeView*, ukazujące graficznie zależności pomiędzy wykorzystywanymi komponentami i pozwalające na szybki ich wybór,

- ♦ **okno inspektora obiektów**, zatytułowane *Object Inspector*, umożliwiające w fazie projektowej modyfikację właściwości komponentów i określanie metod obsługi zdarzeń generowanych podczas wykonywania aplikacji,
- ♦ **okno projektanta formularza**, zwane krótko **formularzem** i zatytułowane wstępnie *Form1*, pozwalające na niezwykle wygodne, wizualne projektowanie okna aplikacji i graficznego interfejsu użytkownika,
- ♦ **okno edytora kodu**, zatytułowane wstępnie *Unit1.pas*, zawierające kod źródłowy modułu związanego z formularzem i umożliwiające jego edycję.



Rys. D.1. Środowisko Delphi 7 w systemie Windows XP

Edycja kodu źródłowego aplikacji wymaga często przełączania dwóch okien zajmujących to samo miejsce na ekranie – okna edytora kodu na okno projektanta formularza, i odwrotnie. Niekiedy pomaga tu Delphi, udostępniając jedno z nich zgodnie z oczekiwaniami programisty, ale nieraz musi to robić on sam. Najwygodniej jest wówczas skorzystać z przycisku *Toggle Form/Unit* (przełącz formularz/moduł) znajdującego się na pasku narzędziowym poniżej menu (rys. D.2) lub posłużyć się klawiszem *F12*. Pasek zawiera szereg innych użytecznych przycisków, m.in. *New Items* (nowe elementy aplikacji), *Open* (otwórz plik), *Save* (zapisz

plik), *Save All* (zapisz wszystkie pliki), *Run* (uruchom aplikację), *Help* (pomoc kontekstowa). Można go też dowolnie konfigurować, korzystając z okna dialogowego *Customize* (dostosuj) przywoływanego ostatnim poleceniem z rozwiniętej listy *Tollbars* w menu *View*.



Rys. D.2. Przycisk przełączania formularza i kodu modułu

Podczas tworzenia aplikacji oprócz okna głównego korzysta się przede wszystkim z inspektora obiektów, projektanta formularza i edytora kodu. W przypadku prostych projektów okno drzewa komponentów i lewa część edytora kodu<sup>1</sup> są raczej mało przydatne, najlepiej jest więc oba te elementy zamknąć, a okno inspektora obiektów rozciągnąć w górę tak, by zajęło uprzednio zwolnione miejsce, co znacznie ułatwi dostęp do jego zawartości. Oczywiście nic nie stoi na przeszkodzie, aby zamknięte okno ponownie otworzyć. Na przykład niewidoczne okno drzewa komponentów pojawi się, gdy w menu *View* wybierzemy polecenie *Object TreeView*, bądź też gdy naciśniemy kombinację klawiszy *Shift+Alt+F11*<sup>2</sup>.

Skonfigurowany układ okien środowiska, zwany **ustawieniami pulpitu**, można zapisać, korzystając z przycisku *Save current desktop* znajdującego się na pasku narzędziowym po prawej stronie menu (rys. D.3) lub polecenia *Save Desktop* wybranego z rozwiniętej listy *Desktops* w menu *View*. W okienku dialogowym *Save desktop*, które się wtedy pojawi, należy wpisać nazwę i nacisnąć przycisk *OK*.

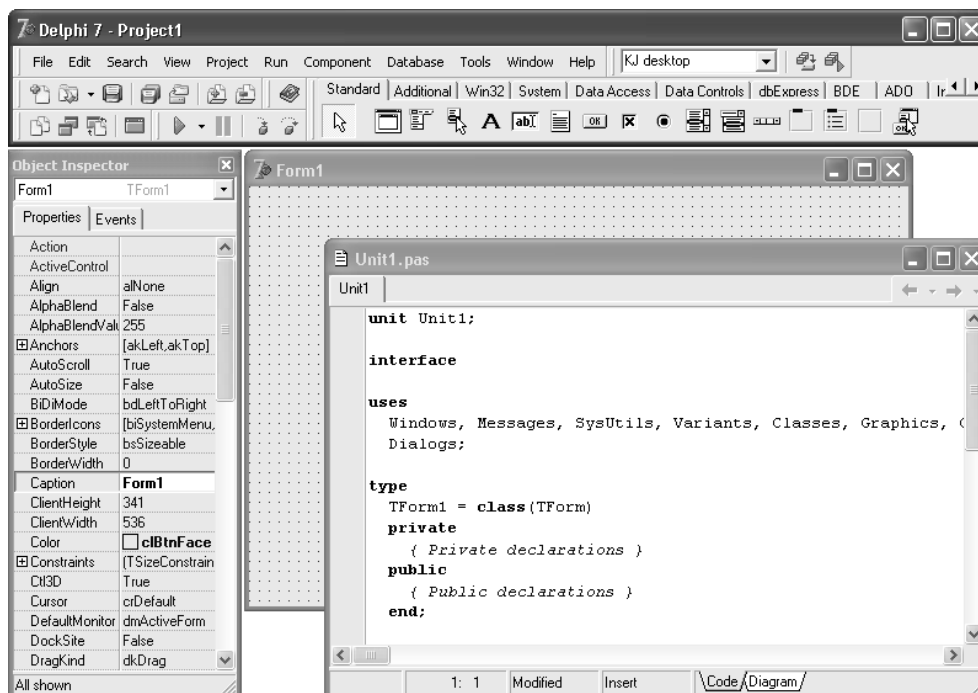


Rys. D.3. Przycisk zapisu ustawień pulpitu

<sup>1</sup> Lewa część edytora kodu jest w rzeczywistości oknem dokowalnym o nazwie *Code Explorer* (eksplorator kodu), które można odczepić od edytora kodu i przesunąć w inne miejsce lub przycumować do innego okna.

<sup>2</sup> Lewy *Alt*. Prawy umożliwia dostęp do polskich liter, ale żeby wszystkie były dostępne, należy w kluczu *HKEY\_CURRENT\_USER\Software\Borland\Delphi\7.0\Editor\Options* rejestru Windows utworzyć nowy klucz o nazwie *NoCtrlAltKeys* i wartości 1.

Przykładowa konfiguracja Delphi 7, zapisana pod nazwą *KJ desktop*, jest pokazana na rysunku D.4. Przypomina ona bardzo Delphi 3 i wydaje się wprost idealna dla osób rozpoczynających przygodę z Delphi.



Rys. D.4. Środowisko Delphi 7 skonfigurowane jak Delphi 3

Ustawień pulpitu może być więcej niż jedno. Wszystkie są wyszczególnione na liście rozwijanej usytuowanej na lewo od przycisku *Save current desktop*. Każdorazowo, gdy Delphi zostanie uruchomione, odtworzy ustawienia pulpitu zgodnie z ostatnio wybraną pozycją listy. Pozycja *<None>* oznacza, że użytkownik nie wybrał żadnego ustawienia i środowisko ma przyjąć wygląd standardowy.

## D.2. Klasy, obiekty, komponenty

Programowanie wizualne jest metodą tworzenia programu za pomocą narzędzi umożliwiających wybór standardowych elementów sterujących, zwanych **komponentami** lub **składnikami**, i automatyczną generacją kodu. Komponenty realizują rozmaite funkcje i są szczególnym przypadkiem tzw. **obiektów**. Delphi spełnia oba warunki programowania wizualnego. Zawiera obszerną bibliotekę komponentów

o nazwie VCL (ang. *Visual Component Library*)<sup>3</sup> i automatyzuje proces tworzenia kodu źródłowego. Językiem programowania w Delphi jest *Object Pascal*, nazywany od wersji 7 środowiska *językiem Delphi*<sup>4</sup>. Programując w Delphi operujemy na obiektach, aczkolwiek na początku możemy nawet nie mieć pojęcia o zasadach programowania zorientowanego obiektowo – OOP (ang. *Object Oriented Programming*), nazywanego krócej programowaniem obiektowym.

Definicja typu obiektowego, zwanego **klasą**, określa **polą** i **metody** obróbki przechowywanych w tych polach danych. Metody realizują rozmaite zadania związane z funkcjonowaniem obiektu. Mogą być nimi zarówno procedury, jak i funkcje. Klasa jest definiowanym przez użytkownika typem danych stanowiącym formalnie rozszerzenie typu rekordowego, ponieważ oprócz wewnętrznych danych (pola) określających jej stan zawiera operacje (metody) określające jej zachowanie. Ścisłej mówiąc, klasa opisuje stan i zachowanie obiektów będących jej reprezentantami, zajmujących pewne obszary w pamięci komputera. Innymi słowy, klasa jest typem, a obiekt zmienną tego typu.

Zgodnie z filozofią programowania obiektowego, pola klasy powinny być poza nią niedostępne, a wszelkie operacje na nich powinny być wykonywane za pomocą metod. Zasada ta ma na celu uniknięcie przypadkowej zmiany danych obiektu, która prowadziłaby do nieprzewidzianego jego zachowania. Obiekt można sobie wyobrazić jako hermetycznie zamkniętą „czarną skrzynkę”, której tylko niektóre elementy wystają na zewnątrz. Te elementy to metody tworzące tzw. interfejs, który pozwala na komunikowanie się z obiektem i wpływanie na jego stan.

**Komponent** jest obiektem realizującym określone zadanie, dostępnym do wykorzystania w trakcie projektowania programu w postaci ikony umieszczanej na formularzu lub tworzoną dynamicznie podczas wykonywania programu. Cechy komponentu i jego zachowanie określają **właściwości** (ang. *properties*), metody i **zdarzenia** (ang. *events*).

Właściwości mogą być związane z polami lub metodami. Właściwość związana z polem może dotyczyć nie tylko wartości tego pola, lecz także sposobu jej odczytywania i modyfikacji. Oznacza to, że zmiana wartości właściwości może pociągać, oprócz zmiany wartości takiego pola, wykonanie pewnych dodatkowych czynności (metody), np. kontrolę przypisywanej polu wartości lub przemałowanie komponentu na ekranie. Z kolei zdarzenia określają czynności, jakie mają być pod-

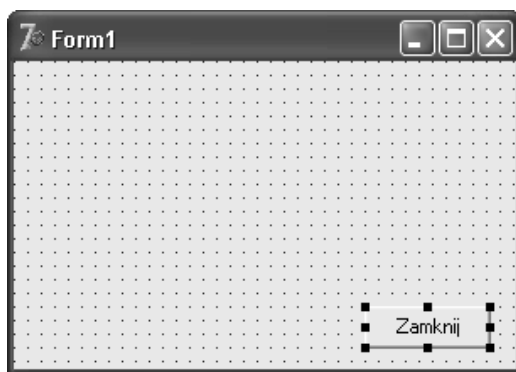
---

<sup>3</sup> Od wersji 6 Delphi dostępna jest również druga biblioteka komponentów – CLX (wym. „klik”, skrót od ang. *Component Library X-Platform for Cross Platform*), wspólna dla systemów Windows i Linux [6].

<sup>4</sup> Prawdopodobnie zmiana ta nastąpiła ze względów marketingowych, ponieważ Pascal nigdy nie był językiem popularnym w USA, w przeciwieństwie do Europy i wielu innych rejonów świata.

jęte w pewnych sytuacjach podczas wykonania programu, np. takich jak kliknięcie przyciskiem myszy. W istocie są one właściwościami związanymi z metodami. Aby obsłużyć zdarzenie, należy przypisać mu na etapie projektowania aplikacji lub podczas jej wykonania odpowiednią metodę – **procedurę obsługi zdarzenia**.

Przykładem komponentu w Delphi jest przycisk zamieszczony na formularzu (rys. D.5). Jest on obiektem standardowej klasy *TButton*, posiadającym szereg właściwości, m.in. *Name* (nazwa), *Width* i *Height* (szerokość i wysokość), *Left* i *Top* (odległość od lewej i górnej krawędzi obszaru roboczego formularza) oraz *Caption* (etykieta na przycisku). Metodami przycisku są np. *Show* i *Hide* (pokazanie i ukrycie), a najczęściej wykorzystywanym zdarzeniem jest *OnClick* (kliknięcie lewym przyciskiem myszy). W momencie wstawienia przycisku do formularza wartością domyślną właściwości *Caption* określoną przez Delphi była etykieta *Button1*, ale programista zmienił ją na *Zamknij*.



Rys. D.5. Formularz z zamieszczonym na nim przyciskiem

### D.3. Paleta komponentów i jej wykorzystanie

Większość komponentów biblioteki VCL Delphi jest dostępna w fazie projektowania aplikacji za pośrednictwem **palety komponentów** usytuowanej w dolnej części okna głównego (rys. D.6). Paleta komponentów jest podzielona tematycznie na szereg podłużnych kart wybieranych za pomocą zakładek: *Standard* (podstawowe komponenty interfejsu użytkownika), *Additional* (dodatkowe komponenty interfejsu użytkownika), *Win32* (komponenty charakterystyczne dla 32-bitowych systemów Windows), *System* (komponenty udostępniające funkcje systemowe), *Dialogs* (komponenty reprezentujące standardowe okna dialogowe) itd. Liczba kart palety i zamieszczonych na nich komponentów, widocznych jako przyciski oznaczone ikonami, zależy od wersji i ustawień środowiska Delphi.



Rys. D.6. Paleta komponentów VCL

Chociaż w czasie projektowania aplikacji komponenty mają postać widocznych obiektów, którymi można manipulować, dzielimy je na dwie kategorie: **widzialne** i **niewidzialne**<sup>5</sup>. Pierwszą stanowią komponenty widoczne podczas wykonywania aplikacji, np. *Button* (przycisk), *CheckBox* (pole wyboru), *Edit* (pole edycji), *Label* (etykieta), *ListBox* (pole listy), *Panel* (panel grupujący inne obiekty), *RadioButton* (przycisk opcji), *RadioGroup* (grupa przycisków opcji), *ScrollBar* (pasek przewijania), *StringGrid* (siatka łańcuchów złożona z wierszy i kolumn) itp. Są to tzw. **kontrolki** (ang. *controls*), z których jest tworzony graficzny interfejs użytkownika (GUI). Komponenty drugiej kategorii nie mają reprezentacji wizualnej w czasie wykonywania aplikacji, pełnią jedynie różne funkcje sterujące. Przedstawicielem tej kategorii jest komponent *Timer* (zegar), który generuje z jednakową częstotliwością zdarzenia, pozwalając zaprogramować wykonanie pewnych operacji w jednakowych odstępach czasu. Można go np. wykorzystać w animacji.

Umieszczenie komponentu na formularzu jest wyjątkowo proste. Wystarczy kliknąć jego ikonę na palecie komponentów, a następnie kliknąć w dowolnym miejscu na formularzu. Można również dwukrotnie kliknąć ikonę komponentu na palecie, a pojawi się on automatycznie pośrodku formularza. Wstawiony komponent można przesunąć myszą w inne miejsce, a jeśli reprezentuje kontrolkę, można też zmienić jego rozmiar, przeciągając uchwyty (małe czarne kwadraciki) usytuowane na jego brzegu (zob. rys. D.5). Komponent jest „przyciągany” do węzłów siatki wyświetlanych na formularzu, dlatego zmiany są skokowe (domyślnie co 8 pikseli). Obie te operacje można wykonać precyzyjniej (co 1 piksel) za pomocą klawiszy strzałek przy wciśniętym klawiszu *Ctrl* (przesuwanie) lub *Shift* (zmiana rozmiaru), a także poprzez ustawienie właściwości *Left* i *Top* lub *Width* i *Height* w oknie inspektora obiektów. Zbędny komponent można usunąć z formularza za pomocą klawisza *Delete*.

Nie są to bynajmniej jedyne sposoby wstawiania, przesuwania, zmiany rozmiaru i usuwania komponentów ani jedyne tego typu operacje. Komponenty można wyrównywać, kopiować do schowka, wycinać i wklejać oraz ustalać ich kolejność, można również wykonywać różne funkcje na grupach komponentów [6]. Umożliwiają to m.in. odpowiednie klawisze skrótów oraz polecenia w menu *Edit* i *View*.

<sup>5</sup> Można spotkać podział komponentów na wizualne i niewizualne (ang. *visual* i *nonvisual*), ale taka terminologia nie wydaje się trafna, ponieważ nazwa biblioteki VCL sugeruje, że wszystkie zawarte w niej komponenty są wizualne.

## D.4. Inspektor obiektów, właściwości i zdarzenia

Niezwykle ważną rolę w fazie projektowania aplikacji pełni okno inspektora obiektów (rys. D.7). Współpracuje ono z oknem projektanta formularza i edytora kodu, pozwalając na odczytywanie i łatwe modyfikowanie właściwości i definiowanie zdarzeń formularza i umieszczanych na nim komponentów.



Rys. D.7. Właściwości przycisku w oknie inspektora obiektów

Okno inspektora obiektów jest podzielone na dwie karty wybierane za pomocą zakładek *Properties* (właściwości) i *Events* (zdarzenia). Obie karty mają postać dwukolumnowej tabeli. W kolumnie lewej podany jest wykaz właściwości lub zdarzeń, a w kolumnie prawej ich wartości. W górnej części okna znajduje się lista rozwijana ukazująca nazwę i typ bieżącego obiektu, którego właściwości i zdarzenia są wyświetlone niżej. Wybranie obiektu z listy powoduje nie tylko wyświetlenie wykazu jego właściwości i zdarzeń, lecz także wybranie go na formularzu. Podobnie, wybranie formularza lub znajdującego się na nim komponentu powoduje wyświetlenie nazwy tego obiektu w polu listy oraz wykazu jego właściwości i zdarzeń. Jeżeli zostanie wybranych na raz kilka obiektów, w polu listy będzie pokazana ich liczba, a zestaw właściwości i zdarzeń będzie wspólny dla nich wszystkich.



Z tego udogodnienia można skorzystać w przypadku nadawania takich samych właściwości lub zdarzeń wielu komponentom jednocześnie.

Na karcie właściwości wyszczególnione są te właściwości, których wartości można modyfikować w fazie projektowania aplikacji. Wiele z nich ma wstępnie ustawione wartości domyślne. Niektóre właściwości wymagają wprowadzenia liczby, inne łańcucha, jeszcze inne wybrania wartości z dopuszczalnej listy ustawień, są też i takie, które trzeba ustawiać za pomocą okna dialogowego. Istnieją także **właściwości zagnieżdżone**, które składają się z innych właściwości. Są one oznaczone krzyżykiem (+) lub minusem (–). Kliknięcie krzyżyka lewym przyciskiem myszy powoduje rozwinięcie właściwości zagnieżdżonej i wyświetlenie minusa, a kliknięcie minusa zwinięcie jej i wyświetlenie krzyżyka.

