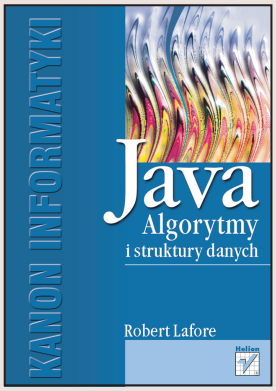
# Java. Algorytmy i struktury danych




Autor: Robert Lafore
Tłumaczenie: Przemysław Kowalczyk (rozdz. 1 – 5), Piotr Rajca (rozdz. 6 – 9), Paweł Koronkiewicz (rozdz. 9 – 15, dod. A – C)
ISBN: 83-7361-123-1
Tytuł oryginału: Data Structures & Algorithms in Java, 2nd Edition
Format: B5, stron: 704

Książka „Java. Algorytmy i struktury danych" jest łatwym do zrozumienia podręcznikiem poświęconym złożonym zagadnieniom gromadzenia i zarządzania danymi w taki sposób, aby uzyskać maksymalną efektywność działania programów komputerowych. Niezależnie od używanej platformy systemowej oraz języka programowania, opanowanie zagadnień przedstawionych w niniejszej książce poprawi jakość i efektywność tworzonego oprogramowania. Dzięki wykorzystaniu Javy do implementacji najważniejszych pojęć, uniknięto problemów związanych ze złożonością języków C oraz C++ i w pełni skoncentrowano się na prezentacji algorytmów i struktur danych.

Autor — Robert Lafore — prezentuje proste i zrozumiałe przykłady unikając niepotrzebnej matematyki i skomplikowanych dowodów, często pojawiających się w książkach o tej tematyce. W prezentowanym drugim wydaniu książki, autor udoskonalił i rozbudował przykłady, wykorzystując w nich najnowsze możliwości Javy. Na końcu każdego z rozdziałów zostały zamieszczone pytania i odpowiedzi, umożliwiające sprawdzenie stopnia zrozumienia i opanowania omawianych zagadnień.

W książce opisano:

- Tablice
- Proste i złożone algorytmy sortowania
- Stosy i kolejki
- Listy
- Zastosowania rekurencji
- Różne rodzaje drzew i sposoby ich implementacji
- Tablice rozproszone
- Sterty
- Grafy i grafy ważone
- Dobór właściwych algorytmów i struktur danych

**Robert Lafore** pisze o programowaniu od 1982 roku. Do jego najbardziej poczytnych książek należą: „Object-oriented Programming in C++", która została sprzedana w ponad 200 tysiącach egzemplarzy na całym świecie, „Assembly Language Programming for the IBM PC", „C Programming Using Turbo C++" oraz „C++ Interactive Course". Lafore posiada tytuły naukowe z matematyki i elektryki, a programowaniem zajmuje się aktywnie od czasów, gdy królowały komputery PDP-5 i gdy 4 kB

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

# Spis treści

## 2.

## Tablice ... 49

# 7.

## Zaawansowane algorytmy sortowania ... 297



# 8.

## Drzewa binarne ... 345

# 9.

## Drzewa czerwono-czarne ... 405



# 10.

## Drzewa 2-3-4 i pamięć zewnętrzna ... 437

# 13.



# 14.

# B



# C

# 6
# Rekurencja

W tym rozdziale:

- Liczby trójkątne.
- Silnia.
- Anagramy.
- Rekurencyjne wyszukiwanie binarne.
- Wieże Hanoi.
- Sortowanie przez scalanie.
- Eliminacja rekurencji.
- Niektóre interesujące zastosowania rekurencji.

Rekurencja jest techniką programistyczną, polegającą na wywoływaniu funkcji wewnątrz niej samej. Można sądzić, że postępowanie takie jest dziwne, a nawet, że jest ono katastrofalnym błędem. Niemniej jednak rekurencja jest jedną z najbardziej interesujących technik programistycznych i to w dodatku techniką zaskakująco efektywną. Z początku może się wydawać, że rekurencja jest czymś niesamowitym, czymś co można by porównać z ciągnięciem siebie samego za sznurowadła butów (masz sznurowadła, nieprawdaż?). Niemniej jednak rekurencja nie tylko działa, lecz co więcej, daje unikalną pojęciową podstawę do rozwiązywania wielu problemów.

W tym rozdziale zostaną przedstawione liczne przykłady pokazujące szeroką gammę sytuacji, w których można zastosować rekurencję. Między innymi zostanie zaprezentowany sposób obliczania liczb trójkątnych, silni, generacji anagramów, wyszukiwania binarnego, rozwiązywania problemu Wież Hanoi oraz sortowania przez scalanie. Przedstawione zostaną także aplety demonstrujące problem Wież Hanoi oraz sortowania przez scalanie.

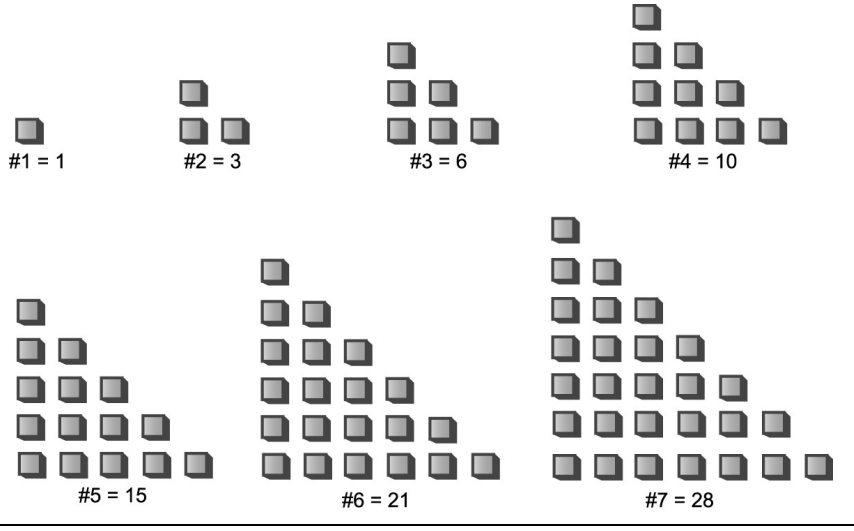
W rozdziale zostaną także omówione mocne i słabe strony rozwiązań rekurencyjnych oraz sposób przekształcania algorytmów rekurencyjnych na algorytmy wykorzystujące stos.

## Liczby trójkątne

Twierdzi się, że pitagorejczycy — grupa greckich matematyków pracujących pod przewodnictwem Pitagorasa (autor słynnego twierdzenia nazwanego jego nazwiskiem) — czuli mistyczny związek z ciągiem liczb: 1, 3, 6, 10, 15, 21, … (gdzie trzykropek oznacza, że ciąg jest nieskończony). Czy jesteś w stanie określić kolejną liczbę tego ciągu?

N-ty element ciągu jest uzyskiwany poprzez dodanie liczby n do poprzedniego elementu ciągu. A zatem, drugi element ciągu to 2 plus wartość pierwszego elementu (czyli 1), co daje 3. Trzeci element ciągu to 3 plus 3 (wartość drugiego elementu), co daje 6; i tak dalej.

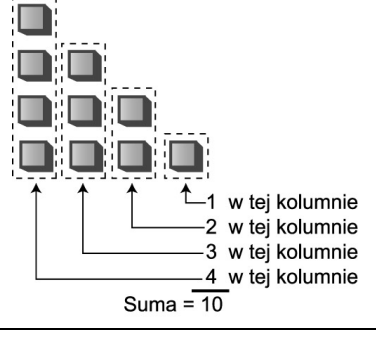
Liczby należące do tego ciągu są nazywane *liczbami trójkątnymi*, gdyż można je przedstawić za pomocą odpowiedniej ilości obiektów rozmieszczonych w formie trójkąta. Na rysunku 6.1 użyto w tym celu niewielkich trójkątów.




RYSUNEK 6.1. Liczby trójkątne

## Określanie wartości n-tego elementu przy użyciu pętli

Załóżmy, że chcemy znaleźć wartość pewnego (dowolnego) elementu ciągu — na przykład, czwartego elementu (którego wartość wynosi 10). W jaki sposób można ją obliczyć? Po przeanalizowaniu rysunku 6.2 można by dojść do wniosku, że można w tym celu zsumować wszystkie kwadraciki w poszczególnych kolumnach.




RYSUNEK 6.2. Liczba trójkątna przedstawiona w postaci kolumn

W czwartym elemencie ciągu pierwsza kolumna zawiera cztery kwadraciki, druga — trzy kwadraciki, i tak dalej. Dodając 4 + 3 + 2 + 1 uzyskujemy wartość 10.

Przedstawiona poniżej metoda `triangle()` oblicza liczby trójkątne, wykorzystując opisaną wcześniej metodę „kolumnową". Sumuje ona wszystkie kolumny, zaczynając od tej o wysokości *n* i kończąc na kolumnie o wysokości 1:

```
int triangle( int n ) {
   int total = 0;

   while(n > 0)             // aż do n = 1
      {
      total = total + n;    // dodajemy n (wysokość kolumny) do zmiennej total
      --n;                  // dekrementujemy wysokość kolumny
      }
   return total;
}
```

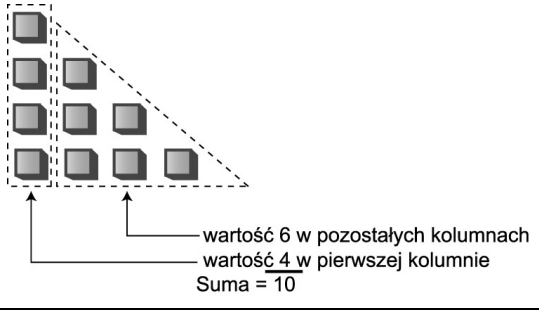
Metoda wykonuje zawartość pętli *n* razy; za pierwszym razem do zmiennej `total` dodawana jest wartość *n*, za drugim — *n* – 1 i tak dalej, aż do wartości 1. Pętla kończy się w momencie, gdy zmienna *n* osiągnie wartość 0.

## Określanie wartości n-tego elementu przy użyciu rekurencji

Metoda wykorzystująca pętlę może się wydawać bardzo prosta, jednak istnieje także inny sposób podejścia do zagadnienia. Otóż wartość *n*-tego elementu można potraktować jako sumę dwóch liczb, a nie sumę wszystkich elementów ciągu. Liczbami tymi są:

(1) Pierwsza (najwyższa) kolumna o wartości *n*.
(2) Suma wszystkich pozostałych kolumn.

Podejście takie przedstawia rysunek 6.3.




RYSUNEK 6.3. Liczba trójkątna jako suma kolumny i trójkąta

## Określanie wartości pozostałych kolumn

Znając sposób obliczania sumy pozostałych kolumn, można by zmodyfikować metodę `triangle()` zwracającą wartość *n*-tej liczby trójkątnej w następujący sposób:

```
int triangle(int n)
   {
   return( n + sumaPozostalychKolumn(n) );   // metoda niepełna
   }
```

Jednak jaka jest zaleta tego rozwiązania? Można sądzić, że napisanie metody `sumaPozostalychKolumn()` jest równie złożonym zadaniem jak napisanie metody `triangle()` przedstawionej w poprzedniej części rozdziału.