Do najważniejszych właściwości dostępnych w inspektorze obiektów, wspólnych dla wielu komponentów, należą:

- ◆ *Align* – sposób wyrównania kontrolki w obrębie kontrolki nadrzędnej,
- ◆ *Caption* – napis na kontrolce (etykieta),
- ◆ *Color* – kolor tła kontrolki,
- ◆ *Enabled* – komponent aktywny (*true*) lub nieaktywny (*false*),
- ◆ *Font* – właściwości czcionki (krój, rozmiar, kolor i inne atrybuty),
- ◆ *Height* – wysokość kontrolki (liczba pikseli),
- ◆ *Hint* – tekst podpowiedzi,
- ◆ *Left* – odległość od lewego brzegu kontrolki nadrzędnej (liczba pikseli),
- ◆ *Name* – nazwa komponentu wykorzystywana w programie (zbudowana zgodnie z regułami tworzenia identyfikatorów w Pascalu),
- ◆ *ShowHint* – włączenie (*true*) lub wyłączenie (*false*) wyświetlania tekstu podpowiedzi,
- ◆ *TabOrder* – kolejność przechodzenia do kontrolki za pomocą klawisza *Tab*,
- ◆ *Text* – tekst zmienny przechowywany w kontrolce reprezentującej pole edycji (wprowadzany z klawiatury lub zmieniany programowo),
- ◆ *Top* – odległość od górnego brzegu kontrolki nadrzędnej (liczba pikseli),
- ◆ *Visible* – kontrolka widoczna (*true*) lub ukryta (*false*),
- ◆ *Width* – szerokość kontrolki (liczba pikseli).

W przypadku formularza właściwości *Align*, *Left* i *Top* dotyczą położenia reprezentowanego przez niego okna na ekranie monitora. Formularz ma jeszcze kilka specyficznych mu właściwości, m.in.:

- ◆ *BorderIcons* – zestaw ikon wyświetlanych na pasku tytułowym okna,
- ◆ *BorderStyle* – rodzaj ramki okna,

- ♦ *ClientHeight* – szerokość obszaru roboczego okna (liczba pikseli),
- ♦ *ClientWidth* – wysokość obszaru roboczego okna (liczba pikseli),
- ♦ *WindowState* – stan okna (normalne, zminimalizowane, zmaksymalizowane).

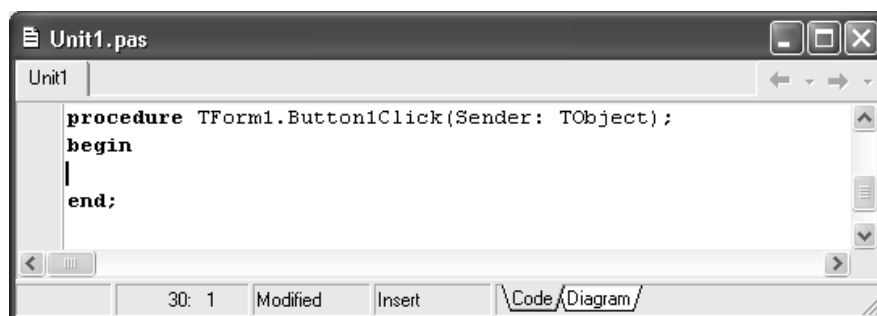
Na karcie zdarzeń inspektora obiektów wstępnie zdarzeniom nie są przypisane żadne wartości domyślne (rys. D.8). W celu zdefiniowania procedury obsługi zdarzenia wystarczy dwukrotnie kliknąć w pustym polu obok nazwy tego zdarzenia, a Delphi udostępni okno edytora kodu z wygenerowanym przezeń szablonem metody, który należy uzupełnić stosownym kodem w języku Object Pascal.



**Rys. D.8.** Zdarzenia przycisku w oknie inspektora obiektów

Na przykład dla zdarzenia *OnClick* przycisku o nazwie *Button1* (właściwość *Name*) i etykiecie *Zamknij* (właściwość *Caption*) w oknie edytora kodu zostanie umieszczona pusta procedura przedstawiona na rysunku D.9. Jeśli jej kod uzupełnimy, wstawiając instrukcję wywołania bezparametrowej metody *Close* formularza, to gdy aplikacja zostanie uruchomiona, kliknięcie w jej oknie na przycisku *Zamknij* spowoduje zamknięcie okna i zakończenie wykonania aplikacji<sup>6</sup>.

<sup>6</sup> Aplikacja zostanie zakończona tylko wtedy, gdy zamykane okno jest jej oknem głównym.



**Rys. D.9.** Szablon procedury obsługi zdarzenia w oknie edytora kodu

W momencie zdefiniowania procedury obsługi zdarzenia w oknie inspektora obiektów po prawej stronie nazwy zdarzenia pojawi się nazwa skojarzonej z tym zdarzeniem metody. W przytoczonym wyżej przykładzie procedurą obsługi zdarzenia *OnClick* przycisku *Button1* jest metoda *Button1Click* klasy *TForm1*, toteż nazwa *Button1Click* pojawi się obok nazwy *OnClick*. Ponowne podwójne kliknięcie na wypełnionym polu obok nazwy zdarzenia powoduje przejście do okna edytora kodu i udostępnienie kodu skojarzonej ze zdarzeniem metody.

Alternatywnym sposobem definiowania procedury obsługi zdarzenia jest podwójne kliknięcie na komponencie w oknie projektanta formularza bądź wpisanie nazwy w polu obok nazwy zdarzenia w oknie inspektora obiektów i naciśnięcie klawisza *Enter*. Można też do obsługi kilku zdarzeń użyć tej samej procedury, co prowadzi do krótszego kodu programu. W tym celu należy kliknąć strzałkę z prawej strony nazwy zdarzenia i z rozwiniętej listy kompatybilnych metod wybrać jedną z nich, oczywiście pod warunkiem, że lista ta nie jest pusta.

Aby odłączyć procedurę obsługi od obsługiwanego przez nią zdarzenia, można usunąć jej nazwę z odpowiedniego pola w oknie inspektora obiektów. Kod procedury nadal pozostaje w programie, można go więc ponownie podłączyć do tego zdarzenia lub wykorzystać do obsługi innych zdarzeń. Jeżeli kod procedury jest zbędny, należy usunąć z niego tylko te fragmenty, które wprowadził programista, tj. doprowadzić do takiej samej postaci, jaką Delphi automatycznie wygenerowało podczas definiowania procedury obsługi zdarzenia. Przy zapisywaniu na dysku lub kompilowaniu programu kod wszystkich pustych metod wraz z odwołaniami do nich w inspektorze obiektów zostanie usunięty. Wykorzystując tę technikę nie musimy odłączać procedury od zdarzenia, zostanie to zrobione automatycznie.

Podobnie jak w przypadku właściwości, szereg zdarzeń dostępnych w inspektorze obiektów jest wspólnych dla wielu komponentów. Należą do nich m.in.:

- ◆ *OnChange* – zmiana danych przechowywanych w komponencie,
- ◆ *OnClick* – kliknięcie lewym przyciskiem myszy,

- ◆ *OnDblClick* – podwójne kliknięcie lewym przyciskiem myszy,
- ◆ *OnKeyDown* – naciśnięcie klawisza,
- ◆ *OnKeyPress* – naciśnięcie klawisza odpowiadającego znakowi ASCII,
- ◆ *OnKeyUp* – zwolnienie klawisza,
- ◆ *OnMouseDown* – naciśnięcie jednego z przycisków myszy,
- ◆ *OnMouseMove* – przesunięcie kursora myszy nad kontrolką,
- ◆ *OnMouseUp* – zwolnienie wciśniętego przycisku myszy.

Oprócz zdarzeń typowych dla większości komponentów formularz ma szereg zdarzeń mu specyficznych. Do najważniejszych z nich należą:

- ◆ *OnClose* – zamykanie okna,
- ◆ *OnCreate* – tworzenie okna,
- ◆ *OnDestroy* – likwidacja okna,
- ◆ *OnPaint* – rysowanie zawartości obszaru roboczego okna,
- ◆ *OnResize* – zmiana rozmiaru okna.

W inspektorze obiektów **opublikowane** (ang. *published*) są tylko te właściwości i zdarzenia, które są wizualnie dostępne w czasie projektowania aplikacji. Oczywiście możliwe jest wykorzystanie kodu w Object Pascalu, by w trakcie jej wykonywania ustawiać właściwości, przypisywać zdarzeniom metody obsługi lub je odłączać. Zazwyczaj obiekty mają także właściwości, zdarzenia i metody, które nie są opublikowane, tj. takie, które nie są dostępne w interfejsie projektowym. Ich wykorzystanie wymaga zamieszczenia odpowiedniego kodu w programie.

Najważniejszymi nieopublikowanymi właściwościami obiektów, dostępnymi tylko podczas wykonywania aplikacji, są:

- ◆ *Owner* – komponent odpowiedzialny za zniszczenie obiektu (właściciel),
  - ◆ *Parent* – kontrolka nadrzędna (rodzic), na której obiekt jest umiejscowiony,
- a najważniejszymi metodami nieopublikowanymi:
- ◆ *Create* – utworzenie obiektu (metoda specjalna, **konstruktor** obiektu),
  - ◆ *Destroy* – zniszczenie obiektu i zwolnienie zajmowanej przez niego pamięci (metoda specjalna, **destruktor** obiektu),
  - ◆ *Execute* – wyświetlenie standardowego okna dialogowego, np. *OpenDialog*,
  - ◆ *Free* – bezpieczne zniszczenie obiektu (przed wywołaniem destruktora metoda sprawdza, czy obiekt istnieje),
  - ◆ *Hide* – ukrycie kontrolki,
  - ◆ *Invalidat*e – powiadomienie systemu Windows, że powierzchnia kontrolki zawiera nieaktualne informacje i wymaga przemalowania,

- ♦ *SetFocus* – kontrolka staje się odbiorcą komunikatów generowanych przez klawiaturę (otrzymuje tzw. fokus),
- ♦ *Show* – wyświetlenie kontrolki,
- ♦ *ShowModal* – wyświetlenie okna w trybie modalnym.

Niektóre obiekty udostępniają podczas wykonywania aplikacji bardzo specyficzną właściwość zagnieżdżoną o nazwie **Canvas** (płótno malarskie, powierzchnia rysowania) umożliwiającą kreślenie tekstu, linii, prostokątów, elips, łuków, wycinków eliptycznych, łamanych, wielokątów i wielu innych elementów graficznych [5, 6, 14, 25]. W szczególności obiektami takimi są formularze oraz komponenty *Image* (obraz) i *PaintBox* (prostokątny obszar malowania). *Canvas* jest obiektem klasy *TCanvas* dysponującym m.in. następującymi właściwościami:

- ♦ *Brush* – pędzel do wypełniania zamkniętych obszarów,
- ♦ *Font* – czcionka do pisania tekstów,
- ♦ *Pen* – pióro do kreślenia linii,
- ♦ *PenPos* – bieżąca pozycja pióra (współrzędne punktu),
- ♦ *Pixels* – kolory punktów obrazu (tablica),

a także metodami, z których najważniejszymi są:

- ♦ *Draw* – rysowanie bitmapy,
- ♦ *Ellipse* – rysowanie elipsy,
- ♦ *LineTo* – rysowanie linii od bieżącej pozycji pióra do określonego punktu,
- ♦ *MoveTo* – przeniesienie pióra do określonego punktu,
- ♦ *Rectangle* – rysowanie prostokąta,
- ♦ *TextOut* – rysowanie tekstu.

W celu uzyskania pełnej informacji na temat właściwości, zdarzeń i metod komponentów biblioteki VCL można posłużyć się systemem pomocy ekranowej Delphi. Pomoc przywołuje się za pomocą poleceń menu *Help* lub poprzez naciśnięcie klawisza *F1*.

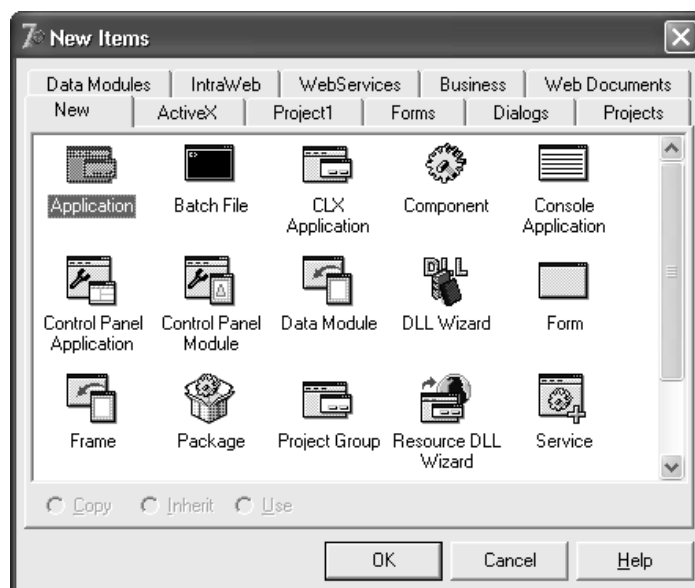
## D.5. Pliki tworzone przez środowisko

Projektując wizualnie aplikację umieszczamy na jej formularzach komponenty, określamy ich niektóre właściwości i definiujemy odpowiednie zdarzenia. Delphi automatycznie dopisuje podstawową część potrzebnego kodu, my go tylko uzupełniamy. Dla każdego formularza tworzony jest **moduł** (ang. *unit*) kodu źródłowego w Object Pascalu. Nie jest to jednak jedyny plik związany z formularzem ani tym bardziej jedyny plik projektu budowanej aplikacji. Delphi tworzy wiele plików

o różnym przeznaczeniu, nadając im nazwy projektu lub modułów i predefiniowane rozszerzenia. Najważniejszymi z nich są:

- ♦ \*.dpr – plik główny programu (projektu) w Object Pascalu,
- ♦ \*.res – binarny plik zasobów programu (najczęściej zawiera ikonę aplikacji),
- ♦ \*.pas – plik z kodem źródłowym modułu w Object Pascalu,
- ♦ \*.dfm – plik definiujący formularz (od Delphi 5 tekstowy, wcześniej binarny),
- ♦ \*.exe – wynikowy plik wykonywalny aplikacji,
- ♦ \*.dcu – wynikowy plik modułu (binarny).

Oprócz wyżej wymienionych generowane są pliki zawierające opcje projektu, informacje konfiguracyjne kompilatora, kopie zapasowe o rozszerzeniach rozpoczynających się od tyldy (~) i wiele innych, z których część nie jest nawet udokumentowana [6, 14]. Ponieważ w trakcie tworzenia projektu powstaje duża liczba plików, wskazane jest zapisywanie każdego projektu w oddzielnym katalogu. Efektem końcowym kompilacji jest plik \*.exe, który można uruchamiać z poziomu systemu Windows poza środowiskiem Delphi<sup>7</sup>. Aby utworzyć kopię zapasową projektu, wystarczy skopiować pliki \*.dpr, \*.res, \*.pas i \*.dfm.



Rys. D.10. Składnica obiektów Delphi

<sup>7</sup> Wynikiem kompilacji może być również plik \*.dll, którego zawartością jest kod binarny biblioteki dołączanej dynamicznie (ang. *Dynamic Linked Library*) do różnych programów.

**Aplikacja okienkowa** ma co najmniej jeden formularz umożliwiający przyjazne komunikowanie się z nią (GUI). Z każdym formularzem jest związana para plików \*.pas i \*.dfm. **Aplikacja konsolowa** nie ma formularzy, nie zawiera więc plików \*.dfm. Komunikacja z nią odbywa się za pomocą procedur *Read* i *ReadLn* oraz *Write* i *WriteLn*, podobnie jak w tradycyjnym programie w Turbo Pascalu. Oczywiście, zarówno aplikacja okienkowa, jak i konsolowa może zawierać moduły \*.pas nieskojarzone z formularzami.

Delphi przy każdym uruchomieniu tworzy nową aplikację okienkową. Istnieje wiele poleceń menu, które pozwalają tworzyć nowe aplikacje, moduły, formularze i inne elementy programowe. Wszystkie te polecenia są dostępne po rozwinięciu pozycji *New* w menu *File*. Jeśli na przykład wybierzemy polecenie *Application*, Delphi utworzy nową aplikację okienkową, a jeśli wybierzemy *Other*, utworzy okno dialogowe *New Items* (nowe elementy), zwane **składnicą** lub **repozytorium obiektów** (ang. *Object Repository*), które zawiera wiele kart obejmujących wszystkie elementy programowe, jakie można utworzyć (rys. D.10). Jednym z nich jest aplikacja konsolowa (ang. *Console Application*). Składnicę można rozbudowywać o nowe projekty, formularze i moduły [6].

Polecenia zawarte w menu *File* umożliwiają również zapisywanie plików projektu na dysku i ich otwieranie. Pierwszą grupę tworzą polecenia *Save* (zapisanie modułu), *Save As* (zapisanie modułu pod inną nazwą), *Save Project As* (zapisanie projektu pod inną nazwą) i *Save All* (zapisanie wszystkich otwartych plików), zaś drugą *Open* (otwarcie modułu), *Open Project* (otwarcie projektu) i *Reopen* (otwarcie jednego z ostatnio używanych projektów lub modułów).

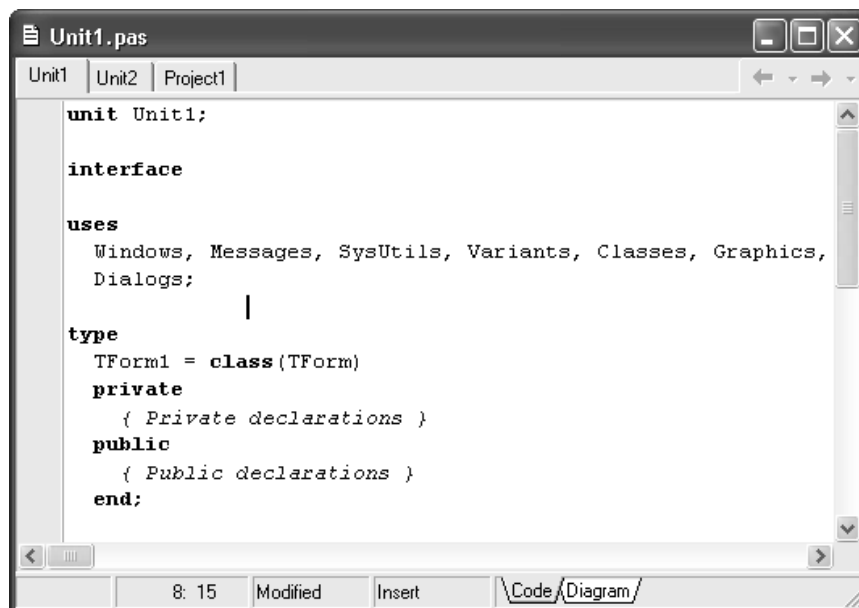
Delphi zawiera prosty mechanizm nazywania tworzonych elementów programowych, polegający na dodawaniu kolejnego numeru do uniwersalnej nazwy elementu. I tak, kolejne projekty nazywa *Project1*, *Project2*, ..., moduły – *Unit1*, *Unit2*, ..., formularze – *Form1*, *Form2*, ..., przyciski – *Button1*, *Button2*, ... itp. Programiści często kierują się wygodnictwem i pozostają przy niesugestywnych nazwach domyślnych. Jeżeli projekt jest mały, nie jest to niebezpieczne, ale gdy staje się większy, łatwo utracić nad nim kontrolę, ponieważ korzystanie z niego wymaga więcej czasu na przeglądanie i zastanawianie się, czego dany fragment kodu dotyczy i jak działa.

Należy pamiętać, że wprowadzonego przez Delphi kodu nie powinno się bezpośrednio w oknie edytora usuwać ani modyfikować, gdyż takie postępowanie może prowadzić do różnych problemów, z którymi trudno sobie poradzić. W szczególności nie wolno w oknie edytora zmieniać nazwy projektu, modułu i komponentu, ani usuwać pustej procedury obsługi zdarzenia. Początkujący programista może sobie pozwolić jedynie na usuwanie komentarzy. Nazwę projektu lub modułu określamy w oknie dialogowym *Save ... As* podczas zapisywania projektu lub modułu na dysku, zaś nazwę komponentu w oknie inspektora obiektów

poprzez ustawienie właściwości *Name*. W obu przypadkach nazwa musi spełniać reguły tworzenia identyfikatorów w Object Pascalu, tj. zaczynać się od litery lub znaku podkreślenia i zawierać tylko symbole należące do zbioru liter, cyfr i znaku podkreślenia. Wielkie i małe litery nie są w nazwach rozróżnialne.

## D.6. Kod źródłowy aplikacji i jego edycja

Okno edytora kodu ma zakładki z nazwami plików (rys. D.11). Umożliwiają one szybkie przechodzenie do edycji kodu źródłowego lub innego tekstu dowolnego pliku wchodzącego w skład aplikacji. Nazwa aktualnie edytowanego pliku jest wyświetlona na pasku tytułowym okna. Pasek stanu usytuowany w dolnej części okna informuje m.in. o bieżącym położeniu kursora (*wiersz: kolumna*), stanie pliku źródłowego (*Modified* – zmodyfikowany, *Read only* – tylko do odczytu) i trybie wpisywania tekstu (*Insert* – wstawianie, *Overwrite* – zamiana), przełączanym za pomocą klawisza *Insert*. Podstawowe polecenia edytora, takie jak zaznaczanie, wycinanie, wklejanie, przeciąganie i kopiowanie tekstu, są dość intuicyjne.



Rys. D.11. Kod źródłowy modułu w oknie edytora kodu

Najczęściej edytowanymi plikami aplikacji są pliki kodu źródłowego modułów związanych z formularzami. Dla każdego nowego formularza Delphi automatycznie tworzy i umieszcza w oknie edytora kod następującej postaci:



```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

Kod rozpoczyna się nagłówkiem określającym nazwę modułu, zawiera dwie części – **sekcję interfejsu** (ang. *interface*) i **sekcję implementacji** (ang. *implementation*)<sup>8</sup>, kończy słowem kluczowym **end** i kropką. W sekcji interfejsu wyszczególnione są te elementy programowe, które są dostępne dla innych modułów programu korzystających z tego modułu, zaś w sekcji implementacji te, które są ukryte przed resztą programu. Sekcja interfejsu stanowi więc część publiczną modułu, a sekcja implementacji jego część prywatną.

Na początku sekcji interfejsu podana jest lista innych modułów, z których dany moduł korzysta (ang. *uses*). Składa się ona z nazw modułów standardowych. Jeśli zachodzi potrzeba, można ją rozszerzyć lub utworzyć drugą w sekcji implementacji. Dalej zdefiniowany jest najważniejszy element programowy modułu, jest nim nowy typ obiektowy – klasa *TForm1* wywodząca się od standardowej klasy *TForm* i opisująca formularz skojarzony z modułem. Nowa klasa dziedziczy wszystkie elementy klasy bazowej i początkowo niczym się od niej nie różni, ale w miarę rozbudowywania formularza dodawane są do niej automatycznie przez Delphi lub bezpośrednio przez programistę nowe pola, metody lub właściwości. Definicja każdej nowej metody, tj. pełny jej kod źródłowy, musi być zamieszczona w sekcji implementacji modułu.

---

<sup>8</sup> W najbardziej ogólnej postaci moduł może jeszcze zawierać sekcję inicjalizacji (ang. *initialization*) i sekcję finalizacji (ang. *finalization*) [6, 14, 25].

Jeżeli na przykład umieścimy na formularzu przycisk i zdefiniujemy dla niego procedurę obsługi zdarzenia *OnClick*, to Delphi rozszerzy definicję klasy *TForm1* o dwie deklaracje – deklarację pola *Button1* i deklarację metody *Button1Click*, przez co definicja klasy przyjmie postać następującą:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Jednocześnie w sekcji implementacji modułu umieści domyślny szablon metody *Button1Click* postaci (por. rys. D.9):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

Deklaracje nowych pól i metod można wpisywać w miejscach oznaczonych komentarzami, tj. w **sekcji prywatnej** (ang. *private*) lub **sekcji publicznej** (ang. *public*) klasy<sup>9</sup>. W pierwszym przypadku elementy te będą dostępne jedynie w bieżącym module, zaś w drugim – nie tylko w nim, lecz także we wszystkich częściach programu wykorzystujących ten moduł. Komentarze i puste sekcje można usunąć. Aby wymusić na Delphi wygenerowanie szablonu zadeklarowanej w ten sposób metody, można skorzystać z funkcji edytora zwanej **dokańczaniem klas** (ang. *Class Completion*), którą uaktywnia się poprzez naciśnięcie kombinacji klawiszy *Ctrl+Shift+C*.

Na końcu sekcji interfejsu modułu znajduje się deklaracja zmiennej globalnej o nazwie *Form1* reprezentującej obiekt formularza. Mówiąc dokładniej, zmienna ta zawiera odwołanie do obiektu formularza klasy *TForm1*, tj. wskaźnik do obszaru pamięci zajmowanego przez okno utworzone w czasie wykonywania aplikacji.

W sekcji implementacji modułu skojarzonego z formularzem standardowo występuje dyrektywa *{ \$R \*.dfm }* informująca kompilator, że ma on dołączyć plik *\*.dfm* o tej samej nazwie, co moduł. Podczas rozbudowywania formularza w sekcji

---

<sup>9</sup> Klasa może również zawierać sekcję chronioną (ang. *protected*) i opublikowaną (ang. *published*) [5, 6, 14, 25]. Elementy z sekcji chronionej są dostępne w module bieżącym i klasach potomnych, zaś elementy z sekcji opublikowanej są publiczne oraz widoczne w fazie projektowej w oknie inspektora obiektów.

implementacji umieszczany jest przez Delphi i programistę kod nowych metod obsługi zdarzeń i ewentualnie innych elementów programowych (typów, stałych, zmiennych, funkcji i procedur), niedostępny poza tym modulem.

Kontynuując powyższy przykład założmy, że intencją programisty jest zamknięcie okna uruchomionej aplikacji w momencie naciśnięcia zawartego w nim przycisku. Pełny kod metody *Button1Click* będzie wówczas wyglądał następująco:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Close;
end;
```

Jej pusty szablon automatycznie utworzy Delphi w trakcie definiowania procedury obsługi zdarzenia *OnClick*, a wywołanie metody *Close* dopisze programista. Parametr *Sender* standardowej klasy *TObject*, od której wywodzą się bezpośrednio lub pośrednio wszystkie klasy, wskazuje na obiekt, który spowodował zdarzenie. W tym przypadku jest nim przycisk *Button1* klasy *TButton*.

Przytoczony przykład pokazuje, jak niewiele kodu źródłowego trzeba napisać, by zbudować prostą aplikację okienkową. Oczywiście ta jest skrajnie prosta, jej działanie sprowadza się jedynie do pokazania okna głównego i zamknięcia go, gdy naciśnięty zostanie przycisk znajdujący się w jego obszarze roboczym. Niemniej jednak, gdyby ją tworzyć w tradycyjnym środowisku programowania, wymagałoby to nieporównywalnie większego wysiłku niż w Delphi.

Plik główny projektu aplikacji okienkowej zawiera kod w Object Pascalu, stanowiący jej moduł startowy. Kod ten całkowicie generuje Delphi i zazwyczaj nie ma potrzeby dokonywania w nim jakichkolwiek zmian. Aby umieścić go w oknie edytora, należy wybrać polecenie *View Source* w menu *Project* bądź w liście modułów, wyświetlonej w oknie dialogowym *View Unit* przywołanym poleceniem *Units* w menu *View*, wybrać nazwę projektu. Kod ten ma postać przypominającą prosty program napisany w języku Turbo Pascal:

```
program Project1;

uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

Kod rozpoczyna się nagłówkiem określającym nazwę projektu i zawiera listę wykorzystywanych w projekcie modułów, dyrektywę `{ $R *.res }` polecającą dołączenie pliku `*.res` o tej samej nazwie, co projekt, oraz trzy instrukcje wewnątrz pary słów kluczowych **begin** i **end** zakończonej kropką. Instrukcje te są wywołaniami metod standardowego obiektu *Application*, tworzonego automatycznie i reprezentującego aplikację podczas jej wykonania. Te trzy metody to: inicjalizacja obiektu aplikacji, utworzenie okna głównego aplikacji (obektu *Form1* reprezentującego formularz klasy *TForm1* zdefiniowanej w module *Unit1*) i wykonanie aplikacji (obsługa komunikatów dotyczących aplikacji) [6, 14].

Plik projektu aplikacji konsolowej również zawiera kod w Object Pascalu. Jest on generowany przez Delphi, gdy w składnicy obiektów przywołanej poleceniem *Other*, dostępnym po rozwinięciu pozycji *New* w menu *File*, podwójnie klikniemy ikonę *Console Application*. Kod ten jest bardzo prosty:

```
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Kod rozpoczyna się nagłówkiem z nazwą projektu, po którym następuje dyrektywa `{ $APPTYPE CONSOLE }` określająca typ aplikacji i klauzula **uses** informująca kompilator, że w projekcie mogą być wykorzystywane elementy programowe zdefiniowane w standardowym module *SysUtils*<sup>10</sup>. Zasadniczą treść programu, zawartą wewnątrz pary słów kluczowych **begin** i **end** zakończonej kropką, stanowi komentarz zaczynający się *TODO* (do zrobienia), który można usunąć.

Rozbudowa aplikacji konsolowej polega przede wszystkim na bezpośrednim wpisywaniu kodu w miejscu oznaczonym komentarzem. Można również dołączać moduły niezwiązane z formularzami, np. korzystając z polecenia *Unit* dostępnego po rozwinięciu pozycji *New* w menu *File*. Ponieważ aplikacja nie ma formularzy, do wprowadzania danych i wyprowadzania wyników trzeba korzystać z procedur *Read* i *ReadLn* oraz *Write* i *WriteLn*, podobnie jak w tradycyjnych programach

---

<sup>10</sup> Moduł *SysUtils* zawiera wiele funkcji systemowych, m.in. dotyczących formatowania tekstów: *IntToStr* (zamiana liczby całkowitej na łańcuch) i *StrToInt* (zamiana łańcucha na liczbę całkowitą), *FloatToStr* (zamiana liczby rzeczywistej na łańcuch) i *StrToFloat* (zamiana łańcucha na liczbę rzeczywistą), *DateToStr* (zamiana daty na łańcuch) i *StrToDate* (zamiana łańcucha na datę) oraz ogólną funkcję formatowania *Format* [5, 6].

pascalowych. Na końcu kodu należy dopisać bezparametrową instrukcję *ReadLn*, której zadaniem będzie wstrzymanie wykonania programu do chwili naciśnięcia klawisza *Enter*. Bez niej wypisane na ekranie wyniki migną użytkownikowi przed oczami, gdyż działanie programu zostanie zakończone.

Pliki \*.dfm zawierają informację o graficznej postaci formularzy i zamieszczonych na nich obiektach. Począwszy od Delphi 5 są one plikami tekstowymi, można je więc edytować, ale trzeba przy tym zachować składnię prostego języka skryptowego [6]. Najprostszym sposobem wyświetlenia skryptu formularza jest kliknięcie prawym przyciskiem myszy na formularzu i wybranie w menu kontekstowym polecenia *View as Text*. Poniższy wydruk przedstawia zawartość pliku *Unit1.dfm* definiującego formularz aplikacji okienkowej omówionej wyżej w tym podrozdziale (zob. też rys. D.5):

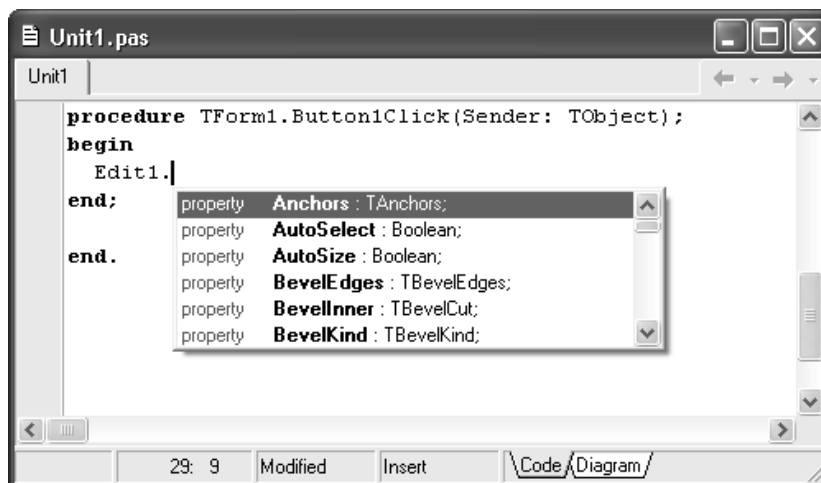
```
object Form1: TForm1
  Left = 192
  Top = 114
  Width = 307
  Height = 215
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 208
    Top = 144
    Width = 75
    Height = 25
    Caption = 'Zamknij'
    TabOrder = 0
    OnClick = Button1Click
  end
end
```

Zrozumienie skryptu nie powinno nastręczać żadnych trudności. Definiuje on obiekt typu *TForm1* o nazwie *Form1* (formularz) i należący do niego obiekt typu *TButton* o nazwie *Button1* (przycisk). Opisy obu obiektów mają postać listy właściwości z podanymi po znakach równości wartościami. Zauważmy, że zdarzenie *OnClick* jest właściwością, której przypisano wartość *Button1Click*, tj. nazwę obsługującej go metody. Nietrudno zmienić niektóre właściwości wpisując liczby, łańcuchy, wartości typu zbiorowego *set* lub inne wartości. Trzeba jednak uważać,

aby nie wpisać czegoś niepoprawnego, gdyż w przeciwnym przypadku Delphi wyświetli komunikat o błędzie. Aby powrócić do widoku formularza, wystarczy w menu kontekstowym przywołanym prawym przyciskiem myszy wybrać polecenie *View as Form* lub nacisnąć kombinację klawiszy *Alt+F12*.

Oprócz wspomnianej wyżej funkcji dokończania klas edytor ma wiele innych udogodnień. Delphi ciągle analizuje pisany przez użytkownika kod i stara się udzielać różnych odpowiedzi i wskazówek. Funkcja ta, nazywana ogólnie **wglądem do kodu** (ang. *Code Insight*), obejmuje szereg usług przydatnych zarówno początkującym, jak i zaawansowanym programistom.

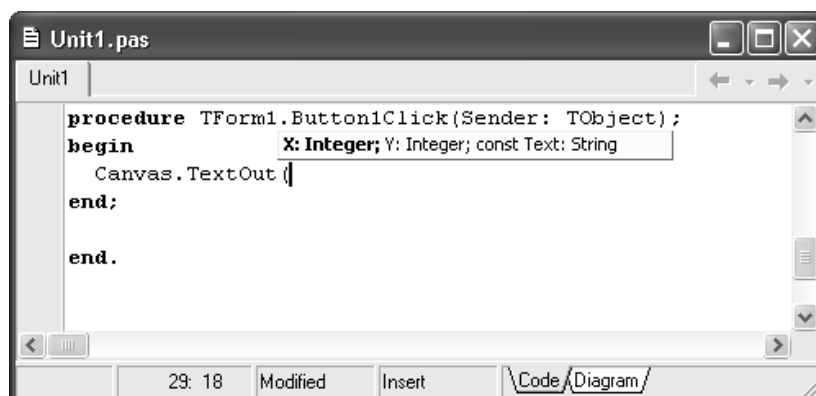
Szczególnie użyteczną usługą wspomagającą edycję kodu jest **dokończanie kodu** (ang. *Code Completion*). Pozwala ona na wybór właściwości lub metody obiektu z listy. Okienko listy pojawia się automatycznie, gdy po wpisaniu nazwy obiektu, np. *Edit1*, dopiszemy kropkę i chwilę poczekamy (por. rys. D.12). Gdy lista zostanie wyświetlona, można w niej wybrać pozycję, a następnie nacisnąć klawisz *Enter* lub kliknąć podwójnie myszą, a wybrany kod pojawi się w oknie edytora. Jeżeli rozpoczniemy pisanie kodu po kropce, lista będzie dostosowywać swoją zawartość do wpisywanego tekstu (filtrowanie). Aby wymusić pojawienie się okienka listy, należy nacisnąć kombinację klawiszy *Ctrl+spacja*, zaś aby go usunąć, należy nacisnąć klawisz *Esc*.



Rys. D.12. Funkcja dokończania kodu

Ważnymi usługami funkcji wglądu do kodu są również dołączanie **szablonów kodu** (ang. *Code Template*) i podpowiadanie **kodu parametrów** (ang. *Code Parameters*). Pierwsza umożliwia wyświetlenie listy predefiniowanych szablonów kodu, z której można, podobnie jak poprzednio, wybrać pozycję i wstawić do okna

edytora. Uruchamia się ją za pomocą kombinacji klawiszy *Ctrl+J*. Druga jest uruchamiana automatycznie, gdy wpisze nazwę funkcji lub procedury i utworzymy nawias. W okienku podpowiedzi, które się wówczas pojawi, wyświetlane są nazwy i typy parametrów z wytłuszczonym parametrem bieżącym (por. rys. D.13). Aby wymusić pojawienie się okienka podpowiedzi, należy nacisnąć kombinację klawiszy *Ctrl+Shift+spacja*.



Rys. D.13. Funkcja podpowiedzi kodu parametrów

Edytor zawiera więcej funkcji usprawniających pisanie kodu źródłowego, udostępnia też wiele innych skrótów klawiaturowych, a jego zachowanie można nawet konfigurować [6, 14].

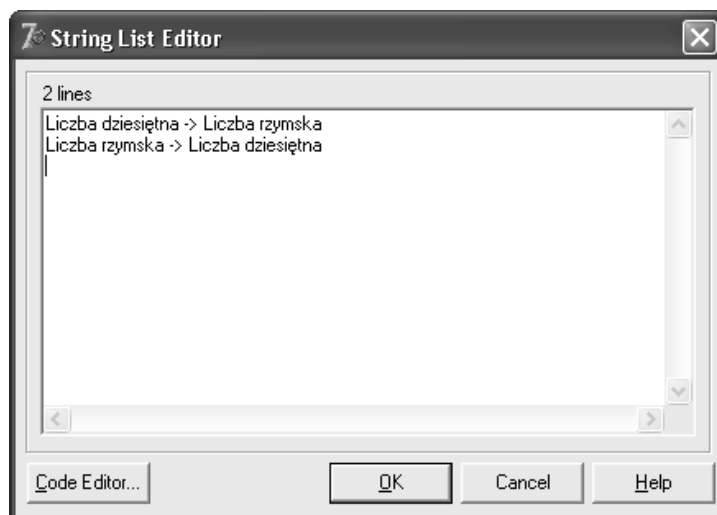
## D.7. Liczby rzymskie

Zbudujemy teraz przykładową aplikację okienkową od podstaw. Jej zadaniem będzie konwersja zapisu nieujemnych liczb całkowitych z systemu dziesiętnego na rzymski, i odwrotnie. Rozpocznijmy od zaprojektowania graficznego interfejsu użytkownika (GUI), wykorzystując standardowe komponenty biblioteczne Delphi. Potem zajmiemy się opracowaniem algorytmów konwersji i zaprogramowaniem funkcjonalności aplikacji. Ze względu na dużą liczbę komponentów, które oferuje Delphi, istnieje wiele możliwości przedstawienia graficznej postaci interfejsu. Aby zachować jego spójny i typowy dla Windows wygląd, wykorzystamy najbardziej podstawowe i najczęściej występujące w aplikacjach kontrolki.