Patrząc na rysunek 6.3 można jednak zauważyć, że suma wszystkich pozostałych kolumn dla *n*-tego elementu ciągu jest taka sama jak wartość wszystkich kolumn elementu $n - 1$. A zatem, znając metodę sumującą wszystkie kolumny elementu *n*-tego, można by jej użyć do obliczenia wszystkich pozostałych kolumn tego elementu, przekazując w jej wywołaniu argument $n - 1$:

```
int triangle(int n)
   {
   return( n + sumaWszystkichKolumn(n - 1) );   // metoda niepełna
   }
```

Zastanawiając się nad powyższą metodą, można dojść do wniosku, że robi ona dokładnie to samo co metoda `triangle()`, czyli sumuje wszystkie kolumny dla pewnej wartości *n*, przekazanej jako argument jej wywołania. Czy zatem nie można by użyć samej metody `triangle()` zamiast wprowadzania kolejnej metody? Rozwiązanie takie miałoby następującą postać:

```
int triangle(int n)
   {
   return( n + triangle(n-1) );   // metoda niepełna
   }
```

Być może pomysł, aby metoda wywoływała samą siebie jest nieco szokujący, ale dlaczego miałoby to być niemożliwe? W końcu wywołanie metody jest (między innymi) przekazaniem sterowania na początek określonej metody. To przekazanie może być wykonane zarówno spoza metody jak i z jej wnętrza.

## Przekazywanie koszyka

Wszystkie te metody mogą nieco przypominać przekazywanie koszyka. Ktoś każe nam określić wartość dziewiątej liczby trójkątnej. Wiemy, że jej wartość to `9` plus wartość ósmej liczby trójkątnej, zatem wołamy Henia i prosimy go po określenie wartości ósmej liczby trójkątnej. Gdy Henio udzieli nam odpowiedzi, podaną przez niego wartość dodajemy do 9 i w ten sposób uzyskujemy poszukiwaną odpowiedź.

Henio wie, że ósma liczba trójkątna to 8 plus wartość siódmej liczby trójkątnej, dlatego woła Sabinkę i prosi ją o obliczenie siódmej liczby trójkątnej. Ten proces powtarza się, a każda kolejna osoba przekazuje koszyk następnej.

W którym momencie to przekazywanie koszyka kończy się? W pewnym momencie ktoś musi być w stanie podać odpowiedź bez konieczności proszenia o pomoc kolejnej osoby. Gdyby to się nie zdarzyło, powstałby nieskończony łańcuch osób, proszących kolejne osoby o pomoc — coś w stylu arytmetycznego schematu Ponziego, który nigdy by się nie zakończył. W przypadku metody `triangle()` oznaczałoby to, że metoda ta w wywoływałaby samą siebie w nieskończonej sekwencji, która doprowadziłaby do awarii programu.

## W tym miejscu kończy się przekazywanie koszyka

Aby zapobiec powstaniu nieskończonej sekwencji wywołań, osoba poproszona o podanie wartości pierwszej liczby trójkątnej (czyli liczby, dla której n przyjmuje wartość 1), musi widzieć że wartość ta wynosi 1, i to bez proszenia o pomoc kogokolwiek innego. W takim przypadku nie można już prosić kogoś o obliczenie mniejszej liczby trójkątnej, nie można już do niej dodać niczego innego, a zatem właśnie w tym momencie kończy się przekazywanie koszyka. Warunek ten można wyrazić poprzez dodanie do metody `triangle()` odpowiedniej instrukcji warunkowej:

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

Warunek, który sprawia, że metoda rekurencyjna zwraca wartość bez kolejnego wywołania, nazywany jest *przypadkiem bazowym*. Umieszczenie takiego warunku w każdej metodzie rekurencyjnej ma kluczowe znaczenie, gdyż tylko dzięki niemu metoda nie wywołuje samej siebie w nieskończoność, doprowadzając w końcu do awarii programu.

# Program triangle.java

Czy metody rekurencyjne sprawdzają się w praktyce? Uruchamiają program *triangle.java* można się przekonać, że tak. Po podaniu numeru obliczanego elementu (liczby *n*), program wyświetli wartość odpowiedniej liczby trójkątnej. Kod programu został przedstawiony na listingu 6.1.

LISTING 6.1.
Program triangle.java

```
// triangle.java
// Program wyznacza wartości liczb trójkątnych
// Aby uruchomić program program: C>java TriangleApp
import java.io.*;
////////////////////////////////////////////////////////////////
class TriangleApp
   {
   static int theNumber;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Podaj liczbe: ");
      theNumber = getInt();
      int theAnswer = triangle(theNumber);
      System.out.println("Wartosc liczby trojkatnej="+theAnswer);
      }  // koniec metody main()
```

```
//--------------------------------------------------------------
   public static int triangle(int n)
      {
      if(n==1)
         return 1;
      else
         return  ( n + triangle(n-1) );
      }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // koniec klasy TriangleApp
////////////////////////////////////////////////////////////////
```

Główna metoda programu — `main()` — prosi użytkownika o podanie wartości *n*, a następnie wyświetla wartość wynikową. Z kolei metoda `triangle()` wykonuje całe zadanie, wywołując rekurencyjnie sama siebie.

Oto przykładowe wyniki działania programu:

```
Podaj liczbe: 1000
Wartosc liczby trojkatnej = 500500
```

Jeśli nie jesteś przekonany co do poprawności działania metody `triangle()`, to tak się składa, że wartość liczby trójkątnej można także wyznaczyć przy użyciu poniższego wzoru:

wartość *n*-tej liczby trójkątnej = $(n^2 + n) / 2$

# Co się tak naprawdę dzieje?

Zmodyfikujmy nieco metodę `triangle()`, tak aby mieć wgląd w to co się dzieje podczas jej wykonywania. Do kodu oryginalnej metody wstawimy wywołania metody `println()`, które zapewnią nam dostęp do informacji o przekazywanych argumentach i zwracanych wartościach wynikowych:

```
public static int triangle(int n)
   {
   System.out.println("Dane wejsciowe: n=" + n);
```

```
   if(n==1)
      {
      System.out.println("Zwracana wartosc = 1");
      return 1;
      }
   else
      {
      int temp = n + triangle(n-1);
      System.out.println("Zwracana wartosc = " + temp);
      return temp;
      }
   }
```

Poniżej zostały przedstawione wyniki wygenerowane przez zmodyfikowaną metodę `triangle()`, gdy użytkownik wpisał wartość **5**:

```
Podaj liczbe: 5

Dane wejsciowe: n=5
Dane wejsciowe: n=4
Dane wejsciowe: n=3
Dane wejsciowe: n=2
Dane wejsciowe: n=1
Zwracana wartosc = 1
Zwracana wartosc = 3
Zwracana wartosc = 6
Zwracana wartosc = 10
Zwracana wartosc = 15
Wartosc liczby trojkatnej = 15
```

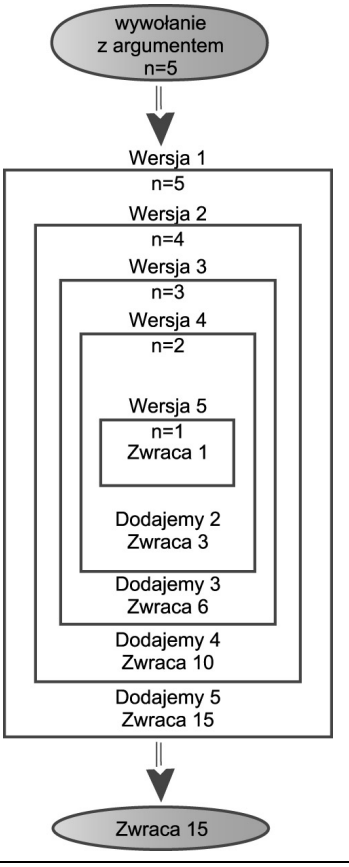
Za każdym razem gdy metoda `triangle()` wywołuje sama siebie, argument jej wywołania (który początkowo przyjmuje wartość 5) jest pomniejszany o 1. Kolejne wywołania metody są kontynuowane aż do momentu, gdy argument wywołania przyjmie wartość 1. W tym momencie metoda zwraca wartość wynikową. Doprowadza to do zwracania wartości i zakańczania kolejnych wywołań metody `triangle()` — metoda „cofa się" coraz wyżej w sekwencji wywołań. Za każdym razem wynik zwrócony przez metodę jest dodawany do wartości *n* przekazanej jako argument jej wywołania.

Wartości wynikowe podsumowują sekwencję liczb trójkątnych, która kończy się przekazaniem wartości wynikowej do metody `main()`. Rysunek 6.4 przedstawia, że każde wywołanie metody `triangle()` można sobie wyobrazić jako zdarzenie zachodzące „wewnątrz" wcześniejszego wywołania.

Należy zauważyć, że tuż przed momentem gdy najbardziej wewnętrzne wywołanie metody `triangle()` zwróci wartość 1, istnieje jednocześnie aż pięć „inkarnacji" (wywołań) tej metody. Do najbardziej zewnętrznego wywołania została przekazana wartość 5, a do najbardziej wewnętrznego — wartość 1.

## Charakterystyczne cechy metod rekurencyjnych

Choć metoda `triangle()` jest bardzo krótka, zawiera jednak wszystkie elementy charakterystyczne dla wszystkich procedur rekurencyjnych:




RYSUNEK 6.4. Rekurencyjna metoda triangle()

- Wywołuje sama siebie.
- Jeśli wywołuje sama siebie, to robi to w celu rozwiązania „mniejszego" problemu.
- Istnieje pewna wersja problemu, która jest na tyle prosta, że metoda może ją obliczyć bez konieczności ponownego wywoływania samej siebie.

W każdym kolejnym rekurencyjnym wywołaniu metody przekazywany do niej argument staje się mniejszy (lub, być może, mniejszym staje się zakres opisywany przez kilka przekazywanych argumentów); co odzwierciedla fakt, że rozwiązywany problem staje się „mniejszy" lub prostszy. Gdy argument lub zakres dojdzie do pewnej minimalnej wielkości, zostaje spełniony warunek, a metoda zwraca wartość bez kolejnego wywoływania samej siebie.

## Czy rekurencja jest efektywna?

Wywoływanie metody wiąże się z pewnymi narzutami. Konieczne jest przekazanie sterowania z miejsca w jakim znajduje się aktualnie na początek metody. Co więcej, należy zapisać na wewnętrznym stosie adres miejsca gdzie ma być przekazane sterowanie po zakończeniu metody oraz jej argumenty, tak aby metoda miała dostęp do przekazywanych wartości i wiedziała, gdzie zwrócić wartość wynikową.

W przypadku metody `triangle()` jest całkiem możliwe, iż narzuty te sprawiają, że rozwiązanie bazujące na wykorzystaniu pętli `while` jest bardziej efektywne od rozwiązania rekurencyjnego. Pogorszenie efektywności może nie być znaczące, niemniej jednak, jeśli rekurencyjne rozwiązanie problemu wymaga wykonania bardzo wielu wywołań metody, to celowe może się okazać skorzystanie z rozwiązania, które nie bazuje na rekurencji. Zagadnienia te zostaną dokładniej opisane na końcu rozdziału.