Zaczynamy od otwarcia nowego projektu aplikacji okienkowej i określenia właściwości formularza. Domyślnie reprezentuje on okno, które można maksymalizować oraz swobodnie zmniejszać i powiększać. W tym przypadku możliwość regulacji rozmiaru okna nie ma większego sensu, toteż posługując się inspektorem

obiektów ustawiamy we właściwości zagnieżdżonej *BorderIcons* formularza podwłaściwość *biMaximize* na *false* (okno nie może być maksymalizowane), a właściwość *BorderStyle* na *bsSingle* (okno z ramką, bez możliwości zmiany rozmiarów). Ponadto ustawiamy właściwość *Caption* (napis na pasku tytułowym okna) na *Liczby rzymskie*, a właściwość *Position* na *poDesktopCenter* (po uruchomieniu okno zajmie centralną pozycję na ekranie). Zmniejszamy też rozmiary formularza do około 347×195 pikseli.

Podstawowym komponentem pozwalającym na wprowadzanie i wyświetlanie pojedynczej linii tekstu jest *Edit* (pole edycyjne). Do formularza wstawiamy dwa takie komponenty, Delphi nazwie je *Edit1* i *Edit2* (właściwości *Name*). Pierwszy posłuży do wygodnego wprowadzania z klawiatury łańcucha reprezentującego liczbę dziesiętną lub rzymską, a drugi do wyświetlania wyniku obliczeń. Właściwości *Text* (tekst wyświetlany w polu edycji) obu komponentów czyścimy. Aby zablokować możliwość edycji tekstu wynikowego za pomocą klawiatury, właściwość *ReadOnly* (tylko do odczytu) komponentu *Edit2* ustawiamy na *true*. Możemy też wizualnie wskazać na jego odmienny charakter, zmieniając jego właściwość *Color* np. na *clInfoBk* (kolor domyślnie jasnokremowy). Następnie opisujemy oba pola edycyjne, umieszczając przed nimi dwa komponenty *Label* (etykieta). Ich właściwości *Caption* ustawiamy na *Liczba dziesiętna* i *Liczba rzymska*.



Rys. D.14. Edytor listy łańcuchów

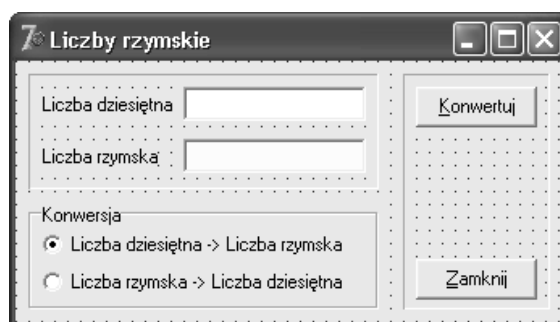
Aby umożliwić użytkownikowi swobodny wybór rodzaju konwersji, skorzystamy z komponentu *RadioGroup* (grupa przycisków opcji). Po umieszczeniu tego komponentu na formularzu ustawiamy jego właściwość *Caption* na *Konwersja*,



a po odnalezieniu właściwości *Items* klikamy dwukrotnie na polu obok jej nazwy albo raz na przycisku oznaczonym wielokropkiem. Następnie w oknie edytora listy łańcuchów (ang. *String List Editor*), które się wówczas pojawi, wpisujemy dwa łańcuchy: *Liczba dziesiętna* → *Liczba rzymska* i *Liczba rzymska* → *Liczba dziesiętna* (rys. D.14). Zapis potwierdzamy naciskając *OK*. Aby pierwszy przycisk opcji został zaznaczony, nadajemy właściwości *ItemIndex* wartość 0 (zero). Gdy użytkownik kliknie drugi przycisk opcji, zostanie on włączony, a pierwszy wyłączony. Jednocześnie właściwości *ItemIndex* zostanie przypisana wartość 1. Numerowanie elementów listy od indeksu 0 jest nagminnie stosowaną zasadą w Delphi. W ogólnym przypadku wartością *ItemIndex* jest numer wybranego elementu listy pomniejszony o 1, a gdy żaden element nie jest wybrany, wartość -1.

Na koniec umieszczamy na formularzu dwa przyciski *Button*, które Delphi nazwie *Button1* i *Button2*. Ich właściwości *Caption* zmieniamy, wpisując etykiety *&Konwertuj* i *&Zamknij*. Znak ampersand (&) oznacza, że występująca za nim litera zostanie podkreślona, co w czasie wykonywania aplikacji umożliwia wykorzystanie skrótu klawiaturowego. I tak, gdy użytkownik naciśnie *Alt+K*, wykonana zostanie konwersja liczby z systemu dziesiętnego na rzymski lub odwrotna, a gdy naciśnie *Alt+Z*, aplikacja zostanie zakończona.

Zaprojektowany formularz jest przedstawiony na rysunku D.15. W celu uatrakcyjnienia interfejsu użytkownika na formularzu umieszczone zostały jeszcze dwa komponenty *Bevel* z karty *Additional* palety komponentów. Dzięki nim uzyskano efekt trójwymiarowości (wgłębienie wokół pól edycyjnych i przycisków). Komponent *Bevel* nie ma żadnych zdarzeń, pełni jedynie rolę dekoracyjną.



Rys. D.15. Projekt okna aplikacji

Zaprogramowanie funkcjonalności aplikacji zaczniemy od zdefiniowania procedury obsługi zdarzenia *OnClick* kontrolki *RadioGroup1*, które nastąpi w momencie, gdy użytkownik przestawi kontrolkę, wybierając inny rodzaj konwersji. Ponieważ pole *Edit1* ma służyć do wprowadzania zarówno zapisu dziesiętnego, jak i rzymskiego liczby, a pole *Edit2* do wyświetlania jej zapisu alternatywnego, zmia-

na stanu kontrolki *RadioGroup1* powinna pociągać za sobą zmianę opisujących te pola etykiet *Label1* i *Label2*. Zważywszy na wygodę obsługi okna, naturalnym wydaje się też wyczyszczenie obu pól edycyjnych i ustawienie pierwszego jako przyjmującego zdarzenia generowane przez klawiaturę:

```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  if RadioGroup1.ItemIndex = 0 then
  begin
    Label1.Caption := 'Liczba dziesiętna';
    Label2.Caption := 'Liczba rzymska';
  end else
  begin
    Label1.Caption := 'Liczba rzymska';
    Label2.Caption := 'Liczba dziesiętna';
  end;
  Edit1.Text := '';
  Edit2.Text := '';
  Edit1.SetFocus;
end;
```

Wypada wreszcie zapisać niedokończony jeszcze moduł i projekt na dysku (pamiętamy o utworzeniu osobnego katalogu) i sprawdzić zachowanie się aplikacji. Przykładowo, moduł zapisujemy w pliku o nazwie *MainUnit.pas*, a projekt w pliku *Lrzym.dpr*. Aby skompilować projekt, wybieramy polecenie *Compile Lrzym* w menu *Project* lub naciskamy kombinację klawiszy *Ctrl+F9*. Aby uruchomić aplikację, wybieramy polecenie *Run* w menu o tej samej nazwie lub naciskamy *F9*. Możemy też od razu wymusić uruchomienie aplikacji, a wtedy Delphi ją najpierw skompiluje, jeśli jest to konieczne, a potem uruchomi. Aplikację zamykamy, naciskając *Alt+F4* lub klikając przycisk z krzyżykiem na pasku tytułowym okna.

Aplikacja zachowuje się przyzwoicie, o ile pominiemy fakt, iż nie reaguje do końca na kliknięcie przycisków *Konwertuj* i *Zamknij*. Definiujemy zatem procedury obsługi zdarzeń *OnClick* komponentów *Button1* i *Button2*. Kod pierwszej wymaga większych przemyśleń, toteż na razie zastępujemy go pustym komentarzem (same znaki *//*), żeby Delphi nie usunęło jej podczas kompilacji lub zapisu modułu na dysku. Kod drugiej sprowadza się do wywołania metody *Close* formularza:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

Znaleźliśmy się w sytuacji, w której musimy zająć się algorytmami konwersji notacji dziesiętnej liczby na rzymską i notacji rzymskiej na dziesiętną. Mówiąc ściślej, chodzi o przekształcenie liczby całkowitej (wartości typu *integer*) na łańcuch (wartość typu **string**) reprezentujący tę liczbę w notacji rzymskiej i przekształcenie takiego łańcucha na liczbę całkowitą. Cały kod można by zamieścić w metodzie *Button1Click*, ale stałaby się ona przepastnie rozbudowana i zagramatwana. Zdecydowanie lepszym rozwiązaniem jest zbudowanie dwóch funkcji konwersji i umieszczenie ich w odrębnym module. Program stanie się przez to bardziej czytelny i przejrzysty, a stworzony moduł jednostką do wielokrotnego wykorzystania w innych programach. Podział kodu na niezależne moduły stanowi podstawową strategię prowadzącą do unikania spletanego kodu (ang. *spaghetti code*).

Wzorując się na nazwach *IntToStr* i *StrToInt* funkcji z modułu *SysUtils*, dotyczących dwukierunkowej konwersji liczby całkowitej na łańcuch i łańcucha na liczbę całkowitą, nadamy naszym funkcjom równie sugestywne nazwy *IntToRom* i *RomToInt*, a modułowi nazwę *RomUtils*. Wybierając polecenie *Unit*, dostępne po rozwinięciu pozycji *New* w menu *File*, lub dwukrotnie klikając ikonę *Unit* w oknie składnicy obiektów, dołączamy do aplikacji szablon nowego modułu:

```
unit Unit2;  
  
interface  
  
implementation  
  
end.
```

Następnie w sekcji interfejsu wpisujemy deklaracje dwóch funkcji konwersji:

```
function IntToRom(n: integer): string;  
function RomToInt(t: string): integer;
```

a w sekcji implementacji ich wstępne definicje:

```
function IntToRom(n: integer): string;  
begin  
    Result := '';  
end;  
  
function RomToInt(t: string): integer;  
begin  
    Result := 0;  
end;
```

Niezależnie od wartości parametru wynikiem wywołania funkcji *IntToRom* jest na razie łańcuch pusty, a funkcji *RomToInt* wartość całkowita 0. W standardowym

Pascalu wynik końcowy należało przypisać nazwie funkcji. W Object Pascalu można także użyć predefiniowanej zmiennej *Result* [25, 26] (właśnie skorzystano z tej możliwości). Nowy moduł, chociaż jeszcze niedokończony, warto zapisać na dysku, nadając plikowi nazwę *RomUtils.pas*.

Zanim zajmiemy się rozbudową modułu *RomUtils*, wyjaśnimy pokrótce, na czym polega system rzymski zapisu liczb. Otóż używa się w nim siedmiu podstawowych symboli jednoliterowych i sześciu dwuliterowych, którym przypisane są wybrane wartości całkowite, zwane **wagami**:

I	1	IV	4
V	5	IX	9
X	10	XL	40
L	50	XC	90
C	100	CD	400
D	500	CM	900
M	1000		

Ciąg takich symboli reprezentuje liczbę równą sumie odpowiadających im wag. Na przykład MCDXLIV oznacza liczbę 1444 ( $1000 + 400 + 40 + 4$ ). Ogólna reguła zapisu liczb rzymskich orzeka, że symbol określający wagę mniejszą powinien być umieszczony za symbolem określającym wagę większą<sup>11</sup>. Obowiązuje również zasada, że symbol jednoliterowy może wystąpić w zapisie co najwyżej trzy razy, zaś symbol dwuliterowy nie więcej niż jeden raz. Ponadto zapis powinien składać się z możliwie jak najmniejszej liczby znaków. Na przykład liczba 9 powinna być zapisana jako IX, nie zaś jako VIII ( $5 + 1 + 1 + 1 + 1$ ) czy VIV ( $5 + 4$ ).

Powyższe symbole i ich wagi mogą być reprezentowane przez dwie tablice o indeksach przebiegających typ wyliczeniowy, zdefiniowany jako lista nazw wszystkich symboli uporządkowanych od największej do najmniejszej wagi:

```
type
  Symbol = (M, CM, D, CD, C, XC, L, XL, X, IX, V, IV, I);

const
  s: array[Symbol] of string = (
    'M', 'CM', 'D', 'CD', 'C', 'XC', 'L', 'XL', 'X', 'IX',
    'V', 'IV', 'I');

  w: array[Symbol] of integer = (
    1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1);
```

<sup>11</sup> Symbole dwuliterowe są wyjątkiem od tej reguły, gdyż są złożone z dwóch symboli jednoliterowych ustawionych w odwrotnej kolejności. Takie przestawienie oznacza, że od wagi większej należy odjąć wagę mniejszą, stąd np. zapis XL określa liczbę 40 ( $50 - 10$ ).

Typ *Symbol* został użyty w celu poprawienia czytelności programu; równie dobrze można było wykorzystać typ okrojony 0..12 (dokładnie taki jest odpowiednik numeryczny typu *Symbol*), ale kod nie byłby tak obrazowy. Tablice *s* i *w* są zainicjalizowane: elementy pierwszej są łańcuchami umożliwiającymi składanie zapisu rzymskiego liczby, zaś elementy drugiej są wagami odpowiadającymi tym łańcuchom. Łańcuchy, z których składa się tablica *s*, są alokowane dynamicznie i zajmują dokładnie tyle bajtów pamięci, ile potrzeba [5, 25]<sup>12</sup>.

Po umieszczeniu powyższego kodu na początku sekcji implementacji modułu możemy rozwinąć definicje funkcji *IntToRom* i *RomToInt*. W przypadku pierwszej sprawdzamy, czy kolejny symbol, poczynając od M a kończąc na I, ma wystąpić w zapisie liczby *n*, tj. czy waga tego symbolu jest mniejsza od wartości *n* lub jest jej równa. Dopóki ten warunek jest spełniony, symbol uwzględniamy w łańcuchu wynikowym, a jego wagę odejmujemy od *n*. Gdy warunek nie jest spełniony, przechodzimy do symbolu o mniejszej wadze i wykonujemy dla niego tę samą pętlę sprawdzającą. Algorytm ten możemy zaprogramować następująco:

```
function IntToRom(n: integer): string;
var
  r: Symbol;
begin
  Result := '';
  for r:=M to I do
    while w[r]<=n do
      begin
        Result := Result + s[r];
        Dec(n, w[r]);
      end;
  end;
```

Niemal identyczny algorytm można zastosować w przypadku drugiej funkcji. Podobnie jak poprzednio, wystarczy przebiegać po wszystkich symbolach i badać, czy występują one w zapisie liczby. Różnica polega na tym, iż w każdym kroku pętli wewnętrznej sprawdzamy, czy łańcuch *s[r]* reprezentujący kolejny symbol *r* rozpoczyna łańcuch *t* stanowiący zapis rzymski liczby, nie zaś, czy waga *w[r]* nie przekracza wartości *n*. Dopóki warunek jest spełniony, wagę dodajemy do wartości wynikowej funkcji, a z łańcucha *t* usuwamy jego początkowe znaki odpowiadające łańcuchowi *s[r]*. Do zaprogramowania operacji na łańcuchach wykorzystamy standardowe funkcje *Length* (długość łańcucha) i *Copy* (kopia fragmentu łańcucha) oraz procedurę *Delete* (usunięcie ciągu znaków z łańcucha):

---

<sup>12</sup> W Turbo Pascalu byłyby statyczne, zajmowałyby po 256 bajtów (ciąg 255 znaków poprzedzony bajtem informującym o jego faktycznej długości). Dla zaoszczędzenia pamięci należałoby je zadeklarować jako **string[2]**, a wtedy zajmowałyby po 3 bajty.

```

function RomToInt(t: string): integer;
var
  r: Symbol;
begin
  Result := 0;
  for r:=M to I do
    while s[r]=Copy(t, 1, Length(s[r])) do
      begin
        Inc(Result, w[r]);
        Delete(t, 1, Length(s[r]));
      end;
  end;

```

Po zapisaniu modułu *RomUtils* na dysku przechodzimy do jego wykorzystania w module *MainUnit*. Najpierw rozszerzamy listę **uses** nazw wykorzystywanych w nim modułów o nazwę *RomUtils*, a potem rozbudowujemy metodę *Button1Click* po uprzednim usunięciu z niej pustego komentarza (//):

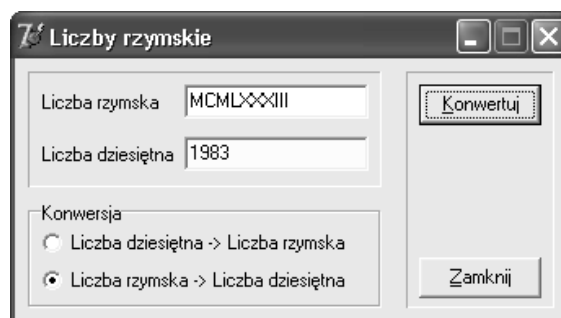
```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if RadioGroup1.ItemIndex = 0
    then Edit2.Text := IntToRom(StrToInt(Edit1.Text))
    else Edit2.Text := IntToStr(RomToInt(Edit1.Text));
end;

```

W przypadku konwersji liczby z notacji dziesiętnej na rzymską, łańcuch cyfr wprowadzony w polu *Edit1* jest najpierw zamieniany na liczbę całkowitą, a ta jest następnie przekształcana na łańcuch reprezentujący zapis rzymski. W przypadku konwersji odwrotnej łańcuch reprezentujący liczbę rzymską jest zamieniany na liczbę całkowitą, a ta na łańcuch cyfr. Wynik jest umieszczany w polu *Edit2*.

Można wreszcie, po zapisaniu na dysku, uruchomić aplikację i sprawdzić jej działanie. Rysunek D.16 przedstawia przykładowy wynik jej wykonania. Proste testy pokazują jednak, że nie jest ona wolna od wad.



Rys. D.16. Wynik wykonania aplikacji

Aby użytkownik nie musiał przełączać klawiatury na wielkie litery podczas wprowadzania liczby rzymskiej, ustawiamy właściwość *CharCase* komponentu *Edit1* na *ecUpperCase*, a wtedy wszystkie małe litery wpisywane z klawiatury będą automatycznie zamieniane na wielkie<sup>13</sup>. Nie rozwiązuje to jednak wszystkich problemów wynikających z wprowadzenia tekstu, który ma być zinterpretowany jako liczba rzymska, a żadnego, gdy ma on być zapisem nieujemnej liczby całkowitej. Błędny tekst spowoduje albo pojawienie się okna z komunikatem w języku angielskim o błędzie konwersji, albo wyświetlenie błędnego wyniku w polu *Edit2* okna aplikacji. O ile wyświetlenie informacji obcojęzycznej o błędzie jest rozwiązaniem nienajlepszym, ale do zaakceptowania, to niewątpliwie brak jakiejkolwiek informacji o błędzie i podanie niewłaściwego wyniku jest niedopuszczalne.