Kolejnym czynnikiem pogarszającym efektywność działania metod rekurencyjnych jest fakt, że wszystkie pośrednie argumenty i zwracane wartości są przechowywane na wewnętrznym stosie. W przypadku gdy ilości danych przechowywanych w ten sposób są bardzo duże, może to doprowadzić do przepełnienia stosu.

Rekurencja jest zazwyczaj używana nie ze względu na swoją efektywność, lecz dlatego, że jest w stanie pojęciowo uprościć rozwiązanie problemu.

## Indukcja matematyczna

Rekurencja jest programistycznym odpowiednikiem indukcji matematycznej. Indukcja matematyczna to sposób definiowania pewnego zagadnienia poprzez to zagadnienie (ten sam termin jest także używany w odniesieniu do podobnej metody udowadniania poprawności twierdzeń). Wykorzystując indukcję matematyczną można zdefiniować liczby trójkątne w następujący sposób:

$$\text{tri}(n) = 1 \quad \text{jeśli } n = 1$$
$$\text{tri}(n) = n + \text{tri}(n-1) \quad \text{jeśli } n > 1$$

Definiowanie pewnego zagadnienia przy użyciu jego samego może się bardzo wydawać dziwne, niemniej jednak sposób ten jest całkowicie poprawny (zakładając, że istnieje warunek bazowy).

# Silnia

Silnia jest pojęciowo zbliżona do liczb trójkątnych, z tą różnicą, iż zamiast dodawania jest używane mnożenie. Liczba trójkątna dla pewnej wartości *n* jest określana poprzez dodanie tej wartości do liczby trójkątnej odpowiadającej wartości *n* – 1. Z kolei wartość *n* silnia uzyskiwana jest poprzez pomnożenie *n* razy wartość *n* – 1 silnia. Oznacza to, że wartością piątej liczby trójkątnej jest 5 + 4 + 3 + 2 + 1, natomiast 5 silnia to 5 * 4 * 3 * 2 * 1, co daje 120. W tabeli 6.1 zostały przedstawione silnie liczb od 0 do 9.

Na mocy definicji wartość 1 silnia wynosi 1. Jak widać wartości silni rosną niezwykle szybko.

Silnie można obliczać przy użyciu metody rekurencyjnej przypominającej metodę `triangle()`. Oto jej kod:

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

TABELA 6.1.
Silnie

| Liczba | Obliczenie | Silnia |
|---|---|---|
| 0 | z definicji | 1 |
| 1 | 1 * 1 | 1 |
| 2 | 2 * 1 | 2 |
| 3 | 3 * 2 | 6 |
| 4 | 4 * 6 | 24 |
| 5 | 5 * 24 | 120 |
| 6 | 6 * 120 | 720 |
| 7 | 7 * 720 | 5 040 |
| 8 | 8 * 5 040 | 40 320 |
| 9 | 9 * 40 320 | 362 880 |

Istnieją jedynie dwie różnice pomiędzy metodami `factorial()` oraz `triangle()`. Po pierwsze w metodzie `factorial()` jest używany operator mnożenia (*), a nie dodawania (+):

```
n * factorial(n-1)
```

A po drugie, warunek bazowy zachodzi w przypadku gdy argument `n` ma wartość 0, a nie 1, jak to było w przypadku metody `triangle()`. Poniżej zostały przedstawione przykładowe wyniki wygenerowane przez program obliczający silnię, podobny do programu *triangle.java*:

```
Podaj liczbe: 6
Silnia = 720
```

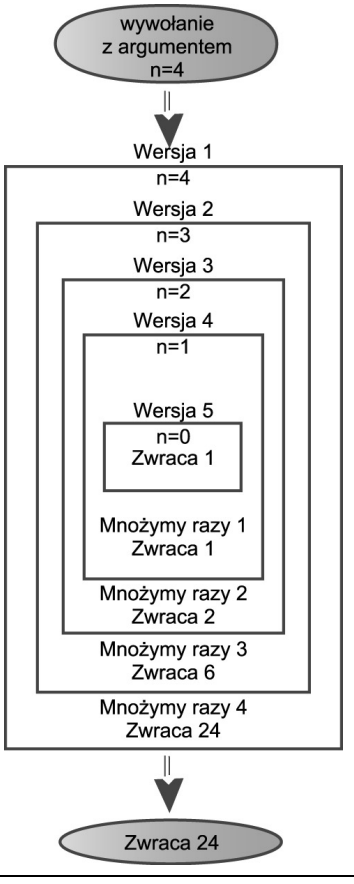
Rysunek 6.5 pokazuje, w jaki sposób kolejne „inkarnacje" metody `factorial()` wywołują same siebie, w przypadku gdy w pierwszym wywołaniu została przekazana wartość 4.

Obliczenie silni jest klasycznym sposobem prezentacji zagadnienia rekurencji. Niemniej jednak sposobu obliczania silni nie można przedstawić w sposób graficzny równie łatwo jak wyznaczania liczb trójkątnych.

W podobny, rekurencyjny sposób można rozwiązywać wiele innych problemów numerycznych, takich jak obliczanie największego wspólnego dzielnika dwóch liczb (służącego do uzyskania jak najmniejszego ułamka), podnoszenie liczby do potęgi, i tak dalej. Także w tych przypadkach są to jedynie ciekawe sposoby prezentacji algorytmów rekurencyjnych, które nie będą zapewne używane w praktyce, gdyż algorytmy wykorzystujące pętle są znacznie bardziej efektywne.

# Anagramy

W tej części rozdziału został przedstawiony kolejny problem, który w elegancki sposób można rozwiązać przy użyciu rekurencji. Permutacja jest rozmieszczeniem elementów w ściśle określonej kolejności. Załóżmy, że chcielibyśmy stworzyć listę anagramów pewnego słowa — czyli wszystkie permutacje liter tworzących dane słowo (niezależnie do tego, czy uzyskane w ten sposób ciągi liter tworzą sensowne słowa, czy też nie). Stworzenie listy anagramów słowa *kot* dałoby następujące wyniki:




RYSUNEK 6.5. Rekurencyjna metoda factorial()

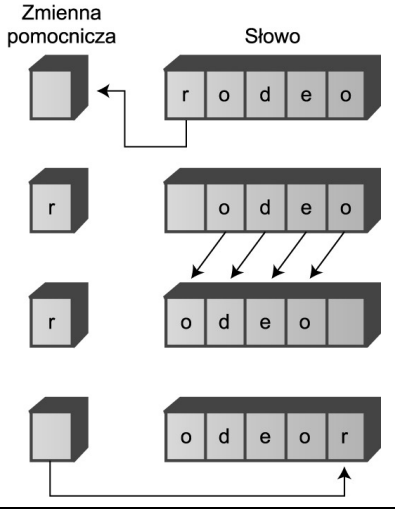
- kot,
- kto,
- otk,
- okt,
- tok,
- tko.

Warto spróbować samodzielnie ułożyć taką listę anagramów. Można się przekonać, że ilość dostępnych możliwości odpowiada silni z ilości liter tworzących słowo. Dla 3 liter istnieje 6 możliwych słów, dla 4 liter — 24 słowa, dla 5 liter — 120 słów, i tak dalej (zakładamy przy tym, że wszystkie litery słowa są unikalne; w przypadku powtarzania się liter ilość uzyskiwanych słów będzie mniejsza).

W jaki sposób można by napisać program tworzący anagramy słowa? Oto jedno z możliwych rozwiązań. Załóżmy, że słowo składa się z *n* liter.

(1) Utworzenie anagramów *n* – 1 liter słowa z pominięciem skrajnej lewej litery.
(2) Cykliczne przesunięcie wszystkich n liter.
(3) Powtórzenie powyższych czynności n razy.

*Cykliczne przesunięcie* słowa oznacza przesunięcie wszystkich jego liter o jedną pozycję w lewo, z wyjątkiem skrajnej, lewej litery, która jest zapisywana z prawej strony wyrazu, tak jak pokazano na rysunku 6.6.




RYSUNEK 6.6. Cykliczne przesuwanie słowa

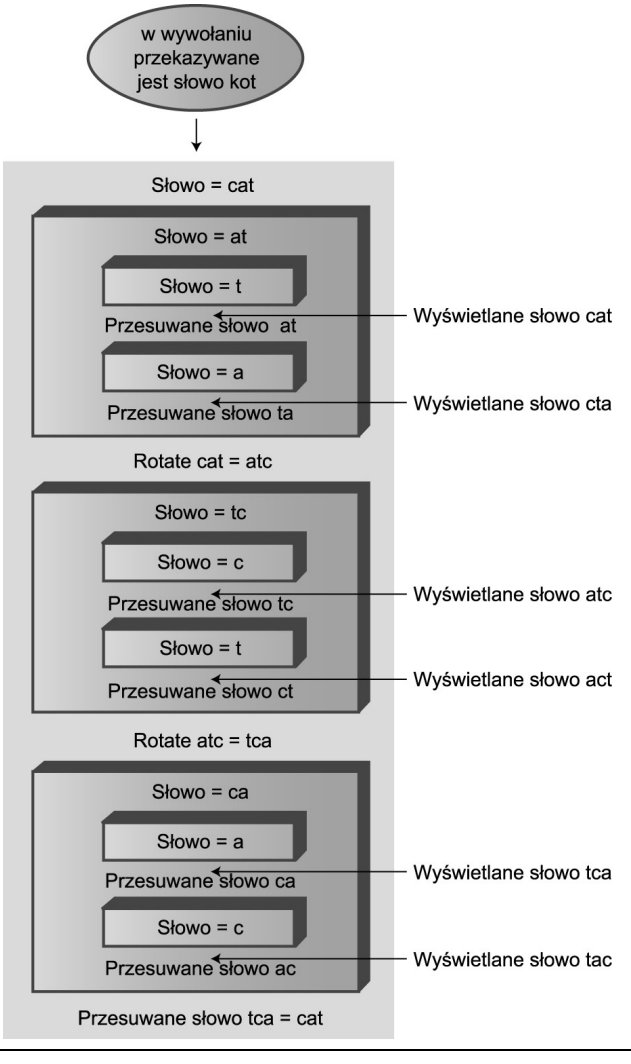
Cykliczne przesuwanie słowa n razy powoduje, że każda litera znajdzie się na jego początku. W momencie gdy dana litera słowa zajmuje miejsce na jego początku, tworzone są anagramy wszystkich pozostałych liter (czyli są one zapisywane na wszystkich możliwych pozycjach). Dla słowa *kot* składającego się wyłącznie z trzech liter, cykliczne przesunięcie dwóch pozostałych liter sprowadza się do zamienienia ich miejscami. Cała sekwencja tworzenia anagramów słowa *kot* została przedstawiona w tabeli 6.2.

TABELA 6.2.
Tworzenie anagramów słowa kot

| Słowo | Wyświetlać? | Pierwsza litera | Pozostałe litery | Operacja |
|---|---|---|---|---|
| kot | Tak | k | ot | Cyklicznie przesuwamy ot |
| kto | Tak | k | to | Cyklicznie przesuwamy to |
| kot | Nie | k | ot | Cyklicznie przesuwamy kot |
| otk | Tak | o | tk | Cyklicznie przesuwamy tk |
| okt | Tak | o | kt | Cyklicznie przesuwamy kt |
| otk | Nie | o | tk | Cyklicznie przesuwamy otk |
| tko | Tak | t | ko | Cyklicznie przesuwamy ko |
| tok | Tak | t | ok | Cyklicznie przesuwamy ok |
| tko | Nie | t | ko | Cyklicznie przesuwamy tko |
| kot | Nie | k | ot | Gotowe |

Należy zauważyć, że przed przesunięciem słowa trzyliterowego konieczne jest dodatkowe przesunięcie słowa dwuliterowego, które przywraca ich oryginalną kolejność. W ten sposób generowana jest sekwencja słów: kot, kto, kot. Powtarzające się słowa nie są wyświetlane.

A w jaki sposób można stworzyć anagramy *n* – 1 liter oryginalnego słowa? Wystarczy wywołać tę samą metodę. Rekurencyjna metoda `doAnagram()` pobiera tylko jeden argument — liczbę określającą wielkość słowa, dla którego należy utworzyć anagramy. Słowo to tworzy n liter oryginalnego słowa, zapisanych z jego prawej strony. Za każdym razem gdy metoda `doAnagram()` wywołuje sama siebie, przetwarzane słowo jest o jedną literę krótsze niż poprzednio, co pokazano na rysunku 6.7.

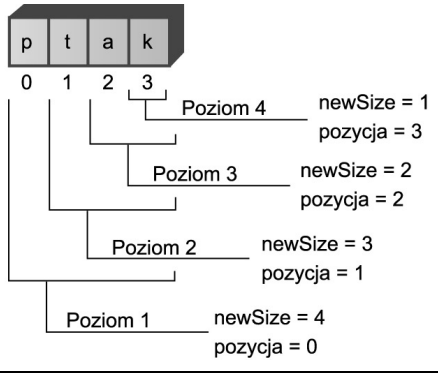



RYSUNEK 6.7. Rekurencyjna metoda doAnagram()

Warunek bazowy jest spełniony w momencie gdy słowo, dla którego metoda ma utworzyć listę anagramów, ma długość jednej litery. W takim przypadku wykonywanie metody kończy się, gdyż nie można zmienić kolejności w jakiej jest zapisana jedna litera. W każdym innym przypadku, metoda tworzy listę anagramów wszystkich liter przekazanego słowa, z wyjątkiem jego skrajnej lewej litery, a następnie cyklicznie przesuwa całe słowo. Te dwie czynności są wykonywane *n* razy, gdzie *n* jest długością słowa przekazanego w wywołaniu metody. Poniżej został przedstawiony kod rekurencyjnej metody `doAnagram()`:

```
public static void doAnagram(int newSize)
   {
   if(newSize == 1)                 // jeśli słowo zbyt małe
      return;                       // kończymy
   for(int j=0; j<newSize; j++)     // dla każdej pozycji
      {
      doAnagram(newSize-1);         // tworzymy anagramy pozostałych liter
      if (newSize==2)               // jeśli najbardziej zagłębione wywołanie
         displayWord();             // wyświetlamy słowo
      rotate(newSize);
      }
   }
```

Za każdym razem gdy metoda `doAnagram()` wywołuje sama siebie, słowo na jakim operuje jest mniejsze o jedną literę, a jego początek jest przesuwany o jedną literę w prawo. Proces ten został przedstawiony na rysunku 6.8.




RYSUNEK 6.8.
Coraz mniejsze wyrazy

Pełny kod programu anagram.java przedstawiono na listingu 6.2. Metoda `main()` pobiera słowo podane przez użytkownika, zapisuje je w tablicy znaków, tak aby można na nim operować w konwencjonalny sposób, a następnie wywołuje metodę `doAnagram()`.

LISTING 6.2.
Program anagram.java

```
// anagram.java
// Program generuje anagramy
// Aby uruchomić program: C>java AnagramApp
import java.io.*;
////////////////////////////////////////////////////////////////
class AnagramApp
   {
   static int size;
   static int count;
   static char[] arrChar = new char[100];
   //-------------------------------------------------------------
   public static void main(String[] args) throws IOException
      {
```

```
   System.out.print("Wpisz slowo: ");     // pobieramy słowo
   String input = getString();
   size = input.length();                 // określamy jego rozmiar
   count = 0;
   for(int j=0; j<size; j++)              // zapisujemy w tablicy
      arrChar[j] = input.charAt(j);
   doAnagram(size);                       // generujemy anagramy
   }  // koniec metody main()
//--------------------------------------------------------------
public static void doAnagram(int newSize)
   {
   int limit;
   if(newSize == 1)                       // jeśli słowo zbyt małe,
      return;                             // kończymy wykonywanie metody

   for(int j=0; j<newSize; j++)           // dla każdej pozycji,
      {
      doAnagram(newSize-1);               // anagramy pozostałych liter
      if(newSize==2)                      // jeśli najbardziej wewnętrzny,
         displayWord();                   // wyświetlamy słowo
      rotate(newSize);                    // przesuwamy cyklicznie słowo
      }
   }
//--------------------------------------------------------------
// metoda przesuwa o jedną pozycję w lewo wszystkie litery
// od podanej pozycji aż do końca słowa
public static void rotate(int newSize)
   {
   int j;
   int position = size - newSize;
   char temp = arrChar[position];         // zapamiętujemy pierwszą literę
   for(j=position+1; j<size; j++)         // przesuwamy pozostałe
      arrChar[j-1] = arrChar[j];
   arrChar[j-1] = temp;                   // zapisujemy pierwszą na końcu
   }
//--------------------------------------------------------------
public static void displayWord()
   {
   if(count < 99)
      System.out.print(" ");
   if(count < 9)
      System.out.print(" ");
   System.out.print(++count + " ");
   for(int j=0; j<size; j++)
      System.out.print( arrChar[j] );
   System.out.print("   ");
   System.out.flush();
   if(count%6 == 0)
      System.out.println("");
   }
```

```
   //--------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
   //--------------------------------------------------------
   }  // koniec klasy AnagramApp
//////////////////////////////////////////////////////////////
```

Metoda `rotate()` cyklicznie przesuwa słowo o jedną literę w lewo, w sposób opisany we wcześniejszej części rozdziału. Metoda `displayWord()` wyświetla słowo, dodając do niego liczbę ułatwiającą określenie ilości wygenerowanych słów. Poniżej zostały przedstawione przykładowe wyniki wygenerowane przez program:

```
Wpisz slowo: tran
  1 tran     2 trna     3 tanr     4 tarn     5 tnra     6 tnar
  7 rant     8 ratn     9 rnta    10 rnat    11 rtan    12 rtna
 13 antr    14 anrt    15 atrn    16 atnr    17 arnt    18 artn
 19 ntra    20 ntar    21 nrat    22 nrta    23 natr    24 nart
```

Przedstawionego programu można używać do generowania anagramów słów pięcio-, a nawet sześcioliterowych. Niemniej jednak, ponieważ silnia liczby 6 ma wartość 720, próba sporządzenia anagramów dla tak długiej sekwencji znaków może powodować wygenerowanie większej ilości słów, niż byśmy sobie tego życzyli.

# Rekurencyjne wyszukiwanie binarne

Czy Czytelnik przypomina sobie wyszukiwanie binarne omawiane w rozdziale 2., „Tablice”? Jego celem było odnalezienie konkretnej komórki w posortowanej tablicy przy użyciu minimalnej ilości porównań. Zastosowane rozwiązanie polegało na podzieleniu tablicy na dwie połówki, sprawdzeniu, w której z nich znajduje się poszukiwana komórka, podzieleniu tej połowy tabeli na połowę i tak dalej. Poniżej został przedstawiony kod oryginalnej metody `find()`:

```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;               // znaleziono
```

```
      else if(lowerBound > upperBound)
         return nElems;               // nie można znaleźć
      else                            // dzielimy zakres
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1;   // jest w górnej połówce
         else
            upperBound = curIn - 1;   // jest w dolnej połówce
         }  // koniec klauzuli else dzielącej zakres
      }  // koniec while
   }  // koniec funkcji find()
```

Być może warto przeczytać fragment dotyczący przeszukiwania binarnego w tablicach uporządkowanych zamieszczony w rozdziale 2., wyjaśniający zasadę działania powyższej metody. Można także uruchomić aplet *orderedArray.java* przedstawiony w rozdziale 2., aby przekonać się jak działa wyszukiwanie binarne.

Powyższy algorytm wykorzystujący pętlę stosunkowo łatwo można przekształcić na algorytm rekurencyjny. W metodzie bazującej na pętli najpierw modyfikowane są wartości zmiennych `lowerBound` lub `upperBound` określających przeszukiwany zakres, a następnie wykonywana jest pozostała część pętli. Każde wykonanie pętli powoduje podzielenie zakresu (mniej więcej) na połowę.

## Zastąpienie pętli rozwiązaniem rekurencyjnym

W rozwiązaniu rekurencyjnym, zamiast modyfikowania wartości zmiennych `lowerBound` lub `upperBound`, nowe wartości tych zmiennych są przekazywane w kolejnym wywołaniu metody `find()`. Pętla znika, a jej miejsce zajmuje rekurencyjne wywołanie. Oto jak wygląda nowa wersja metody:

```
private int recFind(long searchKey, int lowerBound,
                                    int upperBound)
   {
   int curIn;

   curIn = (lowerBound + upperBound ) / 2;
   if(a[curIn]==searchKey)
      return curIn;                // znaleziony
   else if(lowerBound > upperBound)
      return nElems;               // nie można znaleźć
   else                            // dzielimy zakres
      {
      if(a[curIn] < searchKey)     // w górnej połówce zakresu
         return recFind(searchKey, curIn+1, upperBound);
      else                         // w dolnej połówce zakresu
         return recFind(searchKey, lowerBound, curIn-1);
      }  // koniec klauzuli else dzielącej zakres
   }  // koniec metody recFind()
```

Użytkownik klasy reprezentowany przez metodę `main()` może nie wiedzieć, jaką wielkość ma przeszukiwana tablica w momencie wywoływania metody `find()`, a co więcej, pod żadnym pozorem nie powinien być zmuszany do samodzielnego określania początkowych wartości zmiennych

upperBound oraz lowerBound. Dlatego też została stworzona specjalna „pośrednia", publiczna metoda find(). Jest ona wywoływana przez metodę main(), a w jej wywołaniu należy przekazać tylko jeden argument — wartość poszukiwanego klucza. Metoda find() podaje poprawne początkowe wartości zmiennych loweBound oraz upperBound (którymi są odpowiednio: 0 oraz nElems-1), a następnie wywołuje prywatną rekurencyjną metodę recFind(). Poniżej przedstawiono kod metody find():

```
public int find(long searchKey)
   {
   return recFind(searchKey, 0, nElems-1);
   }
```

Pełny kod programu *binarySearch.java* został przedstawiony na listingu 6.3.