Użytkownikowi nie powinno się narzucać, co wolno mu wpisywać, a czego nie. Generalnie należy przyjąć zasadę, że może wpisać wszystko, na co pozwala mu aplikacja. To aplikacja powinna wychwytywać nienormalne sytuacje i właściwie na nie reagować, nie obarczając tym użytkownika<sup>14</sup>.

Delphi zawiera wygodny mechanizm **obsługi wyjątków** (ang. *exception*) umożliwiający wychwytywanie błędów i określanie sposobu postępowania w przypadku ich wystąpienia. Jego idea polega na umieszczeniu kodu, który może spowodować błąd, w specjalnym bloku chronionym, a kodu odpowiedzialnego za obsługę tego błędu w oddzielnym bloku obsługi wyjątków. Dzięki wyjątkom można oddzielić obsługę błędów od właściwej części programu, przez co staje się on bardziej uporządkowany, a kontrola błędów bardziej ujednolicona.

Jedna z dwóch możliwych konstrukcji służących do przechwytywania wyjątków ma postać następującą:

```
try
    // Kod, który może spowodować wyjątek
except
    // Kod obsługi wyjątków
end
```

Kod realizujący określone zadanie jest umieszczony w części **try**, a kod obsługujący wyjątki w części **except**. Jeżeli podczas realizacji zadania wygenerowany zostanie wyjątek, nastąpi natychmiastowe przejście do wykonywania instrukcji w części **except** bez wykonywania dalszych instrukcji w części **try**.

---

<sup>13</sup> Niekiedy można spotkać zapis liczb rzymskich złożony z małych liter. Uwzględnienie takiej postaci wymaga modyfikacji programu, np. wykorzystania komponentu *Radiogroup*, który umożliwiałby wybór wielkich lub małych liter.

<sup>14</sup> Wiele cennych wskazówek i przykładów (języki C, C++ i Java) dotyczących dobrego programowania można znaleźć w książce [13].

Druga konstrukcja służąca do przechwytywania wyjątków różni się formalnie od pierwszej tym, że zamiast słowa kluczowego **except** występuje w niej słowo kluczowe **finally**:

```
try
    // Kod, który może spowodować wyjątek
finally
    // Kod wykonywany zawsze
end
```

Kod zamieszczony w części **finally** jest wykonywany zawsze, niezależnie od tego, czy uruchomiony zostanie wyjątek czy nie. Blok ten jest najczęściej używany w przypadku, gdy wymagane są czynności porządkowe, np. zamknięcie pliku lub zwolnienie pamięci przydzielonej w tej części programu [5, 6, 14].

Blok obsługi wyjątków pierwszej postaci zastosujemy teraz w naszej aplikacji konwersji notacji dziesiętnej liczb na rzymską i rzymskiej na dziesiętną, a ściślej, w metodzie *Button1Click* w module *MainUnit*. Oto pełna wersja tego modułu:

```
unit MainUnit;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, ExtCtrls, StdCtrls, RomUtils;

type
    TForm1 = class(TForm)
        Edit1: TEdit;
        Edit2: TEdit;
        Label1: TLabel;
        Label2: TLabel;
        RadioGroup1: TRadioGroup;
        Button1: TButton;
        Button2: TButton;
        Bevel1: TBevel;
        Bevel2: TBevel;
        procedure RadioGroup1Click(Sender: TObject);
        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}
```



```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  if RadioGroup1.ItemIndex = 0 then
  begin
    Label1.Caption := 'Liczba dziesiętna';
    Label2.Caption := 'Liczba rzymska';
  end else
  begin
    Label1.Caption := 'Liczba rzymska';
    Label2.Caption := 'Liczba dziesiętna';
  end;
  Edit1.Text := '';
  Edit2.Text := '';
  Edit1.SetFocus;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  try
    if RadioGroup1.ItemIndex = 0
    then Edit2.Text := IntToRom(StrToInt(Edit1.Text))
    else Edit2.Text := IntToStr(RomToInt(Edit1.Text));
  except
    Application.MessageBox('Nieprawidłowo wypełnione pole',
      'Błąd', mb_IconError or mb_OK);
    Edit1.SetFocus;
  end;
end;

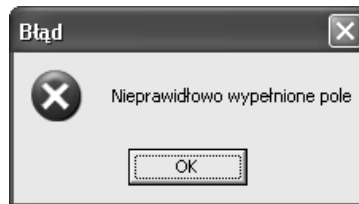
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

Jeśli w polu edycji *Edit1* znajdzie się nieprawidłowy tekst, to jego konwersja spowoduje wyświetlenie okna komunikatu pokazanego na rysunku D.17<sup>15</sup>. Tworzy go metoda *MessageBox* obiektu *Application*, która pozwala na określenie zarówno komunikatu wewnątrz okna, jak i nazwy na pasku tytułowym, a ponadto umożliwia dołączenie ikony i różnych przycisków. Po zamknięciu okna komunikatu pole *Edit1* zostaje ustawione na przyjmowanie zdarzeń generowanych przez klawiaturę, możemy więc od razu korygować zawarty w nim tekst.

---

<sup>15</sup> Jeżeli wyjątki są obsługiwane przez aplikację, można wyłączyć ich obsługę przez debugger, ponieważ jego komunikaty przeszkadzają w testowaniu aplikacji. W tym celu należy w oknie *Debugger Options*, przywołanym poleceniem o tej samej nazwie w menu *Tools*, wybrać zakładkę *Language Exceptions* i wyłączyć opcję *Stop on Delphi Exceptions*.



Rys. D.17. Okno komunikatu o błędzie

Nie jest to jednak koniec udoskonalania naszej aplikacji, ponieważ niektóre błędy w tekście wprowadzonym przez użytkownika nie są przez nią wykrywane. Przyczyna tkwi w module *RomUtils*. Po pierwsze, funkcja *IntToRom* przekształca liczby ujemne na łańcuch pusty, co jest oczywistym błędem. Co więcej, taka konwersja nie ma najmniejszego sensu, ponieważ Rzymianie nie znali liczb ujemnych (nie znali też zera, ale wynik w postaci łańcucha pustego jest do przyjęcia). Po drugie, funkcja *RomToInt* przekształca na liczbę całkowitą nawet te łańcuchy, które nie reprezentują prawidłowego zapisu rzymskiego. Sytuacja taka ma miejsce wówczas, gdy po zakończeniu pętli zewnętrznej łańcuch nie został zniwelowany do łańcucha pustego (wynik jest wtedy odpowiednikiem numerycznym przetworzonej części łańcucha). W obu przypadkach aplikacja powinna zgłosić wyjątek.

Wszystkie wyjątki są obiektami ogólnej klasy *Exception* lub klas potomnych, wywodzących się od niej bezpośrednio lub pośrednio. Aby zgłosić wyjątek, trzeba posłużyć się instrukcją **raise**, w której należy utworzyć obiekt wyjątku za pomocą konstruktora danej klasy. Na przykład w funkcji *IntToRom* można nie dopuścić do konwersji liczb ujemnych na rzymskie, wykorzystując instrukcję:

```
if n<0 then
    raise Exception.Create('Liczba rzymska nie może być ujemna');
```

Tekst będący parametrem wywołania konstruktora *Create* klasy *Exception* zostanie wyświetlony w oknie komunikatu utworzonym przez domyślną procedurę obsługi wyjątków, gdy programista nie zamieści bloku **try ... except** przechwytyującego ten wyjątek. Może go też wykorzystać debugger.

Klasa *Exception* i inne klasy wyjątków są zawarte w module *SysUtils*, toteż w module *RomUtils* należy umieścić listę **uses** z nazwą *SysUtils*. Ostateczna wersja modułu *RomUtils* może mieć postać następującą:

```
unit RomUtils;

interface

function IntToRom(n: integer): string;
function RomToInt(t: string): integer;
```

```
implementation

uses
  SysUtils;

type
  Symbol = (M, CM, D, CD, C, XC, L, XL, X, IX, V, IV, I);

const
  s: array[Symbol] of string = (
    'M', 'CM', 'D', 'CD', 'C', 'XC', 'L', 'XL', 'X', 'IX',
    'V', 'IV', 'I');

  w: array[Symbol] of integer = (
    1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1);

function IntToRom(n: integer): string;
var
  r: Symbol;
begin
  Result := '';
  if n < 0 then
    raise Exception.Create('Liczba rzymska nie może być ujemna');
  for r:=M to I do
    while w[r] <= n do
      begin
        Result := Result + s[r];
        Dec(n, w[r]);
      end;
  end;

function RomToInt(t: string): integer;
var
  r: Symbol;
begin
  Result := 0;
  for r:=M to I do
    while s[r]=Copy(t, 1, Length(s[r])) do
      begin
        Inc(Result, w[r]);
        Delete(t, 1, Length(s[r]));
      end;
  if t <> '' then
    raise Exception.Create('Błąd w zapisie liczby rzymskiej');
end;

end.
```

Być może bardziej eleganckie byłoby wykorzystanie w funkcjach *IntToRom* i *RomToInt* wyjątku klasy *EConvertError* oznaczającego błąd konwersji. Wyjątki tego typu są często zgłaszane podczas wczytywania danych z klawiatury, m.in.

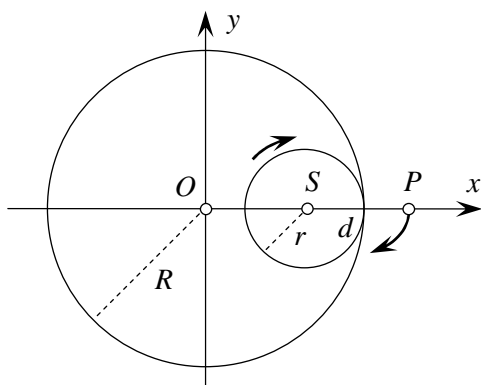
wtedy, gdy przekształcany na liczbę łańcuch wejściowy nie reprezentuje wartości numerycznej. Wymagana modyfikacja kodu obu funkcji jest nieznaczna, wystarczy jedynie zamiast nazwy *Exception* użyć nazwy *EConvertError*.

W Delphi istnieje wiele typów wyjątków, można też definiować nowe. Typ wyjątku jest istotny, gdy decydujemy się obsługiwać tylko niektóre rodzaje wyjątków, a pozostałe zgłaszać ponownie, aby zostały przechwycone w innym miejscu programu lub obsługane przez procedury standardowe [5, 6, 14, 25].

## D.8. Kreślenie hipocykloidy

Delphi zawiera szereg komponentów obsługujących grafikę. Komponent *Shape* (figura) pozwala w fazie projektowej aplikacji na rysowanie kół, elips, kwadratów i prostokątów oraz manipulowanie nimi w fazie wykonywania. Komponent *Image* (obraz) służy do wyświetlania map bitowych, pozwalając na ładowanie ich z pliku zarówno w fazie projektowej, jak i w fazie wykonywania, a ponadto udostępnia właściwość *Canvas* (płótno), która umożliwia rysowanie punktów, linii, wykresów, krzywych i innych skomplikowanych figur geometrycznych. Właściwość tę ma formularz i kilka innych komponentów, m.in. *PaintBox* (obszar malowania).

Podstawowe narzędzia, które oferuje obiekt *Canvas*, wykorzystamy teraz do narysowania krzywej, którą opisuje koniec  $P$  ramienia  $SP$  przytwierdzonego sztywno wzdłuż promienia okręgu toczącego się stycznie wewnątrz większego okręgu (rys. D.18). Krzywą tę nazywamy *hipocykloidą*.



Rys. D.18. Powstawanie hipocykloidy

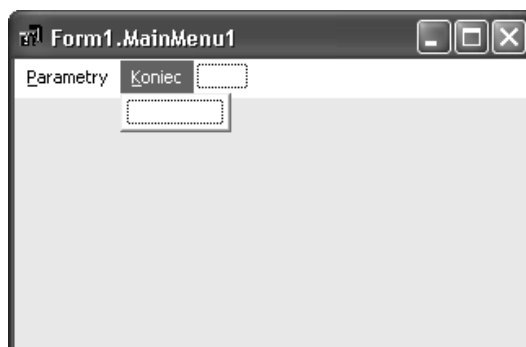
Jeżeli na początku ruchu ramię znajdowało się na osi  $x$ , hipocykloidę można wyrazić za pomocą równań parametrycznych:

$$\begin{aligned} x &= (R - r) \cos \varphi + d \cdot \cos \frac{R - r}{r} \varphi \\ y &= (R - r) \sin \varphi - d \cdot \sin \frac{R - r}{r} \varphi \end{aligned} \quad (0 \leq \varphi \leq 2\pi) \quad (\text{D.1})$$

gdzie  $R$  jest promieniem większego okręgu,  $r$  – mniejszego, a  $d$  – długością ramienia. Jeżeli  $R$  jest podzielne przez  $r$ , krzywa jest zamknięta, a jeżeli nie, zamyka się po pewnej liczbie obiegów mniejszego okręgu, gdy  $R/r$  jest liczbą wymierną, a nie zamyka się nigdy, gdy jest liczbą niewymierną.

Po otwarciu nowego projektu aplikacji zmieniamy właściwość *Caption* formularza na *Hipocykloida*, a właściwość *Color* na *clWhite* (białe tło obszaru roboczego okna). Następnie umieszczamy na formularzu niewidzialny komponent *MainMenu*, który służy do projektowania menu głównego. Aby otworzyć okno edytora menu (ang. *Menu Designer*), klikamy dwukrotnie na komponencie *MainMenu1*, bądź po odnalezieniu właściwości *Items* tego komponentu klikamy dwukrotnie na polu obok jej nazwy albo raz na przycisku oznaczonym wielokropkiem.

Aby utworzyć nowy element menu, w oknie edytora menu wskazujemy jego pozycję, a w oknie inspektora obiektów wpisujemy etykietę (właściwość *Caption*) i naciskamy *Enter*. Możemy przy tym użyć znaku ampersand (&), który określa klawisz skrótu. Delphi nazywa elementy menu (właściwość *Name*), dopisując do ich etykiet numer kolejny i usuwając wszystkie znaki, które nie są literami alfabetu angielskiego ani cyframi<sup>16</sup>, np. pierwszy element o etykiecie *&Otwórz* nazwie *Otwrz1*, drugi – *Otwrz2* itd. Dla potrzeb naszej aplikacji definiujemy dwa elementy menu, nadając im etykiety *&Parametry* i *&Koniec* (rys. D.19). Pierwszy posłuży do zmiany parametrów hipocykloidy, drugi do zakończenia aplikacji.

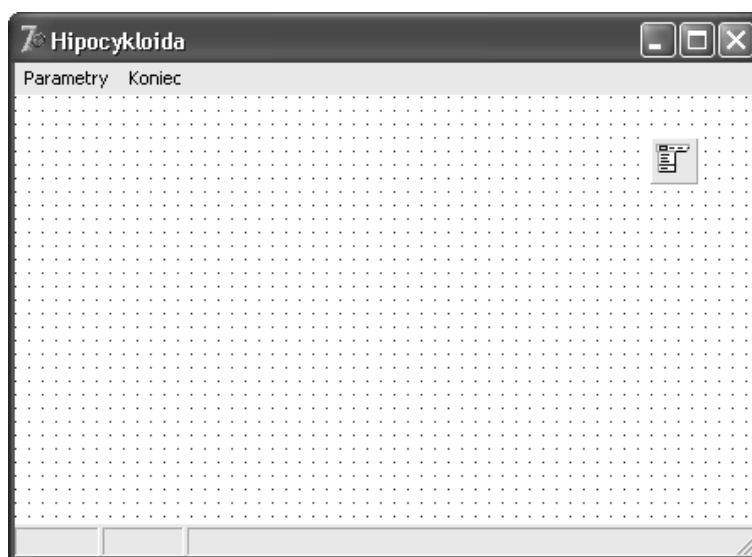


Rys. D.19. Okno edytora menu głównego

<sup>16</sup> Wpisanie łącznika zamiast etykiety powoduje utworzenie separatora elementów menu (poziomej linii). Delphi nazywa separator, dopisując numer kolejny do litery *N*.

Drugim komponentem, który po odnalezieniu na karcie *Win32* palety komponentów wstawiamy do formularza, jest *StatusBar* (pasek statusu). Będzie on służył do wyświetlania parametrów  $R$ ,  $r$  i  $d$  hipocykloidy. Aby dla każdego parametru zdefiniować na pasku statusu odrębny panel, dwukrotnie klikamy na komponencie *StatusBar1*, albo w menu kontekstowym, przywołanym dla tego komponentu prawym przyciskiem myszy, wybieramy polecenie *Panels Editor*. W okienku edytora, które się wówczas pojawi, trzykrotnie tworzymy obiekt klasy *TStatusPanel*, wykorzystując albo przycisk *Add New*, albo klawisz *Insert*, albo polecenie *Add* w menu kontekstowym. Na koniec niepotrzebne już okienko możemy zamknąć.

Zaprojektowany formularz jest pokazany na rysunku D.20. Ponieważ nie zmienialiśmy jego właściwości *BorderIcons* i *BorderStyle*, reprezentowane przez niego okno główne aplikacji będzie można maksymalizować, będzie też można zmieniać jego rozmiary za pomocą myszy lub klawiatury.



**Rys. D.20.** Projekt okna głównego aplikacji

Możemy teraz przystąpić do rozbudowy wygenerowanego przez Delphi kodu źródłowego modułu związanego z formularzem klasy *TForm1*. Zaczynamy od zadeklarowania pięciu pól w części prywatnej klasy<sup>17</sup>:  $xs$ ,  $ys$ ,  $Rd$ ,  $Rm$  i  $d$ . W polach  $xs$  i  $ys$  przechowywane będą współrzędne środka obszaru roboczego okna, w trzech pozostałych parametry hipocykloidy. Ponieważ Pascal nie rozróżnia w nazwach

<sup>17</sup> Zgodnie z ogólną zasadą programowania obiektowego, pola klasy powinny być w niej prywatne, a metody publiczne. Oczywiście, od tej zasady istnieją pewne wyjątki.

wielkich i małych liter, zamiast  $R$  i  $r$  użyliśmy nazw  $Rd$  i  $Rm$ . Wszystkie pola są typu całkowitego, ponieważ współrzędne punktów obszaru roboczego są wyrażane jako liczba pikseli od jego lewej i górnej krawędzi.

Następnie w części publicznej klasy umieszczamy deklaracje dwóch metod, które nazywamy *CalculateCenter* i *UpdateStatusBar*. Obie są procedurami. Pierwsza ma wyliczać wartości  $xs$  i  $ys$ , gdy okno zostanie utworzone lub zmieni rozmiar, druga ma aktualizować informacje wyświetlane na pasku statusu, gdy zmienia się parametry krzywej. Aby Delphi wygenerowało w części **implementation** modułu puste szablony obu metod, naciskamy *Ctrl+Shift+C* (funkcja dokończania klas). Możemy je teraz uzupełnić do postaci:

```
procedure TForm1.CalculateCenter;
begin
  xs := ClientWidth div 2;
  ys := (ClientHeight - StatusBar1.Height) div 2;
end;

procedure TForm1.UpdateStatusBar;
begin
  StatusBar1.Panels[0].Text := ' R = ' + IntToStr(Rd);
  StatusBar1.Panels[1].Text := ' r = ' + IntToStr(Rm);
  StatusBar1.Panels[2].Text := ' d = ' + IntToStr(d);
end;
```

W obliczeniach współrzędnej  $ys$  środka wolnej części obszaru roboczego okna uwzględniona jest wysokość paska statusu umieszczonego na dole tego obszaru. Pasek menu jest pominięty, ponieważ nie znajduje się w obszarze roboczym okna, lecz w jego obszarze systemowym, podobnie jak pasek tytułowy i ramka. Dostęp do paneli paska statusu, reprezentowanych przez właściwość tablicową *Panels*, odbywa się za pomocą typowej notacji nawiasów kwadratowych.

Za każdym razem, gdy tworzony jest obiekt danej klasy, wywoływany jest jej konstruktor, który jest specjalną metodą przydzielającą obiektowi pamięć i inicjalizującą wszystkie jego pola. Standardowo konstruktor *Create* klasy *TObject*, od której wywodzą się wszystkie inne klasy, zeruje wszystkie pola obiektu. Wynika stąd, że w momencie utworzenia okna głównego naszej aplikacji pola  $xs$ ,  $ys$ ,  $Rd$ ,  $Rm$  i  $d$  uzyskują wartość zero. Gdyby aplikacja potrafiła rysować hipocykloidę, to każda próba jej uruchomienia kończyłaby się wyjątkiem *EDivByZero*, ponieważ w opisujących tę krzywą równaniach wystąpiłoby dzielenie przez zero (wartość pola  $Rm$ ). Jeżeli początkowe wartości pól obiektu mają być inne, trzeba zbudować własny konstruktor, który te pola odpowiednio zainicjalizuje.

W przypadku formularza istnieje wygodniejszy sposób dołączania dodatkowego kodu inicjalizującego. Polega on na wykorzystaniu zdarzenia *OnCreate*, które zachodzi właśnie w momencie, gdy okno jest tworzone [5, 6]. Konstruktor klasy

*TForm1* sprawdza, czy zdarzeniu *OnCreate* jest przypisana jakaś metoda, a jeśli tak, to ją wywołuje. Zatem aby prawidłowo zainicjalizować pola *xs*, *ys*, *Rd*, *Rm* i *d*, wybieramy w inspektorze obiektów formularz i definiujemy dla niego procedurę obsługi zdarzenia *OnCreate*. Delphi utworzy pusty szablon metody *FormCreate*, który uzupełniamy do postaci:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    CalculateCenter;  
    Rm := Min(xs, ys) div 7;  
    if Rm<1 then Rm := 1;  
    Rd := 5*Rm;  
    d := 2*Rm;  
    UpdateStatusBar;  
end;
```

Metoda najpierw inicjalizuje pola *xs* i *ys*, wywołując metodę *CalculateCenter*, potem inicjalizuje pola *Rm*, *Rd* i *d*, na koniec aktualizuje pasek statusu, wywołując metodę *UpdateStatusBar*. Wartości przypisywane polom *Rm*, *Rd* i *d* są tak dobrane, aby określały hipocykloidę złożoną z pięciu pętli ( $Rd/Rm = 5$ ), wypełniającą możliwie jak największą część obszaru roboczego okna. Metoda wykorzystuje funkcję *Min* (minimum z dwu wartości) zdefiniowaną w module *Math*<sup>18</sup>, dlatego do listy *uses* modułu należy dopisać nazwę *Math*.

Wypada zająć się teraz problemem rysowania hipocykloidy. Jeśli do tej pory nie zapisywaliśmy tworzonej aplikacji na dysku, czas najwyższy to zrobić. Przykładowo, moduł formularza zapisujemy w pliku *MainUnit.pas*, a projekt w pliku *Hipocykl.dpr*. Aplikację też kompilujemy i uruchamiamy, żeby sprawdzić, czy nie zawiera błędów kompilacji i czy działa poprawnie. Nie powinniśmy przejść do następnego etapu, dopóki nie da się skompilować lub zachowuje się wadliwie.

Rysowanie w Delphi odbywa się za pomocą służących do tego celu metod właściwości *Canvas*. Zważywszy na oszczędność pamięci, system Windows nie zachowuje w niej rysunku, ale każdorazowo, gdy okno wymaga przermalowania, wysyła do niego komunikat *WM\_PAINT*, który jest w programach tworzonych w Delphi odbierany jako zdarzenie *OnPaint*<sup>19</sup>. Metoda obsługująca to zdarzenie stanowi więc miejsce, w którym należy umieścić kod kreślący rysunek.

Zatem definiujemy dla formularza procedurę obsługi zdarzenia *OnPaint*. Szablon metody *FormPaint* utworzony przez Delphi rozbudowujemy, korzystając

---

<sup>18</sup> Moduł *Math* zawiera kilkadziesiąt użytecznych funkcji matematycznych, statystycznych i finansowych. Podstawowe funkcje pascalskie, takie jak *Abs*, *Cos*, *Round*, *Sin*, *Sqrt* i inne, są zawarte w module *System*, który jest automatycznie włączany do każdego modułu.

<sup>19</sup> System komunikatów Windows jest szczegółowo omówiony w książce [20] (język C).



z równań parametrycznych (D.1) oraz metod *MoveTo* i *LineTo* obiektu *Canvas* (przeniesienie pióra i rysowanie odcinka). Zamiast hipocykloidy rysujemy łamaną o odpowiednio dużej liczbie wierzchołków leżących na tej krzywej. Sama technika rysowania łamanej polega na przeniesieniu pióra do jej wierzchołka początkowego, a następnie rysowaniu wszystkich jej kolejnych odcinków łączących sąsiadujące dwa wierzchołki:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  k, n      : integer;
  fi, x, y: real;
begin
  n := (Rd div Rm)*(Rd - Rm + Abs(d));
  Canvas.MoveTo(xs + Rd - Rm + d, ys);
  for k:=1 to n do
    begin
      fi := 2*Pi*k/n;
      x := (Rd-Rm)*Cos(fi) + d*Cos(fi*(Rd-Rm)/Rm);
      y := (Rd-Rm)*Sin(fi) - d*Sin(fi*(Rd-Rm)/Rm);
      Canvas.LineTo(Round(x)+xs, ys-Round(y));
    end;
  end;
```

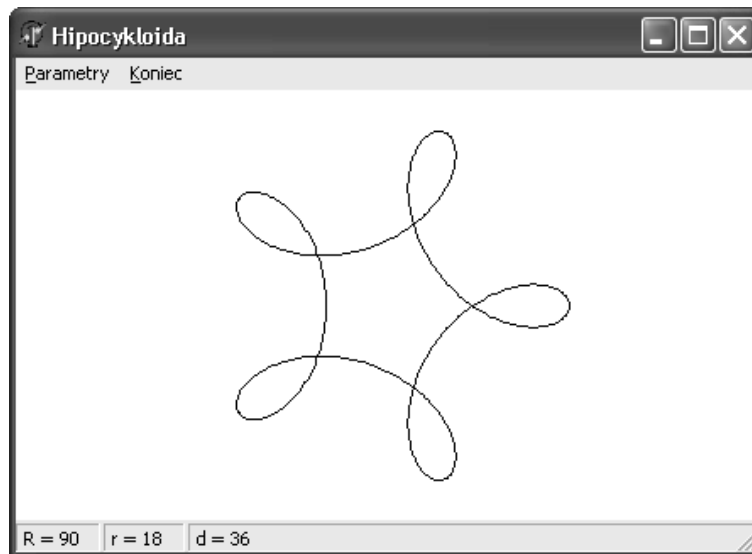
Napisanie kodu metody wymagało rozwiązania dwóch problemów: określenia liczby odcinków łamanej i odwzorowania tradycyjnego układu współrzędnych, w którym hipocykloida jest opisana, na układ ekranowy, w którym jest rysowana. Liczba odcinków powinna być niezbyt duża, ale jednocześnie wystarczająca, by obraz krzywej wiernie oddawał jej kształt. Wydaje się, że przyjęcie iloczynu liczby pętli hipocykloidy i długości promienia okręgu, w którym ta krzywa się mieści, jest satysfakcjonujące. Z kolei przekształcenie układu tradycyjnego na układ ekranowy wymaga przesunięcia osi  $x$  o wartość  $xs$  oraz odwrócenia osi  $y$  i przesunięcia jej o wartość  $ys$ <sup>20</sup>. Odwrócenie osi  $y$  wynika z ogólnie przyjętego założenia w świecie komputerowym, iż początek układu współrzędnych znajduje się w lewym górnym rogu obszaru roboczego okna (lub ekranu), a oś  $y$  jest skierowana w dół.

Funkcja *Round*, występująca dwukrotnie w wywołaniu metody *LineTo*, ma na celu zaokrąglenie liczby rzeczywistej do najbliższej liczby całkowitej. Użycie typu rzeczywistego spowodowałoby błąd niezgodności typów (ang. *incompatible types*), ponieważ współrzędne ekranowe muszą być liczbami całkowitymi. Zmienne  $x$ ,  $y$  można by usunąć, a przypisywane im wyrażenia umieścić bezpośrednio w wywołaniach funkcji *Round*, kod programu stałby się jednak mniej czytelny.

---

<sup>20</sup> W przypadku hipocykloidy zamkniętej, tj. gdy  $Rd$  jest podzielne przez  $Rm$ , odwrócenie osi  $y$  nie ma znaczenia z uwagi na symetrię tej krzywej względem prostej przechodzącej przez jej środek i równoległej do osi  $x$ .

Możemy wreszcie zobaczyć efekt działania metody *FormPaint*, gdy uruchomimy aplikację (rys. D.21). Kilka prób zmiany rozmiarów okna pokazuje, że krzywa nie zawsze jest rysowana pośrodku obszaru roboczego.



Rys. D.21. Wykres hipocykloidy w oknie aplikacji

Aby usunąć ten defekt, definiujemy procedurę obsługi zdarzenia *OnResize* formularza, które jest generowane przy każdej zmianie rozmiaru okna. Jej działanie sprowadza się do wyznaczenia współrzędnych nowego środka obszaru roboczego okna i poinformowania systemu Windows, że cała powierzchnia tego obszaru wymaga ponownego przermalowania:

```
procedure TForm1.FormResize(Sender: TObject);
begin
    CalculateCenter;
    Invalidate;
end;
```

Metoda *Invalidate* jedynie unieważnia obszar roboczy okna (ang. *invalidate area*). Oznacza to, że do kolejki komunikatów Windows oczekujących na realizację wstawiony zostaje komunikat *WM\_PAINT*. Gdy tylko Windows znajdzie czas, wyśle go do naszej aplikacji, a ta odbierze go jako zdarzenie *OnPaint* i odświeży zawartość obszaru roboczego okna, wywołując metodę *FormPaint*. Należy podkreślić, że **nie wolno bezpośrednio wywoływać metody *FormPaint***, by uaktualnić obszar roboczy okna.

Pozostało nam do zrobienia zaprogramowanie funkcjonalności menu. Podobnie jak w przypadku przycisków, definiujemy dla elementów *Parametry1* i *Koniec1* procedury obsługi zdarzenia *OnClick*. Jedyną różnicą jest to, że nie wybieramy obu tych komponentów na formularzu, lecz w oknie edytora menu. Utworzoną przez Delphi metodę *Parametry1Click* uzupełniamy na razie pustym komentarzem, a metodę *Koniec1Click* wywołaniem metody *Close* zamykającej okno:

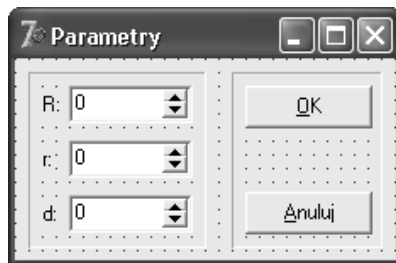
```
procedure TForm1.Parametry1Click(Sender: TObject);
begin
    //
end;

procedure TForm1.Koniec1Click(Sender: TObject);
begin
    Close;
end;
```

Naszym zadaniem jest teraz zastąpienie komentarza kodem umożliwiającym zmianę parametrów hipocykloidy, czyli wartości pól *Rd*, *Rm* i *d*. Najwygodniej jest posłużyć się **podrzędnym formularzem**, w którym te wartości będą wyświetlane i będzie je można modyfikować.

Istnieje kilka sposobów dodawania podrzędnego formularza do aplikacji. I tak, możemy nacisnąć przycisk *New Form* na pasku narzędziowym Delphi, skorzystać z polecenia *Form* dostępnego po rozwinięciu pozycji *New* w menu *File*, bądź też dwukrotnie kliknąć ikonę *Form* w oknie składnicy obiektów.

Po utworzeniu podrzędnego formularza nadajemy mu tytuł *Parametry* i blokujemy możliwość zmiany jego rozmiarów, ustawiając odpowiednio właściwości *BorderIcons* i *BorderStyle*. Następnie wstawiamy do niego po trzy komponenty *SpinEdit* (karta *Samples* palety komponentów) i opisujące je etykiety *Label* oraz po dwa przyciski *Button* i komponenty *Bevel*. Na koniec korygujemy ustawienia właściwości komponentów i rozmiary formularza. Wynik końcowy tych czynności jest pokazany na rysunku D.22.



Rys. D.22. Projekt podrzędnego formularza aplikacji

Kontrolka *SpinEdit* stanowi połączenie pola edycji *Edit* i przycisku *UpDown*, umożliwiając zwiększanie lub zmniejszanie wartości całkowitej za pomocą dwóch strzałek. Wartość bieżąca pola edycji określa właściwość *Value*, a krok, z jakim można ją zmieniać, właściwość *Increment* ustawiana domyślnie na 1. Właściwości *MinValue* i *MaxValue* pozwalają ograniczyć zakres edytowanej wartości.

Korzystamy z tego udogodnienia, ustawiając właściwość *MinValue* kontroltek *SpinEdit1* i *SpinEdit2* na 1, a właściwość *MaxValue* na 1000. Długość promieni obu okręgów wykorzystywanych w konstrukcji hipocykloidy będzie więc można zmieniać w zakresie od 1 do 1000, co przy obecnych rozdzielczościach ekranu jest ograniczeniem rozsądnym. Długość ramienia przytwierdzonego do mniejszego okręgu, określoną w kontrolce *SpinEdit3*, będzie można zmieniać dowolnie.

Gdy podczas wykonywania aplikacji wybierzemy w menu jej głównego okna polecenie *Parametry*, na ekranie powinno ukazać się drugie okno aplikacji, odpowiadające podrzędnemu formularzowi. Zakładamy, że będzie to **okno modalne** (ang. *modal window*), tzn. takie, że gdy pojawi się na ekranie, nie pozwoli na przejście do głównego okna aplikacji, dopóki nie zostanie zamknięte<sup>21</sup>.

Aby pokazać okno modalne, wywołujemy jego metodę *ShowModal*, zaś aby go zamknąć, przypisujemy jego właściwości *ModalResult* liczbę całkowitą różną od zera. Zazwyczaj jest nią predefiniowana wartość związana z przyciskiem, np. *mrOK* oznacza, że okno zostało zamknięte za pomocą przycisku *OK*, a *mrCancel*, że za pomocą przycisku *Anuluj* (ang. *Cancel*). Gdy przycisk zostaje kliknięty, jego właściwość *ModalResult* jest przypisywana właściwości *ModalResult* okna.

Właściwość *ModalResult* przycisku *Button2* o etykiecie *Anuluj* ustawiamy na *mrCancel*, ponieważ jego kliknięcie ma zawsze powodować zamknięcie okna. Inaczej jest w przypadku przycisku *Button1* o etykiecie *OK*: gdy okno zawiera nieprawidłowe dane, nie powinno dać się zamknąć. Pozostawiamy więc domyślną wartość *mrNone* (zero) przycisku *Button1*, a czynności, które będą wykonane, gdy zostanie użyty, zaprogramujemy, definiując procedurę obsługi zdarzenia *OnClick*. Kod metody, która obsługuje to zdarzenie, znajduje się w podanym niżej module podrzędnego formularza zapisanym w pliku *ParmUnit.pas*:

```
unit ParmUnit;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls, ExtCtrls, Spin;
```

---

<sup>21</sup> Okno niemodalne (ang. *modeless window*) pozwala na przejście do innego okna tej samej aplikacji. Zarządzanie oknami niemodalnymi jest nieco inne [5, 6].