LISTING 6.3.
Program binarySearch.java

```
// binarySearch.java
// Prezentuje rekurencyjny alagorytm wyszukiwania binarnego
// Aby wykonać program: C>java BinarySearchApp
////////////////////////////////////////////////////////////////
class ordArray
   {
   private long[] a;                    // odwołanie do tablicy a
   private int nElems;                  // ilość elementów tablicy
   //-------------------------------------------------------------
   public ordArray(int max)             // konstruktor
      {
      a = new long[max];                // tworzymy tablicę
      nElems = 0;
      }
   //-------------------------------------------------------------
   public int size()
      { return nElems; }
   //-------------------------------------------------------------
   public int find(long searchKey)
      {
      return recFind(searchKey, 0, nElems-1);
      }
   //-------------------------------------------------------------
   private int recFind(long searchKey, int lowerBound,
                                       int upperBound)
      {
      int curIn;

      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;                  // znaleziony
      else if(lowerBound > upperBound)
         return nElems;                 // nie można znaleźć
      else                              // dzielimy zakres
         {
```

```
         if(a[curIn] < searchKey)       // w górnej połówce zakresu
            return recFind(searchKey, curIn+1, upperBound);
         else                           // w dolnej połówce zakresu
            return recFind(searchKey, lowerBound, curIn-1);
         }  // koniec klauzuli else dzielącej zakres
      }  // koniec metody recFind()
   //-------------------------------------------------------------
   public void insert(long value)       // zapis elementu w tablicy
      {
      int j;
      for(j=0; j<nElems; j++)           // określamy gdzie go zapisać
         if(a[j] > value)               // (wyszukiwanie liniowe)
            break;
      for(int k=nElems; k>j; k--)       // większe elementy przesuwamy w górę
         a[k] = a[k-1];
      a[j] = value;                     // wstawiamy element
      nElems++;                         // powiększamy wielkość tablicy
      }  // end insert()
   //-------------------------------------------------------------
   public void display()                // wyświetlenie zawartości tablicy
      {
      for(int j=0; j<nElems; j++)       // dla każdego elementu tablicy,
         System.out.print(a[j] + " ");  // wyświetlamy go
      System.out.println("");
      }
   //-------------------------------------------------------------
   }  // koniec klasy ordArray
////////////////////////////////////////////////////////////////
class BinarySearchApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;                // wielkość tablicy
      ordArray arr;                     // odwołanie do tablicy
      arr = new ordArray(maxSize);      // tworzymy tablicę

      arr.insert(72);                   // zapisujemy elementy
      arr.insert(90);
      arr.insert(45);
      arr.insert(126);
      arr.insert(54);
      arr.insert(99);
      arr.insert(144);
      arr.insert(27);
      arr.insert(135);
      arr.insert(81);
      arr.insert(18);
      arr.insert(108);
      arr.insert(9);
      arr.insert(117);
      arr.insert(63);
      arr.insert(36);
```

```
      arr.display();                  // wyświetlamy tablicę

      int searchKey = 27;             // szukamy elementu
      if( arr.find(searchKey) != arr.size() )
         System.out.println("Znaleziono " + searchKey);
      else
         System.out.println("Nie znaleziono " + searchKey);
      }  // koniec metody main()
   }  // koniec klasy BinarySearchApp
////////////////////////////////////////////////////////////////
```

Wewnątrz metody `main()` w tablicy zapisywanych jest 16 liczb. Metoda `insert()` zapisuje je w kolejności rosnącej. Po wstawieniu elementów zawartość całej tablicy jest wyświetlana. W końcu wywoływana jest metoda `find()`, podejmująca próbę odszukania wartości kluczowej — 27. Oto przykładowe wyniki wykonania programu:

```
9 18 27 36 45 54 63 72 81 90 99 108 117 126 135 144
Znaleziono 27
```

W programie *binarySearch.java* w tablicy zostało zapisanych 16 liczb. Na rysunku 6.9 pokazano, w jaki sposób metoda `recFind()` rekurencyjnie wywołuje samą siebie, za każdym razem zmniejszając przeszukiwany zakres tablicy. Gdy wewnętrzne wywołanie metody odnajdzie poszukiwany element, który w tym przypadku ma wartość 27, metoda kończy działanie i zwraca indeks odnalezionej wartości, który w naszym przypadku ma wartość 2 (o czym można się także przekonać, analizując wyświetloną zawartość tablicy). Wartość wynikowa jest następnie przekazywana jako wynik każdego wywołania metody `recFind()`, a w końcu metoda `find()` zwraca go użytkownikowi klasy.

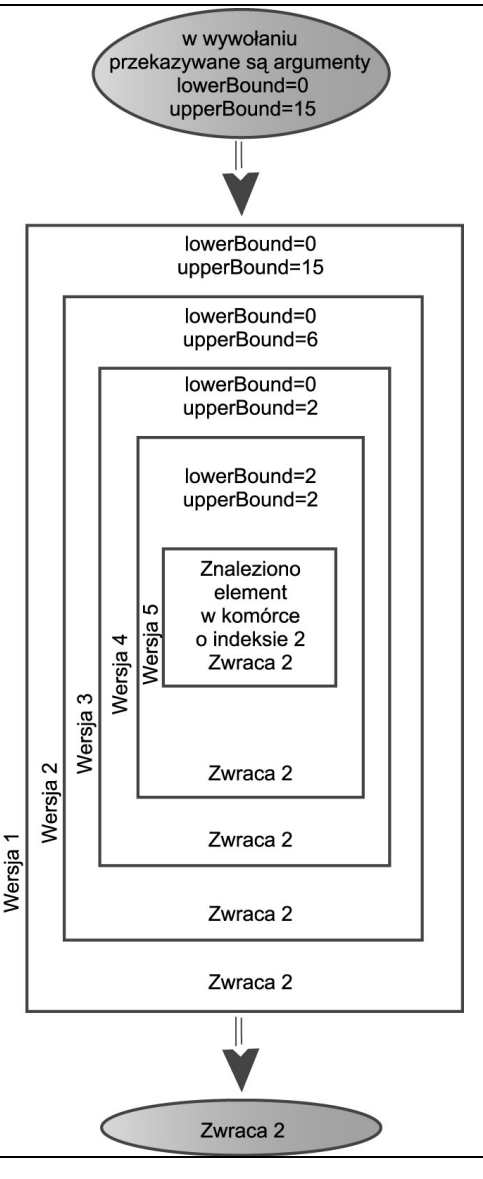
Rekurencyjny algorytm wyszukiwania binarnego ma tą samą złożoność działania co algorytm wykorzystujący pętlę — złożoność $O(\log N)$. Jest on nieco bardziej elegancki, choć jednocześnie nieco wolniejszy.

## Algorytmy „dziel i zwyciężaj”

Rekurencyjne wyszukiwanie binarne jest przykładem algorytmu typu *dziel i zwyciężaj*. Zasada działania algorytmów tego typu polega na podzieleniu dużego problemu na mniejsze, a następnie niezależnego rozwiązywania każdego z nich. Rozwiązanie każdego z mniejszych problemów jest takie samo — jest on dzielony na jeszcze mniejsze problemy, które następnie zostają rozwiązane niezależnie od siebie. Ten proces podziału na coraz to mniejsze problemy jest kontynuowany aż do momentu uzyskania przypadku bazowego, który można rozwiązać w prosty sposób bez dalszego dzielenia zagadnienia.

Metodologia „dziel i zwyciężaj” jest bardzo często używana w algorytmach rekurencyjnych, choć, jak można się było przekonać na przykładzie wyszukiwania binarnego przedstawionego w rozdziale 2., można ją także zastosować w algorytmach, które nie są rekurencyjne.

Metody działające zgodnie z zasadą „dziel i zwyciężaj” zawierają zazwyczaj dwa rekurencyjne wywołania samej siebie, z których każde obsługuje „połowę” problemu. W przypadku wyszukiwania binarnego występują dwa takie wywołania, lecz w rzeczywistości wykonywane jest tylko jedno z nich (to, które z nich zostanie wykonane, zależy od wartości klucza). Algorytm sortowania przez scalanie przedstawiony w dalszej części rozdziału wykonuje oba rekurencyjne wywołania (każde z nich sortuje połowę tablicy).

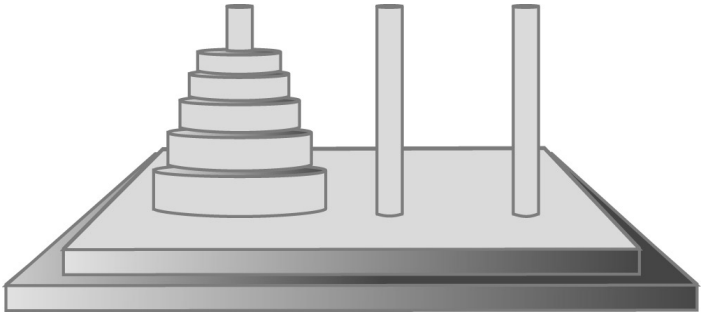



RYSUNEK 6.9. Rekurencyjna metoda binarySearch()

# Wieże Hanoi

Wieże Hanoi to starożytna zagadka składająca się z pewnej ilości dysków umieszczonych na trzech kolumnach, w sposób przedstawiony na rysunku 6.10.

Wszystkie dyski mają różne średnice oraz otwory na samym środku, dzięki czemu można je umieszczać na kolumnach. Początkowo wszystkie dyski są umieszczane na kolumnie A. Celem zagadki jest przeniesienie wszystkich dysków na kolumnę C. W każdym ruchu można przenosić tylko jeden dysk, a dodatkowo żaden dysk nie może być umieszczony na dysku o mniejszej średnicy.

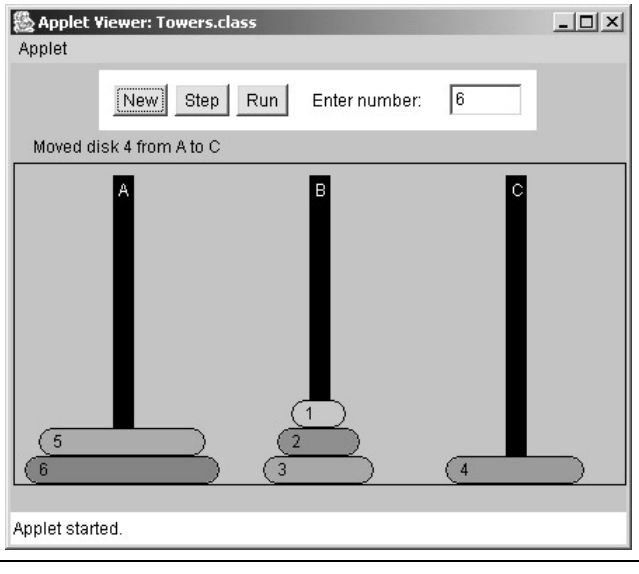


RYSUNEK 6.10.
Wieże Hanoi

Istnieje starożytna legenda twierdząca, że gdzieś w Indiach, w niedostępnej świątyni, mnisi pracują dzień i noc, aby przenieść 64 złote dyski z jednej wysadzanej diamentami kolumny na drugą. Gdy skończą, ma nastąpić koniec świata. Na szczęście, wszelkie obawy jakie można poczuć po poznaniu tej legendy znikną, gdy okaże się, jak wiele czasu zajmuje rozwiązanie tej zagadki, nawet dla znacznie mniejszej ilości dysków.