```
type
  TForm2 = class(TForm)
    SpinEdit1: TSpinEdit;
    SpinEdit2: TSpinEdit;
    SpinEdit3: TSpinEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Button1: TButton;
    Button2: TButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    procedure Button1Click(Sender: TObject);
  end;

var
  Form2: TForm2;

implementation

{$R *.DFM}

procedure TForm2.Button1Click(Sender: TObject);
begin
  if SpinEdit2.Value < SpinEdit1.Value
    then ModalResult := mrOK else
    begin
      Application.MessageBox(
        'Nieprawidłowe dane (powinno być r<R)', 'Błąd',
        mb_IconExclamation or mb_OK);
      SpinEdit1.SetFocus;
    end;
end;

end.
```

Metoda *Button1Click* sprawdza, czy długość promienia toczącego się okręgu jest mniejsza od długości promienia nieruchomego okręgu. Jeżeli warunek ten jest spełniony, okno zostaje zamknięte w wyniku przypisania właściwości *ModalResult* wartości *mrOK*, a jeżeli nie, wyświetlone zostaje okno informujące o błędzie i sugerujące, jaką relację powinny spełniać długości promieni. Po zamknięciu okna komunikatu możemy od razu przystąpić do korekty danych, poczynając od długości promienia nieruchomego okręgu.

Wykorzystamy teraz zdefiniowane w module *ParmUnit* podrzędne okno aplikacji do zmiany parametrów hipocykloidy wyświetlanej w jej głównym oknie. Rozpoczynamy od rozszerzenia listy *uses* modułu *MainUnit* o nazwę *ParmUnit*, przez co uzyskujemy w nim dostęp do klasy *TForm2* i zmiennej globalnej *Form2* reprezentującej podrzędne okno. Wyświetlenie tego okna sprowadza się do zastą-

pienia pustego komentarza (//) w metodzie *Parametry1Click* wywołaniem metody *ShowModal* obiektu *Form2*:

```
Form2.ShowModal;
```

Gdy uruchomimy aplikację i wybierzemy polecenie *Parametry*, podrzędne okno pojawi się na ekranie, ale nie współpracuje z jej głównym oknem. Okno nie pokazuje parametrów wyświetlonej hipocykloidy, nie wymusza też jej przerysowania, gdy je zmienimy i zamknijemy go, klikając przycisk *OK*.

W tym miejscu musimy się zatrzymać na moment, by wyjaśnić dwie kwestie dotyczące zamykania podrzędnego formularza. Po pierwsze, jego metoda *Close* domyślnie ukrywa okno, lecz go nie niszczy. Okno nadal pozostaje w pamięci, chociaż jest niewidoczne. Można się do niego odwoływać, co umożliwia dostęp do jego pól. Po drugie, wynikiem wywołania metody *ShowModal*, która jest funkcją wywołującą metodę *Close*, jest wartość właściwości *ModalResult* zamykanego okna, co pozwala sprawdzić, w jaki sposób zostało ono zamknięte.

Można teraz zaprogramować w metodzie *Parametry1Click* przekazywanie parametrów hipocykloidy pomiędzy polami obu okien oraz odświeżanie zawartości głównego okna, gdy wynikiem wywołania funkcji *ShowModal* jest *mrOK*. Oto pełna wersja modułu *MainUnit*, która zawiera kod tej metody:

```
unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ComCtrls, Menus, Math, ParmUnit;

type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    Parametry1: TMenuItem;
    Koniec1: TMenuItem;
    StatusBar1: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure Parametry1Click(Sender: TObject);
    procedure Koniec1Click(Sender: TObject);
  private
    xs, ys, Rd, rm, d: integer;
  public
    procedure CalculateCenter;
    procedure UpdateStatusBar;
  end;
```

```
var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.CalculateCenter;
begin
    xs := ClientWidth div 2;
    ys := (ClientHeight - StatusBar1.Height) div 2;
end;

procedure TForm1.UpdateStatusBar;
begin
    StatusBar1.Panels[0].Text := ' R = ' + IntToStr(Rd);
    StatusBar1.Panels[1].Text := ' r = ' + IntToStr(Rm);
    StatusBar1.Panels[2].Text := ' d = ' + IntToStr(d);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    CalculateCenter;
    Rm := Min(xs, ys) div 7;
    if Rm < 1 then Rm := 1;
    Rd := 5 * Rm;
    d := 2 * Rm;
    UpdateStatusBar;
end;

procedure TForm1.FormPaint(Sender: TObject);
var
    k, n      : integer;
    fi, x, y: real;
begin
    n := (Rd div Rm) * (Rd - Rm + Abs(d));
    Canvas.MoveTo(xs + Rd - Rm + d, ys);
    for k:=1 to n do
        begin
            fi := 2*Pi*k/n;
            x := (Rd-Rm)*Cos(fi) + d*Cos(fi*(Rd-Rm)/Rm);
            y := (Rd-Rm)*Sin(fi) - d*Ssin(fi*(Rd-Rm)/Rm);
            Canvas.LineTo(Round(x)+xs, ys-Round(y));
        end;
    end;

procedure TForm1.FormResize(Sender: TObject);
begin
    CalculateCenter;
    Invalidate;
end;
```

```
procedure TForm1.Parametry1Click(Sender: TObject);
begin
    Form2.SpinEdit1.Value := Rd;
    Form2.SpinEdit2.Value := Rm;
    Form2.SpinEdit3.Value := d;
    if Form2.ShowModal = mrOK then
    begin
        Rd := Form2.SpinEdit1.Value;
        Rm := Form2.SpinEdit2.Value;
        d  := Form2.SpinEdit3.Value;
        UpdateStatusBar;
        Invalidate;
    end;
end;

procedure TForm1.Koniec1Click(Sender: TObject);
begin
    Close;
end;

end.
```

Metoda *Parametry1Click* najpierw przesyła do pól podrzędnego okna, które jest ukryte, wartości pól głównego okna określające parametry wyświetlonej w nim hipocykloidy. Następnie wyświetla podrzędne okno, które udostępnia parametry hipocykloidy i umożliwia ich modyfikację. Gdy użytkownik kliknie przycisk *OK* lub *Anuluj*, podrzędne okno zostaje ukryte. Jeżeli był to przycisk *OK*, metoda przesyła wartości pól podrzędnego okna do pól głównego okna i odświeża najpierw pasek statusu głównego okna, a potem zawartość jego obszaru roboczego.



## Literatura

- [1] Aho A.V., Hopcroft J.E., Ullman J.D.: *Algorytmy i struktury danych*, Helion, Gliwice 2003.
- [2] Aho A.V., Hopcroft J.E., Ullman J.D.: *Projektowanie i analiza algorytmów*, Helion, Gliwice 2003.
- [3] Banachowski L., Diks K., Rytter W.: *Algorytmy i struktury danych*, wyd. 4, WNT, Warszawa 2003.
- [4] Banachowski L., Kreczmar A.: *Elementy analizy algorytmów*, WNT, Warszawa 1982 (istnieje nowsze wydanie).
- [5] Cantù M.: *Delphi 4. Praktyka programowania*, MIKOM, Warszawa 1999.
- [6] Cantù M.: *Delphi 6. Praktyka programowania*, t. 1 – 2, MIKOM, Warszawa 2002 (istnieją wydania dla Delphi 5 i 7).
- [7] Cormen T.H., Leiserson C.E., Rivest R.L.: *Wprowadzenie do algorytmów*, wyd. 2, WNT, Warszawa 1998 (istnieje nowsze wydanie).
- [8] Drozdek A.: *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004.
- [9] Dryja M., Jankowscy J. i M.: *Przegląd metod i algorytmów numerycznych*, cz. 2, WNT, Warszawa 1982 (istnieje nowsze wydanie).
- [10] Graham R.L., Knuth D.E., Patashnik O.: *Matematyka konkretna*, PWN, Warszawa 2001.
- [11] Jakubczyk K.: *Turbo Pascal i Borland C++. Przykłady*, Helion, Gliwice 2002, 2006.
- [12] Jankowscy J. i M.: *Przegląd metod i algorytmów numerycznych*, cz. 1, WNT, Warszawa 1981 (istnieje nowsze wydanie).
- [13] Kernighan B.W., Pike R.: *Lekcja programowania*, WNT, Warszawa 2002.
- [14] Kimmel P.: *Delphi 6 dla profesjonalistów*, RM, Warszawa 2001.
- [15] Knuth D.E.: *Sztuka programowania*, t. 1: *Algorytmy podstawowe*, WNT, Warszawa 2002.
- [16] Knuth D.E.: *Sztuka programowania*, t. 2: *Algorytmy seminumeryczne*, WNT, Warszawa 2002.
- [17] Knuth D.E.: *Sztuka programowania*, t. 3: *Sortowanie i wyszukiwanie*, WNT, Warszawa 2002.
- [18] Lipski W.: *Kombinatoryka dla programistów*, WNT, Warszawa 1989 (istnieje nowsze wydanie).

- [19] Neapolitan R., Naimipour K.: *Podstawy algorytmów z przykładami w C++*, Helion, Gliwice 2004.
- [20] Petzold Ch.: *Programowanie w Windows*, wyd. 2, RM, Warszawa 1999.
- [21] Ross K.A., Wright C.R.B.: *Matematyka dyskretna*, wyd. 3, PWN, Warszawa 2000.
- [22] Stephens R.: *Algorytmy i struktury danych z przykładami w Delphi*, Helion, Gliwice 2000.
- [23] Sysło M.M.: *Algorytmy*, wyd. 3, WSiP, Warszawa 2002.
- [24] Turski W.M.: *Propedeutyka informatyki*, PWN, Warszawa 1979 (istnieje nowsze wydanie).
- [25] Walmsley S., Williams S.: *Delphi*, RM, Warszawa 1999.
- [26] Walmsley S., Williams S.: *Pascal w środowisku Delphi*, RM, Warszawa 2003.
- [27] Wilson R.J.: *Wprowadzenie do teorii grafów*, wyd. 2, PWN, Warszawa 2002.
- [28] Wirth N.: *Algorytmy + struktury danych = programy*, WNT, Warszawa 1980 (istnieją nowsze wydania).
- [29] Wirth N.: *Wstęp do programowania systematycznego*, WNT, Warszawa 1978.
- [30] Wróblewski P.: *Algorytmy, struktury danych i techniki programowania*, wyd. 3, Helion, Gliwice 2003.

## Skorowidz

### A

**abstract** 127

*Add*, metoda 116, 119, 143

*AddObject*, metoda 119

Algol, język 101

algorytm 5, 9 – 19, 26 – 34, 42

–, analiza 30 – 32, 39

– BFS 157 – 159

–, czas wykonania 28, 30

– DFS 155 – 158

– Dijkstry 169 – 174, 176, 179, 180

–, dobre zdefiniowanie 9

–, efektywność 39

– Euklidesa 15 – 17, 27, 41 – 44, 52

– Floyd-Warshalla 175 – 176, 188

– Hoare’a 66, 88, 91

– iteracyjny 42, 49

– numerycznie stabilny 15

– poprawny 9

– przesiewania 74 – 76, 124

– rekurencyjny 42, 189, 194

– – bezpośrednio 189, 196

– – pośrednio 189, 196

– Shella 100

– sortowania 17, 18

–, weryfikacja 9

– wielomianowy 41 – 42

– wyznaczania spójnych składowych  
168

– zachłanny 50, 172

–, złożoność obliczeniowa 30

– z powrotami (nawrotami) 51, 204,  
211

*Align*, właściwość 117

analiza algorytmu 30 – 32, 39

– numeryczna 15

analizator składni 223

*AnsiLowerCase*, funkcja API 120

API, biblioteka 199

aplikacja 241 – 243, 246 – 250, 252

– konsolowa 255

– okienkowa 255, 260

*Application*, obiekt 85, 260, 273

APSP, problem 174 – 175

arytmetyka zmiennopozycyjna 12, 13

### B

*Bevel*, komponent 19, 82, 199, 265

BFS, algorytm 157 – 159

*BitBtn*, komponent 177, 191, 199

błąd działań zmiennopozycyjnych 12

Borland 5, 116, 236, 237

*BucketFor*, metoda 143

*Button*, komponent 19, 247, 283

### C

C, C++, język 6, 16, 17, 21, 26, 101,  
110, 116, 127, 196, 271

*Canvas*, właściwość 198, 199, 253,  
276, 280, 281

*Capacity*, właściwość 115

*Caption*, właściwość 81, 246, 249

cecha 52

ciąg Fibonacciego 45 – 47, 95 – 96

*Clear*, metoda 120

*Close*, metoda 118, 132, 200, 250,  
259, 266, 283, 286

CLX, biblioteka 245

Cobol, język 26

*Color*, właściwość 196, 249, 264

*ComboBox*, komponent 177

*Compare*, metoda 127, 1291, 130

*Copy*, metoda 29, 269

*Count*, właściwość 115, 127

CR, powrót karetki 116, 227, 228  
*Create*, metoda 118, 252, 274, 279  
cykl 149  
– Eulera 148  
– Hamiltona 239

## D

*DateToStr*, funkcja 260  
*DecodeTime*, procedura 28  
*DefaultColWidth*, właściwość 161  
*Delete*, metoda 116  
*Delete*, procedura 269  
Delphi 5, 14, 29, 30, 33, 63, 82, 101, 115, 123, 126, 143, 241, 244, 246, 250, 251, 253 – 266, 271, 273, 276, 277, 279, 280  
destruktor 252  
DFS, algorytm 155 – 158  
diagram syntaktyczny 224 – 225  
digraf 148  
Dijkstra E. 10  
*Dispose*, procedura 132, 139  
długość ścieżki 149, 169  
dobre zdefiniowanie algorytmu 9  
dokańczanie klas 21, 262, 279  
dowiązanie 5, 58, 101  
*DrawGrid*, komponent 213  
droga (ścieżka) 149  
– (cykl) Hamiltona 239  
drzewo 147, 150  
– binarne 71, 124, 151, 207  
– – zupełne 124  
–, korzeń 46, 150 – 151  
– najkrótszych ścieżek 174  
– przeszukiwań 156, 159, 207, 213  
– –, obcięcie 207, 213  
– rozpinające grafu 147, 159 – 160  
–, wysokość 151

## E

*EConvewrtError*, klasa 275 – 276

*Edit*, komponent 247, 264  
*EDivByZero*, klasa 279  
edytor kodu 242, 248 – 250, 256, 262  
– listy łańcuchów 264, 265  
– menu 277  
– rysunków 214  
efektywność czasowa 31, 25, 38, 71, 80, 84, 87, 140  
egzemplarz problemu 9 – 10, 30  
*Enabled*, właściwość 20, 249  
Euklides 15, 16  
Euler L. 147  
*Exception*, klasa 274, 276  
*Exchange*, metoda 116  
*Execute*, metoda 118, 252  
*Exists*, metoda 143

## F

FIFO, kolejka 104, 124  
*FileSize*, funkcja 96  
*Filter*, właściwość 116  
*Find*, metoda 143  
*First*, metoda 116  
*FloatToStr*, funkcja 260  
Floyd R.W. 73  
*ForEach*, metoda 143  
**for each** 136, 168  
*Format*, funkcja 260  
*FormatDateTime*, funkcja 83  
*FormCreate*, metoda 280  
*FormPaint*, metoda 199, 280, 282  
formularz 242, 246 – 247, 249 – 258, 260 – 266, 279 – 280, 282  
– podrzędny 283 – 284, 286  
–, skrypt 261  
– wirtualny 197 – 199, 201  
Fortran, język 26, 101  
**forward** 196  
*Free*, metoda 118, 252  
funkcja Ackermanna 234 – 235  
– API 199

- *AnsiLowerCase* 120
- *PostMessage* 199
- *SendMessage* 199
- *SetWindowOrgEx* 199
- haszująca 140
- klucza 57
- mieszająca 140 – 143
- porządkująca 57

## G

- Gauss C.F. 210
- graf 147 – 148
  - acykliczny 149, 150
  - , drzewo (las) rozpinające 159
  - etykietowany 154
  - , najkrótsza ścieżka 147
  - nieskierowany 148
  - niezorientowany 148
  - , pętla 148
  - planarny 148
  - , przeszukiwanie w głąb 154 – 157
  - , – wszerz 157 – 159
  - , rozmiar 151
  - rzadki 153
  - skierowany 148
  - , spójna składowa 147, 150, 167
  - , spójność silna 150, 167, 187
  - , – słaba 150, 167, 187
  - spójny 167
  - ważony 154, 169
  - zorientowany 148
- GroupBox*, komponent 81
- GUI, graficzny interfejs użytkownika 241, 247, 255, 263

## H

- haszowanie 140
- Hide*, metoda 246, 252
- High*, funkcja 21, 23
- hipocykloida 276 – 277, 281 – 282
- Hoare C.A.R 66, 88

*HorzScrollBar*, właściwość 197

## I

- IBM, firma 140
- IDE, środowisko programowania 241
- Image Editor*, aplikacja 214
- ImageList*, komponent 161
- implementacja 17, 19, 26, 35, 39, 43,
  - 60, 62, 64, 69, 78, 88, 90, 135,
  - 137 – 139, 141, 162, 179, 212
- listowa nieuporządkowana 138
- – uporządkowana 139
- listy dowiązaniowa 105, 109 – 113
- – tablicowa 105 – 109
- – – cykliczna 108 – 109
- – w Delphi 115 – 116, 127
- indeks, zbiór 58
- IndexOf*, metoda 116, 119
- Insert*, metoda 116
- inspektor obiektów 242, 248 – 252
- interfejs 191, 242, 245, 246, 252, 263
- interpretator 223 – 224
- IntToRom*, funkcja 267, 269, 274
- IntToStr*, funkcja 260, 267
- Invalidate*, metoda 194, 216, 252, 282
- Items*, właściwość 22, 81, 115, 116, 119, 226, 265, 277

## J

- Java, język 6, 127, 271

## K

- klasa 245
  - abstrakcyjna 116
  - bazowa 131
  - *EConvertError* 275 – 276
  - *Exception* 274, 276
  - formularza 117, 196, 201
  - , metoda 245 – 246, 251 – 253, 257
  - , pole 245, 257, 278

- klasa potomna 116, 127, 130, 131, 258
  - , sekcja chroniona, **protected** 258
  - , sekcja opublikowana, **published** 258
  - , – prywatna, **private** 82, 258
  - , – publiczna, **public** 83, 258
  - *TBitmap* 215
  - *TBucketList* 143
  - *TFloatHeap* 129 – 132
  - *THeap* 126 – 131
  - *TList* 115 – 116, 119, 137, 143
  - *TObject* 119, 259, 279
  - *TObjectBucketList* 143
  - *TObjectList* 116
  - *TObjectQueue* 123
  - *TObjectStack* 123, 124
  - *TOrderedList* 126, 127, 137, 143
  - *TQueue* 123, 126, 129
  - *TStack* 123, 124, 126, 129
  - *TStringList* 116, 118 – 120, 137, 143
  - *TStrings* 22, 116
  - Kind*, właściwość 177, 191, 199
  - klika 239
  - klucz 57 – 59, 87, 135, 140 – 142
  - kolejka 102, 107 – 114, 131, 157
    - dwukierunkowa 103 – 105
    - jednokierunkowa 103 – 105
    - komunikatów 126, 199, 282
    - priorytetowa 126, 134, 151
  - kolizja 141
  - kolorowanie grafu 239
  - komórka pamięci 30, 59
  - kompilator 223
  - komponent 244 – 245
    - *Bevel* 19, 82, 199, 265
    - *BitBtn* 177, 191, 199
    - *Button* 19, 247, 283
    - *ComboBox* 177
    - *DrawGrid* 213
    - *Edit* 247, 264
    - *GroupBox* 81
    - *ImageList* 161
    - *Label* 283
    - *ListBox* 19, 22, 116, 118, 130, 132, 226
    - *MainMenu* 116, 196, 277
    - *Memo* 226
    - niewidzialny 247
    - *OpenDialog* 116, 118, 161, 177, 252
    - *RadioGroup* 81, 247
    - *SpinEdit* 19, 82, 130, 191, 199, 283
    - *Splitter* 226
    - *StringGrid* 161, 177, 213
    - *ToolBar* 161
    - *ToolButton* 161
    - widzialny 247
  - komunikat 85, 126, 199, 280
    - *WM\_PAINT* 126, 280, 282
    - *WM\_TIMER* 126
    - *WM\_USER* 199, 201
  - konstruktor 118, 252, 274, 279
  - kontrolka 247
  - kopiec 71, 124, 147
    - , konstrukcja 76
    - , uporządkowanie 72, 124
    - zupełny 72
    - –, wysokość 72, 126
  - korzeń 71, 78, 124, 150
  - krawędź 71, 147 – 148, 207
  - krzywa Hilberta 236
    - Kocha 235 – 236
    - Sierpińskiego 194 – 204
- L**
- Label*, komponent 283
  - las rozpinający grafu 159 – 161
  - Last*, metoda 116
  - Length*, funkcja 21, 269
  - LF, wysuw wiersza 116, 227, 228

- liczby Fibonacciego 45 – 47, 49, 95, 96, 190
- rzymskie 268
- Stirlinga 56
- LIFO, stos 104, 123
- LineTo*, metoda 253, 281
- List*, właściwość 127, 128
- lista 101 – 103 124, 134, 137, 142
- cykliczna 115
- dwukierunkowa 102, 103
- jednokierunkowa 101, 110, 112
- kroków algorytmu 15
- sąsiedztwa 151 – 153
- ListBox*, komponent 19, 22, 116, 118, 130, 132, 226
- liść 71, 74, 76, 78, 124, 151
- Low*, funkcja 21, 63
- LParam*, pole komunikatu 201
- Ł**
- łańcuch 29, 116, 190, 260, 267, 269
- łuk 148
- Łukasiewicz J. 144
- M**
- macierz kosztów 168, 169, 175
- sąsiedztwa 151 – 152, 161, 163
- MainMenu*, komponent 116, 196, 277
- mantysa 52
- mapa bitowa 214, 276
- Math*, moduł 280
- MaxValue*, właściwość 20, 82, 130, 191, 199, 284
- mediana 67, 70, 88, 98
- Memo*, komponent 226
- message** 201
- MessageBox*, metoda 273
- metoda 245 – 246, 251 – 253, 257
- abstrakcyjna 127
- *Add* 116, 119, 143
- *AddObject* 119
- bisekcji (połowienia) 52
- *BucketFor* 143
- *Clear* 120
- *Close* 118, 250, 259, 266, 283, 286
- *Compare* 127, 129, 130
- *Create* 118, 252, 274, 279
- *Delete* 116
- „dziel i zwyciężaj” 48, 49, 67
- *Exchange* 116
- *Execute* 118, 252
- *Exists* 143
- *Find* 143
- *First* 116
- *ForEach* 143
- *FormCreate* 280
- *FormPaint* 199, 280, 282
- *Free* 118, 252
- Gaussa 12
- *Hide* 246, 252
- *IndexOf* 116, 119
- *Insert* 116
- *Invalidate* 194, 282
- *Last* 116
- *LineTo* 253, 281
- *MessageBox* 273
- *Move* 116
- *MoveTo* 253, 281
- *Peek* 123, 127, 129
- *PeekItem* 127
- podstawienia zmiennej 36, 43, 70
- *Pop* 123, 127, 129, 132
- *PopItem* 127
- *ProcessMessages* 85
- prób i błędów 48, 51, 204, 212
- *Push* 123, 127, 129, 132
- *PushItem* 127
- *Remove* 116, 143
- *Show* 246, 253
- *ShowModal* 253, 284, 286
- *Sort* 116
- wirtualna 127

metoda zachłanna 48, 50  
metody numeryczne 15  
mieszanie 140  
*MillisecondsBetween*, funkcja 28  
*MinValue*, właściwość 20, 82, 130, 199, 284  
*ModalResult*, właściwość 284, 286  
moduł 23, 128, 162, 178, 241  
moduł formularza 24, 120, 132, 165, 183, 192, 216, 256  
–, sekcja prywatna, **implementation** 257  
–, – publiczna, **interface** 257  
– standardowy 260  
*Move*, metoda 116  
*MoveTo*, metoda 253, 281  
Muhammad ibn Musa al-Chorezmi 9  
multigraf 151

## N

najkrótsza ścieżka 147, 168 – 175  
największy wspólny dzielnik (NWD) 15 – 16, 42 – 43, 52  
*New*, funkcja, procedura 132, 138  
**nil**, wskaźnik 21, 101, 111  
notacja asymptotyczna 40  
– funkcyjna abstrakcyjna 111  
– *O* (duże *O*) 40, 42  
– odwrotna polska (ONP) 144 – 145  
– przyrostkowa (postfiksowa) 144  
– wzrostkowa (infiksowa) 144  
*NULL*, wskaźnik 101  
NWD 15, 16, 41, 42

## O

obcięcie drzewa przeszukiwań 207, 213  
obiekt 244 – 245  
Object Pascal 6, 12, 22, 43, 62, 64, 69, 90, 96, 245, 250, 260, 268  
*Objects*, właściwość 119

obsługa wyjątków 271  
obszar roboczy 117, 126, 191, 194, 197, 250, 259, 280, 282, 288  
ograniczenie asymptotyczne 40, 42  
okno główne 199, 250, 259, 288  
– modalne 284  
– niemodalne 284  
– podrzędne 199, 285, 288  
*OnChange*, zdarzenie 20, 177, 194, 251  
*OnClick*, zdarzenie 20, 84, 118, 161, 200, 216, 246, 250, 251, 261  
*OnCreate*, zdarzenie 20, 82, 131, 183, 197, 215, 252, 279 – 280  
*OnDestroy*, zdarzenie 131, 215, 252  
*OnDrawCell*, zdarzenie 216  
ONP (odwrotna notacja polska) 144 – 145  
*OnPaint*, zdarzenie 192, 198, 252, 280, 282  
*OnResize*, zdarzenie 194, 197, 252, 282  
OOP, programowanie obiektowe 245  
*OpenDialog*, komponent 116, 118, 161, 177, 252  
operacja *add* 103  
– *construct* 74, 76, 79, 137 – 139  
– *delete* 103, 135, 137 – 139, 140  
– *deletemax* 126, 128  
– dominująca 30, 33, 35, 39, 59, 92  
– *eject* 103, 108, 114  
– *front* 103  
– *inject* 103, 108, 113, 114  
– *insert* 126, 128, 135, 137 – 140  
– *makenull* 137  
– *member* 135, 137  
– *pop* 103, 106, 108, 112 – 114, 172  
– *push* 103, 106, 108, 112, 114  
– *rear* 103  
– *search* 137 – 139, 140  
– *sort* 74, 76, 77, 79, 80



*Options*, właściwość 177  
oś podziału 65, 88 – 90  
**override** 127  
*OwnsObjects*, właściwość 116

## P

paleta komponentów 246 – 247  
Pascal, język 5, 6, 13, 16 – 19, 21, 26, 97, 101, 102, 109, 110, 135, 136, 196, 206, 241, 245, 278  
pasek przewijania 197 – 198, 201, 214, 247  
*Peek*, metoda 123, 127, 129  
*PeekItem*, metoda 127  
*PenPos*, właściwość 198, 253  
pióro 198, 243  
podłoga 36  
*pointer*, typ 115, 119  
*Pop*, metoda 123, 127, 129, 132  
*PopItem*, metoda 127  
*Position*, właściwość 198  
*PostMessage*, funkcja API 199  
problem APSP 174  
– kolorowania grafu 239  
– komiwojażera 148  
– ośmiu hetmanów 210, 213 – 219  
– plecakowy 51, 205, 208  
– sortowania 30, 57  
–, specyfikacja 9  
– wydawania reszty 50  
– wypłacalności kwoty 219  
priorytet 126, 145  
procedura obsługi zdarzenia 20, 118, 131, 161, 192, 197, 200, 246  
*ProcessMessages*, metoda 85  
programowanie dynamiczne 48, 49, 175  
– obiektowe, OOP 245  
projekt aplikacji 253 – 254, 260, 263  
– formularza 19, 81, 130, 196, 242  
przeciążanie (przeładowywanie) 23

przekazywanie parametrów przez  
wartość 17, 111, 158  
– – – zmienną 19, 83, 201  
przesiewanie 62, 74 – 76, 80  
przeszukiwanie 32  
– binarne 34 – 36, 38, 41, 44, 48, 63, 65, 66, 87, 111, 116, 139, 140  
– grafu w głąb 154 – 157  
– – wszerek 157 – 159  
– sekwencyjne 33 – 35, 38, 41, 60  
pseudojęzyk 5, 16 – 19, 26, 32, 33, 35, 58, 60, 67, 77, 90, 111, 138  
*Push*, metoda 123, 127, 129, 132  
*PushItem*, metoda 127

## R

RAD, szybkie tworzenie aplikacji 241  
*RadioGroup*, komponent 81, 247  
**raise** 274  
ramka stosu 43 – 44, 189  
randomizacja 144  
*Range*, właściwość 197  
rekord 57 – 59, 87, 91, 101, 198  
– aktywacji 187  
rekurencja 7, 35 – 37, 42 – 49, 51, 67, 154, 189 – 196, 204, 223  
– końcowa (ogonowa) 45  
–, schemat 190, 194 – 196, 236  
*Remove*, metoda 116, 143  
repozytorium obiektów 254, 255  
**return** 17  
*RomToInt*, funkcja 267, 269, 274, 275  
rozdzielanie 92, 97  
rozmiar danych 30, 38, 42  
– grafu 151  
– problemu (zadania) 30  
rozpraszanie 140, 144  
równanie rekurencyjne 69, 70  
rząd wielkości 38 – 40

**S**

- scalanie naturalne 94, 97
  - proste 92 – 94, 97
  - wielofazowe 94 – 96
- schemat blokowy 26 – 27
  - rekurencji 190, 194 – 196, 236
- sekcja chroniona klasy 258
  - finalizacji modułu 257
  - implementacji modułu 257, 258
  - inicjalizacji modułu 257
  - interfejsu modułu 257
  - opublikowana klasy 258
  - prywatna klasy 258
  - publiczna klasy 258
- seria fikcyjna 95 – 96
  - uporządkowana 91
- SendMessage*, funkcja API 199
- set**, typ 135, 179, 188, 261
- SetLength*, procedura 21, 101, 162
- SetWindowOrgEx*, funkcja API 199
- Show*, metoda 246, 253
- ShowModal*, metoda 253, 284, 286
- siatka 161, 165, 177, 182, 213 – 216
- sieć działań 26
- składnica obiektów 254, 255, 260
- skończoność algorytmu 9
- skrypt formularza 261
- słownik 102, 136 – 140
- Sort*, metoda 116
- Sorted*, właściwość 117
- sortowanie 57
  - bąbelkowe, *bubble sort* 18, 23, 28, 29, 30, 31, 38, 39, 59, 71, 87
  - kopcowe, *heap sort* 71, 74 – 80, 87, 124, 151
  - przez selekcję, *selection sort* 59 – 61, 144
  - przez wstawianie, *insertion sort* 61 – 63, 87, 100
  - – – binarne 63 – 66, 87
  - sekwencyjne 91
  - , specyfikacja problemu 57
  - stogowe (kopcowe) 71
  - szybkie, *quick sort* 66 – 71, 87, 91, 116
  - wewnętrzne 57, 59, 80
  - wielofazowe 94 – 96
  - za pomocą malejących przyrostów 100
  - zewnętrzne 57, 91
- specyfikacja problemu 9, 17
  - – APSP 174
  - – przeszukiwania 32
  - – sortowania 57
  - – znajdowania mediany 88
  - – – najkrótszych ścieżek 168
  - – – spójnych składowych 167
- SpinEdit*, komponent 19, 82, 130, 191, 199, 283
- Splitter*, komponent 226
- spójna składowa 147, 150
- statystyka pozycyjna 88
- sterta 124
- stos 102, 104 – 106, 112, 123, 158
  - , dno 104 – 106
  - , wierzchołek 104 – 106, 123
- stóg 71, 124
- StringGrid*, komponent 161, 177, 213
- Strings*, właściwość 119
- StrToDate*, funkcja 260
- StrToFloat*, funkcja 260
- StrToInt*, funkcja 260, 267
- struktura dowiązaniowa 6, 101
  - dynamiczna 101
- Style*, właściwość 177
- sufit 36
- syntaktyka 223
- System*, moduł 280
- SysUtils*, moduł 28, 260, 267

**Ś**

ścieżka 148

- , długość 149, 169
- prosta 149
- zamknięta 149

**T**

- tablica dynamiczna 21, 29, 135, 138, 162, 208
- mieszająca 141
- otwarta 22, 64
- TBitmap*, klasa 215
- TBucketList*, klasa 143
- Text*, właściwość 249, 264
- TFloatHeap*, klasa 129 – 132
- THeap*, klasa 126 – 131
- Time*, funkcja 28
- TList*, klasa 115 – 116, 119, 137, 143
- TMessage*, typ 201
- TObject*, klasa 119, 259, 279
- TObjectBucketList*, klasa 143
- TObjectList*, klasa 116
- TObjectQueue*, klasa 123
- TObjectStack*, klasa 123, 124
- ToolBar*, komponent 161
- ToolButton*, komponent 161
- TOrderedList*, klasa 126, 127, 137, 143
- TQueue*, klasa 123, 126, 129
- TPoint*, typ 198
- Tracking*, właściwość 197
- translator 223
- trójkąt Pascala 55
- Stirlinga 56
- try** 24, 85, 118, 271, 272, 274
- TStack*, klasa 123, 124, 126, 129
- TStringList*, klasa 116, 118 – 120, 137, 143
- TStrings*, klasa 22, 116
- Turbo Pascal 5, 6, 12, 22, 96, 116, 142, 237, 255, 259, 269
- typ wariantowy, *variant* 23

**U**

- uporządkowanie kopcowe 72, 124
- , przywracanie 74 – 76, 125
- uses** 23, 200, 270, 274

**V**

- Value*, właściwość 20, 82, 130
- variant*, typ 23
- VCL, biblioteka 126, 245, 247, 253
- VertScrollBar*, właściwość 197
- virtual** 127

**W**

- waga 154, 168, 172, 268
- wartość oczekiwana 33
- wartownik 63
- weryfikacja algorytmu 10
- wektor bitów 135
- charakterystyczny 135, 179
- węzeł 71, 148
- wielomian Czebyszewa 55
- wierzchołek 71, 78, 124, 147 – 148, 207
- , głębokość 151
- incydentny 148
- , następnik (potomek) 151
- , numer poziomu 151
- , poprzednik (przodek) 151
- sąsiedni (sąsiadujący) 148
- , stopień 187
- stosu 104
- wewnętrzny 151
- , wysokość 151
- wieże Hanoi 237
- Williams J. 73
- Windows 6, 85, 118, 126, 199, 201, 224, 242, 243, 245, 246, 252, 254, 263, 280, 282
- Wirth N. 102, 211
- właściwość 245, 248, 253
- *Align* 117

właściwość *Canvas* 198, 199, 253, 276, 280, 281

- *Capacity* 115
- *Caption* 81, 246, 249
- *Color* 196, 249, 264
- *Count* 115, 127
- *DefaultColWidth* 161
- *Enabled* 20, 249
- *Filter* 116
- *HorzScrollBar* 197
- *ItemIndex* 81
- *Items* 22, 115, 116, 226
- *Kind* 177, 191, 199
- *List* 127, 128
- *MaxValue* 20, 82, 130, 191, 199
- *MinValue* 20, 82, 130, 199
- *ModalResult* 284, 286
- *Objects* 119
- *Options* 177
- *OwnsObjects* 116
- *PenPos* 198, 253
- *Position* 198
- *Range* 197
- *Sorted* 117
- *Strings* 119
- *Style* 177
- *Text* 249, 264
- *Tracking* 197
- *Value* 20, 82, 130
- *VertScrollBar* 197
- zagnieżdżona 249

*WM\_PAINT*, komunikat 126, 280, 282

*WM\_TIMER*, komunikat 126

*WM\_USER*, komunikat 199, 201

*WParam*, pole komunikatu 201

wskaźnik 5, 58, 101, 109 – 110, 112

współczynnik proporcjonalności 31, 39, 43, 44, 79

- Newtona 55 – 56

„wyciek” pamięci 120

wyjątek 118, 270, 274

wysokość drzewa 151

- kopca zupełnego 72, 126

wywołanie rekurencyjne 43, 190, 192, 195, 205

wzory Viète’a 13

wzór Stirlinga 65

## Z

zachłanność 50, 172, 205, 220

zagliodzenie procesu 126

zbiór 102, 134

zdarzenie 245, 248, 253

- *OnChange* 20, 177, 194, 251
- *OnClick* 20, 84, 118, 161, 200, 216, 246, 250, 251, 261
- *OnCreate* 20, 82, 131, 183, 197, 215, 252, 279 – 280
- *OnDestroy* 131, 215, 252
- *OnDrawCell* 216
- *OnPaint* 192, 198, 252, 280, 282
- *OnResize* 194, 197, 252, 282

złożoność obliczeniowa 30, 135

- czasowa 38, 40, 42, 44, 49, 59, 63, 142, 159
- kwadratowa 41
- liniowa 41, 47
- liniowo-logarytmiczna 41
- logarytmiczna 41, 44
- obliczeniowa 30, 42
- oczekiwana 31
- pamięciowa 30, 44, 59
- pesymistyczna 31, 126
- stała 41
- sześcienna 41
- wielomianowa 41

## Ź

źródło 169

## Summary

The **Introduction to Algorithms and Data Structures** describes the basic algorithmic problems which are currently considered classic. Perhaps, the set of discussed issues seems to be limited, but it is sufficient to create almost any type of efficient program. The pseudo-language close to Pascal, being a blend of Pascal constructions and natural language expressions, is used in description of algorithms and data structures. This approach has a merit that pseudo-language descriptions are clear and intelligible. Moreover, they can be easily transformed into correct Pascal programs which can be compiled and run. Many algorithms are presented in the form of complete programs produced in Borland Delphi environment, because it is based on Pascal and offers object mechanism which makes realisation of many considered data structures easier. These algorithms can be easily implemented in other programming languages, e.g. C, C++ and Java.

The book contains five chapters, the supplement about Delphi, a list of references, and a subject index which is helpful in getting the desired information. At the end of each chapter there are many exercises concerning thematically discussed issues. They should be solved independently by oneself.

Chapter 1 concentrates on the notion of algorithm and provides general introduction into design and analysis of computer algorithms. Exactly, it presents the basic properties and different ways of presentation of algorithms, contains intuitive and formal treatment of computational complexity and its asymptotic behaviour, shows the difference between recursion and iteration. The discussed problems and techniques are supported by examples such as Euclid's algorithm, bubble sorting, sequential and binary searching, calculating of Fibonacci numbers.

Chapter 2 is devoted to sort problem. It describes simple algorithms such as selection sort and insertion sort, also efficient algorithms such as quick sort and heap sort. All algorithms are coded in Object Pascal. Finally, they are compared in a practical way – by using in window application in Delphi. Moreover, the chapter presents external sorting algorithms and determining order statistics.

Chapter 3 focuses on fundamental data structures. It describes the organization, representations and maintenance of linear lists, stacks, single and double-ended queues, also priority queues, sets and dictionaries implemented by lists and hash tables. The discussed data structures are backed up with examples in Delphi.

Chapter 4 introduces to the wide subject of graphs. It provides the basic notions of graph theory and presents ways of representing graphs in computer. Next, it describes algorithms for searching graphs, determining of spanning trees

(depth-first search, breadth-first search), and finding connected components of an undirected graph. Also, it shows how to find the shortest paths to all vertices in the graph from given vertex (Dijkstra's algorithm) and the shortest paths between all pairs of vertices (Floyd-Warshall algorithm).

Chapter 5 reveals the power of recursion. It contains some examples of well-grounded recursion which leads to significantly improved notation of algorithms and makes their code more clear and simple. The presented example programs concern drawing attractive graphic motives (recursive tree, Sierpiński curves) and finding solution by trying one or several choices, i.e. backtracking algorithms for solving knapsack problem, eight queen problem, and problem of payment of given sum of money from the contents of purse. Finally, the chapter shows how to build simple interpreter of elementary programming language.

The supplement contains a quick look at the integrated development environment (IDE) Delphi and explains the most basic visual programming techniques based on components and their specific attributes – properties, events and methods. It has been thought of a group of those Readers, who do not know Delphi. Certainly, it will not duplicate the full textbook about Delphi but will enable them the reading of and understanding this book.

All example programs were prepared and tested using Borland Delphi 7. They can be found on the author's web site at [www.kaj.pr.radom.pl/mat.html](http://www.kaj.pr.radom.pl/mat.html). The Reader is allowed to download and use their source and run versions.