# Aplet Towers Workshop

Uruchom aplet Towers Workshop. Korzystając z niego, można samodzielnie podjąć próbę rozwiązania zagadki, przesuwając myszką najwyższy dysk i umieszczając go na innej kolumnie. Rysunek 6.11 przedstawia wygląd apletu po wykonaniu kilku przesunięć.




RYSUNEK 6.11.
Aplet demonstracyjny — Wieże Hanoi

Apletu można używać na trzy sposoby:

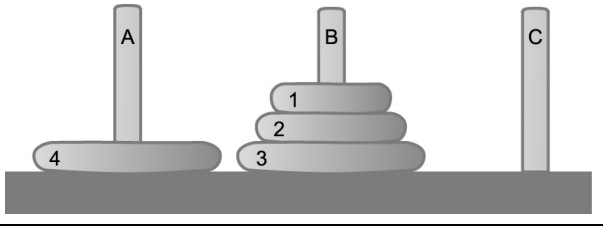
- Można spróbować samodzielnie rozwiązać problem, przeciągając dyski z jednej kolumny na drugą przy użyciu myszki.
- Można klikać przycisk *Step* (Krok), aby obserwować, jak algorytm rozwiązuje zagadkę. Podczas wykonywania każdego kroku w oknie apletu jest wyświetlany komunikat opisujący czynności wykonywane przez algorytm.
- Można także kliknąć przycisk *Run* (Uruchom), aby zobaczyć, jak algorytm rozwiązuje zagadkę bez żadnej interwencji ze strony użytkownika, przesuwając dyski na poszczególne kolumny.

Aby ponownie rozpocząć zagadkę, należy wpisać ilość dysków, jaka ma zostać użyta, a następnie dwukrotnie kliknąć przycisk *New* (Nowa) (pierwsze kliknięcie powoduje wyświetlenie pytania, czy na pewno należy ponownie rozpocząć zagadkę). Początkowo odpowiednio ułożone dyski zostaną umieszczone na kolumnie A. Gdy pierwszy dysk zostanie przeciągnięty myszką na inną kolumnę, przestanie być dostępna możliwość korzystania z przycisków *Step* oraz *Run*. Aby móc z nich skorzystać, konieczne będzie ponowne rozpoczęcie zagadki (z wykorzystaniem przycisku *New*). Niemniej jednak nic nie stoi na przeszkodzie, aby rozpocząć samodzielne rozwiązywanie zagadki podczas krokowego lub w pełni automatycznego działania algorytmu, bądź też, aby przejść do trybu w pełni automatycznego po wcześniejszym klikaniu przycisku *Step* lub do trybu krokowego podczas w pełni automatycznego rozwiązywania zagadki.

Warto podjąć próbę samodzielnego rozwiązania zagadki przy użyciu niewielkiej liczby dysków, na przykład trzech lub czterech. Potem można spróbować ją rozwiązać, gdy dysków będzie więcej. Aplet pozwala na intuicyjne poznanie sposobu rozwiązywania zagadki Wież Hanoi.

## Przesuwanie poddrzew

Załóżmy, że początkowe rozmieszczenie dysków na kolumnie A, przypominające kształtem drzewo lub piramidę, określimy mianem *drzewa*. (Takie „drzewa" nie mają nic wspólnego z drzewiastymi strukturami danych, opisywanymi w innych rozdziałach niniejszej książki). Bawiąc się apletem, można zauważyć, że podobne, lecz mniejsze drzewa są generowane jako fragmenty procesu rozwiązywania zagadki. Te mniejsze drzewa, zawierające jedynie część całości dysków używanych z zagadce, będziemy nazywać *poddrzewami*. Na przykład można zauważyć, że w przypadku zagadki wykorzystującej cztery dyski, w jednym z kroków pośrednich występuje poddrzewo składające się z 3 dysków umieszczonych na kolumnie B (przedstawione na rysunku 6.12).




RYSUNEK 6.12. Poddrzewo na kolumnie B

Takie poddrzewa są wielokrotnie tworzone podczas procesu rozwiązywania zagadki. Wynika to z faktu, że utworzenie poddrzewa jest jedynym sposobem przeniesienia większego dysku z jednej kolumny na drugą — wszystkie mniejsze dyski muszą być umieszczane na środkowej kolumnie, gdzie w naturalny sposób formują poddrzewo.

Oto reguła, która może pomóc w samodzielnym rozwiązywaniu zagadki. Jeśli poddrzewo, które chcemy przesunąć, składa się z nieparzystej liczby dysków, to dysk znajdujący się na jego wierzchołku można umieścić bezpośrednio na docelowej kolumnie. Jeśli natomiast przesuwane poddrzewo zawiera parzystą ilość dysków, dysk znajdujący się na jego wierzchołku należy przesunąć na środkową kolumnę.

## Algorytm rekurencyjny

Rozwiązanie problemu Wież Hanoi można wyrazić w sposób rekurencyjny, wykorzystując przy tym notację poddrzew. Załóżmy, że chcemy przenieść wszystkie dyski z kolumny źródłowej (nazwijmy ją *Z*) na kolumnę docelową (oznaczoną symbolem *D*). Dostępna jest kolumna pomocnicza (oznaczana symbolem *I*). Zakładamy także, że na kolumnie *Z* znajduje się *n* dysków. Oto algorytm:

(1) Przenieś poddrzewo składające się z $n - 1$ dysków z kolumny *Z* na kolumnę *I*.
(2) Przenieś ostatni (największy dysk) z kolumny *Z* na *D*.
(3) Przenieś poddrzewo z kolumny *I* na *D*.

Na samym początku kolumną źródłową jest kolumna *A*, kolumną pomocniczą — kolumna *B*, a docelową — kolumna *C*. Na rysunku 6.13 zostały przedstawione trzy kroki ilustrujące ten sposób rozwiązywania problemu Wież Hanoi.

W pierwszej kolejności poddrzewo składające się z dysków 1, 2 oraz 3 jest przesuwane na kolumnę *B*. Następnie największy dysk — 4 — jest przesuwany na kolumnę *C*. W ostatnim kroku całe poddrzewo jest przesuwane z kolumny *B* na *C*.

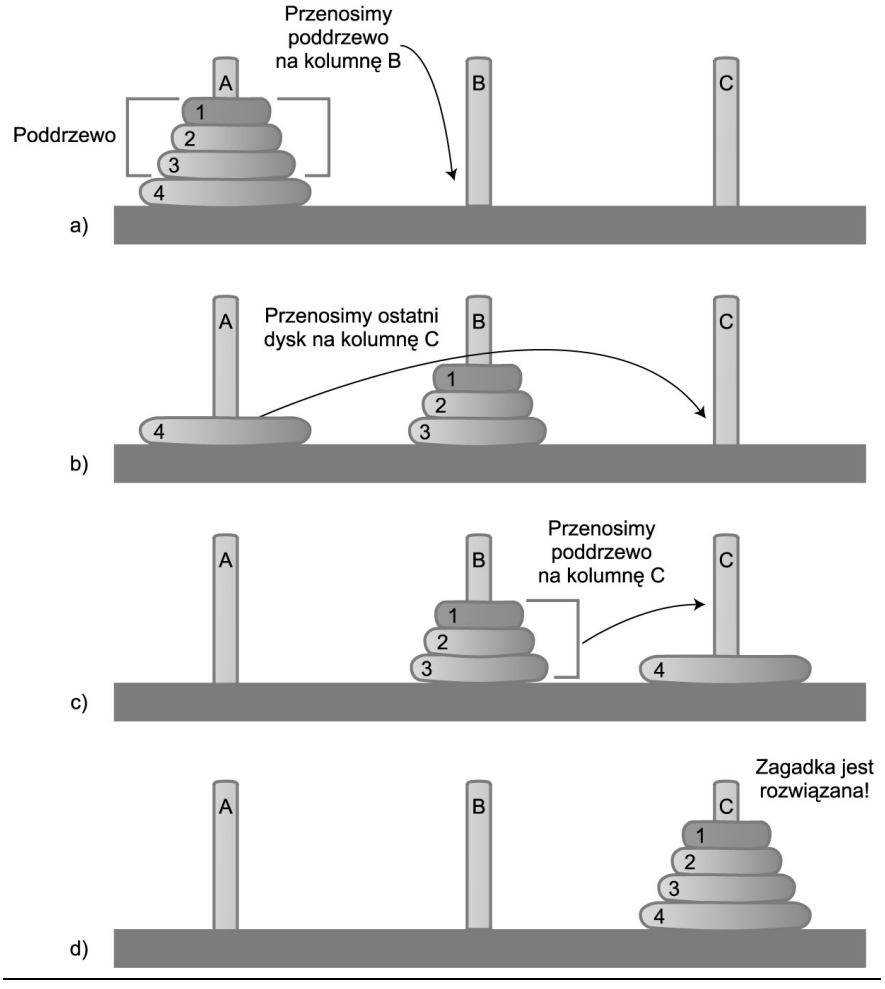
Oczywiście algorytm ten nie pokazuje, w jaki sposób można przenieść poddrzewo składające się z dysków 1, 2 oraz 3 na kolumnę *B*, gdyż w jednym ruchu nie można przenieść całego poddrzewa — jeden ruch umożliwia przeniesienie tylko jednego dysku. Przeniesienie poddrzewa składającego się z trzech dysków nie jest zadaniem prostym. Niemniej jednak jest prostsze niż przeniesienie drzewa składającego się z czterech dysków.

Jak się okazuje, przesunięcie trzech dysków z kolumny *A* na docelową kolumnę *B* także można zrealizować poprzez wykonanie tych samych trzech czynności, koniecznych do przeniesienia czterech dysków. Te czynności to: przeniesienie poddrzewa składającego się z dwóch pierwszych dysków z kolumny *A* na pomocniczą kolumnę *C*, przeniesienie trzeciego dysku z kolumny *A* na docelową kolumnę *B* i w końcu przeniesienie poddrzewa z kolumny *C* na *B*.

A w jaki sposób można przenieść poddrzewo składające się z dwóch dysków z kolumny *A* na *C*? Otóż należy przenieść poddrzewo składające się z jednego dysku (1) z kolumny *A* na *B*. To jest przypadek bazowy: przesuwając jeden dysk, można go po prostu zdjąć z jednej kolumny i umieścić na drugiej — nie wymaga to żadnych dodatkowych czynności. Następnie pozostaje przeniesienie większego dysku (2) z kolumny *A* na *C* i umieszczenie na nim poddrzewa (czyli dysku 1).

## Program towers.java

Program *towers.java* rozwiązuje zagadkę Wież Hanoi przy użyciu algorytmu rekurencyjnego. Program przedstawia wykonywane czynności, wyświetlając stosowne opisy; takie rozwiązanie znacznie upraszcza kod programu, który w przypadku konieczności wyświetlania kolumn i dysków byłby znacznie bardziej skomplikowany. Zadaniem użytkownika programu jest odczytanie listy operacji i samodzielne wykonanie odpowiednich czynności.




RYSUNEK 6.13. Rekurencyjne rozwiązanie problemu Wież Hanoi

Sam kod programu jest bardzo prosty. Metoda `main()` zawiera pojedyncze wywołanie metody `doTowers()`. Z kolei metoda `doTowers()` rekurencyjnie wywołuje sama siebie aż do momentu rozwiązania całego problemu. W tej wersji programu, przedstawionej na listingu 6.4, wykorzystywane są jedynie trzy dyski; choć oczywiście można ponownie skompilować program, tak aby liczba wykorzystywanych dysków była inna.

LISTING 6.4.
Program towers.java

```
// towers.java
// Program rozwiązuje zagadkę Wież Hanoi
// Aby wykonać program: C>java TowersApp
////////////////////////////////////////////////////////////////
class TowersApp
   {
   static int nDisks = 3;
```

```
   public static void main(String[] args)
      {
      doTowers(nDisks, 'A', 'B', 'C');
      }
   //--------------------------------------------------------
   public static void doTowers(int topN,
                               char src, char inter, char dest)
      {
      if(topN==1)
         System.out.println("Dysk 1 z " + src + " na "+ dest);
      else
         {
         doTowers(topN-1, src, dest, inter);   // z kol. źródłowej na pomocniczą

         System.out.println("Dysk " + topN +   // przesuwamy
                            " z " + src + " na "+ dest);
         doTowers(topN-1, inter, src, dest);   // wstawiamy dysk
         }
      }
//--------------------------------------------------------------
   }  // koniec klasy TowersApp
////////////////////////////////////////////////////////////////
```

Należy pamiętać, że trzy dyski są przenoszone z kolumny *A* na kolumnę *C*. Poniżej zostały przedstawione wyniki wykonania programu:

```
Dysk 1 z A na C
Dysk 2 z A na B
Dysk 1 z C na B
Dysk 3 z A na C
Dysk 1 z B na A
Dysk 2 z B na C
Dysk 1 z A na C
```

Argumentami wywołania metody `doTowers()` są: ilość przenoszonych dysków, kolumna źródłowa (`from`), pomocnicza (`inter`) oraz docelowa (`dest`). Za każdym razem gdy metoda wywołuje sama siebie, ilość dysków jest zmniejszana o jeden. Zmieniane są także kolumny — źródłowa, pomocnicza i docelowa.

Poniżej zostały przedstawione wyniki wykonania programu zawierające dodatkowe informacje, pokazujące, w którym miejscu rozpoczyna się wykonywanie metody, jakie są jej argumenty oraz czy dysk jest przesuwany, ponieważ został spełniony warunek bazowy (poddrzewo zawiera tylko jeden dysk), czy też dlatego, że jest to największy dysk pozostały po przeniesieniu poddrzewa:

```
Poczatek (3 dyski): z=A, i=B, d=C
   Poczatek (2 dyski): z=A, i=C, d=B
      Poczatek (1 dysk): z=A, i=B, d=C
         Przypadek bazowy: przesuwamy dysk 1 z A na C
      Koniec (1 dysk)
      Przesuwamy dolny dysk 2 z A na B
      Poczatek (1 dysk): z=C, i=A, d=B
         Przypadek bazowy: przesuwamy dysk 1 z C na B
      Koniec (1 dysk)
```

```
   Koniec (2 dyski)
   Przesuwamy dolny dysk 3 z A na C
   Poczatek (2 dyski): z=B, i=A, d=C
      Poczatek (1 dysk): z=B, i=C, d=A
         Przypadek bazowy: przesuwamy dysk 1 z B na A
      Koniec (1 dysk)
      Przesuwamy dolny dysk 2 z B na C
      Poczatek (1 dysk): z=A, i=B, d=C
         Przypadek bazowy: przesuwamy dysk 1 z A na C
      Koniec (1 dysk)
   Koniec (2 dyski)
Koniec (3 dyski)
```

Przestudiowanie powyższych wyników oraz kodu źródłowego metody `doTowers()` powinno umożliwić zrozumienie zasady działania tej metody. To zadziwiające, że tak prosty fragment kodu jest w stanie rozwiązać pozornie tak bardzo złożony problem.

# Sortowanie przez scalanie

Ostatnim przykładem rekurencji przedstawionym w tym rozdziale będzie algorytm sortowania przez scalanie. Jest to algorytm sortujący, którego efektywność jest znacznie wyższa od efektywności rozwiązania przedstawionego w rozdziale 3., „Proste algorytmy sortowania", przynajmniej pod względem szybkości działania. Algorytm „bąbelkowy" oraz algorytmy bazujące na wstawianiu oraz selekcji mają złożoność rzędu $O(N^2)$, natomiast algorytm sortowania przez scalanie — złożoność $O(N * \log N)$. Wykres przedstawiony na rysunku 2.9 (w rozdziale 2.) pokazuje, jak ogromny jest to wzrost efektywności i szybkości działania. Na przykład, jeśli N (ilość sortowanych elementów) wynosi 10 000, to $N^2$ wynosi 100 000 000, natomiast $N * \log N$ — jedynie 40 000. A zatem zakładając, że posortowanie takiej ilości elementów przy użyciu algorytmu mergestort trwa 40 sekund, to w przypadku algorytmu wstawiania sortowanie zajęłoby 28 godzin.

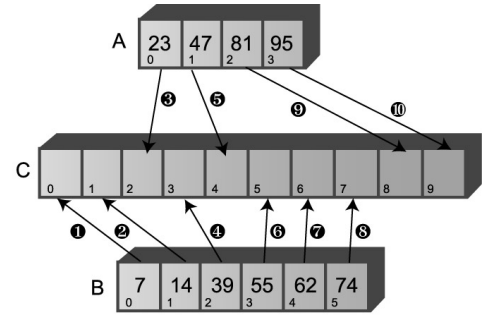
Algorytm sortowania przez scalanie jest stosunkowo łatwy do zaimplementowania. Pojęciowo jest on znacznie łatwiejszy od algorytmów quicksort oraz sortowania Shella, które zostaną omówione w następnym rozdziale.

Wadą tego algorytmu jest konieczność utworzenia w pamięci dodatkowej tablicy o takiej samej wielkości jak sortowana tablica. A zatem, jeśli sortowana tablica zajmuje prawie całą dostępną pamięć komputera, to algorytm ten nie będzie nadawał się do wykorzystania. Jeśli jednak wolnej pamięci jest dużo, to jego zastosowanie będzie dobrym pomysłem.

## Scalanie dwóch posortowanych tablic

Najważniejszą częścią algorytmu sortowania przez scalanie jest scalenie dwóch posortowanych tablic. W wyniku scalenia dwóch posortowanych tablic A i B powstaje trzecia tablica — C, która zawiera posortowane elementy z tablic A i B. W pierwszej kolejności zostanie przedstawiony właśnie proces scalania, a dopiero potem sposób wykorzystania tego procesu w sortowaniu.

Wyobraźmy sobie dwie posortowane tablice. Tablice te nie muszą być tej samej wielkości. Załóżmy przykładowo, że tablica A zawiera 4 elementy, a tablica B 6 elementów. W wyniku scalenia zawartość tych tablic zostanie zapisana w tablicy C, która początkowo składa się z 10 pustych komórek. Tablice te zostały przedstawione na rysunku 6.14.




RYSUNEK 6.14.
Scalanie dwóch tablic

Cyfry umieszczone w kółeczkach widoczne na rysunku 6.14 określają kolejność, w jakiej elementy tablic A i B są zapisywane w tablicy C. W tabeli 6.3 zostały przedstawione porównania, które należy wykonać, aby określić, jakie elementy mają być skopiowane. Kroki zamieszczone w tabeli odpowiadają czynnościom przedstawionym na rysunku. Po każdym porównaniu mniejszy z elementów jest zapisywany w tablicy C.

TABELA 6.3.
Czynności związane ze scalaniem tablic

| Krok | Porównanie (jeśli jest) | Kopiowanie |
|---|---|---|
| 1 | Porównujemy liczby 23 oraz 7 | Kopiujemy liczbę 7 z tablicy B do C |
| 2 | Porównujemy liczby 23 oraz 14 | Kopiujemy liczbę 14 z tablicy B do C |
| 3 | Porównujemy liczby 23 oraz 39 | Kopiujemy liczbę 23 z tablicy A do C |
| 4 | Porównujemy liczby 39 oraz 47 | Kopiujemy liczbę 39 z tablicy B do C |
| 5 | Porównujemy liczby 55 oraz 47 | Kopiujemy liczbę 47 z tablicy A do C |
| 6 | Porównujemy liczby 55 oraz 81 | Kopiujemy liczbę 55 z tablicy B do C |
| 7 | Porównujemy liczby 62 oraz 81 | Kopiujemy liczbę 62 z tablicy B do C |
| 8 | Porównujemy liczby 74 oraz 81 | Kopiujemy liczbę 74 z tablicy B do C |
| 9 | | Kopiujemy liczbę 81 z tablicy A do C |
| 10 | | Kopiujemy liczbę 95 z tablicy A do C |

Warto zwrócić uwagę, iż po wykonaniu kroku 8. tablica B jest pusta, a zatem nie ma potrzeby wykonywania jakichkolwiek dodatkowych porównań; wszystkie pozostałe elementy tablicy A można po prostu skopiować do tablicy C.

Listing 6.5 przedstawia napisany w Javie program realizujący operację scalenia zaprezentowaną na rysunku 6.14 oraz w tabeli 6.3. Nie jest to program rekurencyjny — jest on jedynie wstępem, który pozwoli zrozumieć działanie algorytmu sortowania przez scalanie.

LISTING 6.5.
Program merge.java

```
// merge.java
// Program demonstrujący scalanie dwóch tabel i zapis ich
// zawartości w trzeciej.
// Aby uruchomić program: C>java MergeApp
////////////////////////////////////////////////////////////////
class MergeApp
   {
   public static void main(String[] args)
      {
      int[] arrayA = {23, 47, 81, 95};
      int[] arrayB = {7, 14, 39, 55, 62, 74};
      int[] arrayC = new int[10];

      merge(arrayA, 4, arrayB, 6, arrayC);
      display(arrayC, 10);
      }  // koniec metody main()
   //--------------------------------------------------------------
   // łączymy tablice A i B zapisując ich zawartość w tablicy C
     public static void merge( int[] arrayA, int sizeA,
                               int[] arrayB, int sizeB,
                               int[] arrayC )
      {
      int aDex=0, bDex=0, cDex=0;

      while(aDex < sizeA && bDex < sizeB)    // żadna tablica nie jest pusta
         if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
         else
            arrayC[cDex++] = arrayB[bDex++];

      while(aDex < sizeA)                    // tablica B (arrayB) jest pusta,
         arrayC[cDex++] = arrayA[aDex++];    // ale tablica A (arrayA) nie jest

      while(bDex < sizeB)                    // tablica A (arrayA) jest pusta,
         arrayC[cDex++] = arrayB[bDex++];    // ale tablica B (arrayB) nie jest
      }  // end merge()
   //--------------------------------------------------------------
                                             // wyświetlenie tablicy
   public static void display(int[] theArray, int size)
      {
      for(int j=0; j<size; j++)
         System.out.print(theArray[j] + " ");
      System.out.println("");
      }
```

```
   //--------------------------------------------------------
   }  // koniec klasy MergeApp
////////////////////////////////////////////////////////////
```

W metodzie `main()` tworzone są trzy tablice — `arrayA`, `arrayB` oraz `arrayC`, następnie wywoływana jest metoda `merge()` scalająca tablice `arrayA` i `arrayB` i zapisująca ich zawartość w tablicy `arrayC`, a w końcu zawartość tablicy `arrayC` jest wyświetlana. Oto wyniki wykonania programu:

```
7 14 23 39 47 55 62 74 81 95
```

Wewnątrz metody `merge()` umieszczone są trzy pętle `while`. Pierwsza z nich analizuje zawartość tablicy `arrayA` oraz `arrayB`, porównując ich elementy i zapisując mniejszy z nich do tablicy `arrayC`.

Druga pętla obsługuje sytuację, w której cała zawartość tablicy `arrayB` została zapisana w `arrayC`, natomiast w tablicy `arrayA` wciąż pozostały elementy do przeniesienia (właśnie tak dzieje się w naszym przykładzie, w którym w tabeli `arrayA` pozostają liczby 81 i 95). Pętla ta kopiuje pozostałe elementy tablicy `arrayA` i zapisuje je w tablicy `arrayC`.

Trzecia pętla while obsługuje podobną sytuację, gdy wszystkie elementy z tablicy `arrayA` zostały zapisane w `arrayC`, lecz w tablicy `arrayB` wciąż pozostają elementy, które należy przenieść. Pętla kopiuje te elementy do tablicy `arrayC`.

## Sortowanie przez scalanie

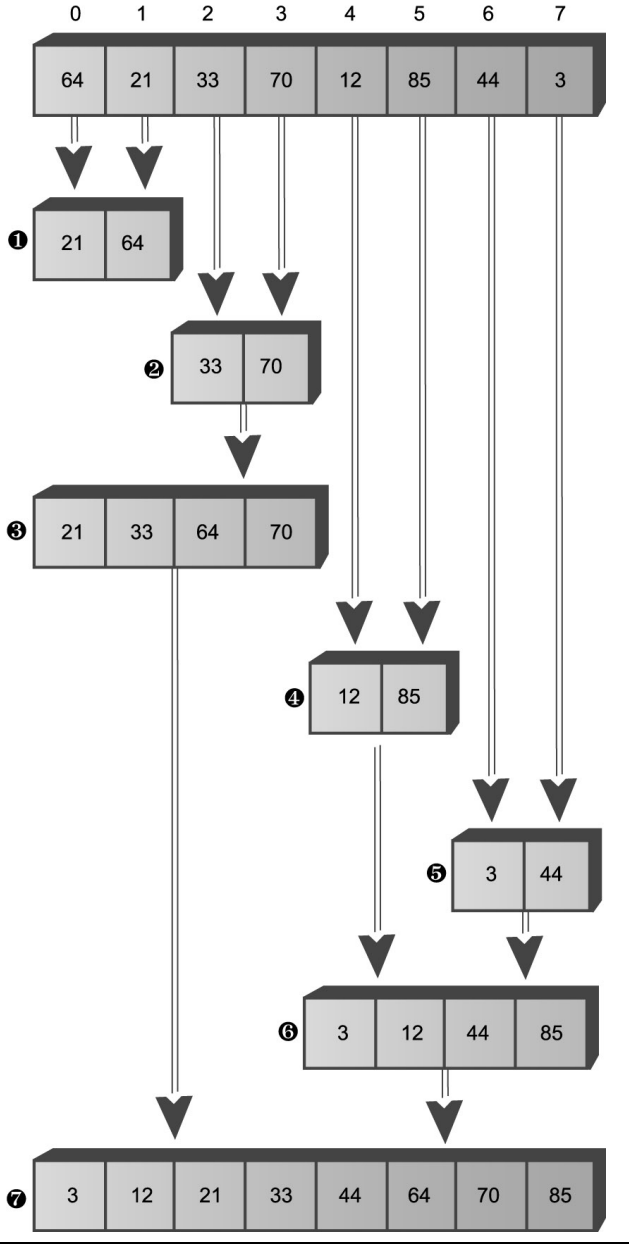
Idea działania algorytmu sortowania przez scalanie polega na podzieleniu tablicy na dwie części, posortowaniu każdej z nich, a następnie scaleniu obu posortowanych połówek z wykorzystaniem metody `merge()` z powrotem w jedną tablicę. A w jaki sposób są sortowane te połówki oryginalnej tablicy? Niniejszy rozdział przedstawia zagadnienia rekurencji, a zatem Czytelnik zapewne już zna odpowiedź — te połówki dzieli się na połowy, które następnie są sortowane i ponownie scalane.

W podobny sposób wszystkie pary ósmych części tablicy są scalane, tworząc posortowane ćwiartki, wszystkie pary szesnastych części tablicy są scalane, tworząc posortowane ósme części tablicy, i tak dalej. Tablica jest dzielona, aż do momentu uzyskania poddrzewa zawierającego tylko jeden element. To jest przypadek bazowy — zakładamy bowiem, że tablica zawierająca jeden element jest posortowana.

Jak można się było przekonać, każde rekurencyjne wywołanie metody powoduje zmniejszenie danych, na jakich operujemy, jednak przed zakończeniem wywołania dane te są ponownie łączone. W przypadku algorytmu sortowania przez scalanie każde rekurencyjne wywołanie metody powoduje podzielenie zakresu danych, na jakich operujemy na połowę i ponowne scalenie mniejszych zakresów przez zakończeniem wywołania.

Gdy metoda `mergeSort()` kończy działanie po dotarciu do dwóch jednoelementowych tablic, elementy te są scalane do postaci posortowanej tablicy dwuelementowej. Wszystkie pary tablicy dwuelementowych są następnie scalane do postaci tablicy czteroelementowych. Proces ten jest kontynuowany, aż do momentu gdy cała tablica zostanie posortowana. Sposób działania algorytmu najłatwiej można przedstawić, w przypadku gdy ilość elementów w początkowej tablicy jest potęgą liczby 2, jak pokazano na rysunku 6.15.

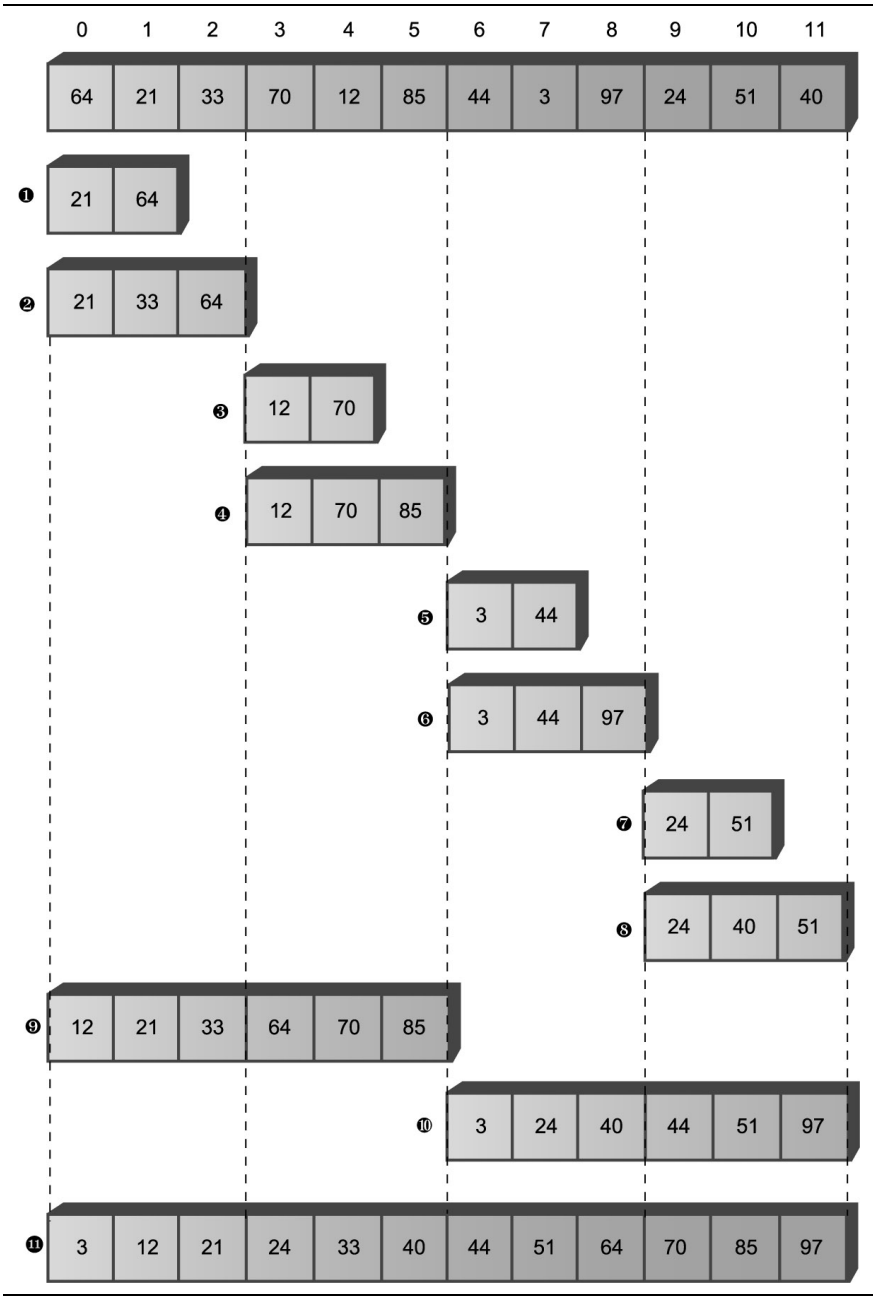
W pierwszej kolejności na samym dole tablicy zakresy 0-0 oraz 0-1 są scalane do zakresu 0-1. Oczywiście w rzeczywistości zakresy 0-0 oraz 1-1 nie są prawdziwymi zakresami — są to pojedyncze elementy, spełniające warunek bazowy. Podobnie zakresy 2-2 oraz 3-3 są scalane do zakresu 2-3. Następnie zakresy 0-1 oraz 2-3 są scalane do zakresu 0-3.




RYSUNEK 6.15. Scalanie coraz większych tablic

W górnej połówce tablicy zakresy 4-4 i 5-5 są scalane do zakresu 4-5, zakresy 6-6 i 7- 7 do zakresu 6-7, a zakresy 4-5 i 6-7 do zakresu 4-7. W końcu dolna połówka tablicy (zakres 0-3) oraz jej górna połówka (zakres 4-7) są łączone w pełną, posortowaną tablicę (zakres 0-7).

Jeśli natomiast ilość elementów tablicy nie jest potęgą liczby 2, zachodzi konieczność scalenia tablic o różnych wielkościach. Na przykład rysunek 6.16 przedstawia sytuację, w której sortowana tablica zawiera 12 elementów. W tym przypadku należy scalić tablice zawierającą 2 elementy z tablicą zawierającą 1 element, uzyskując w ten sposób tablicę trójelementową.




RYSUNEK 6.16. Scalanie tablic, których wielkość nie jest potęgą liczby 2

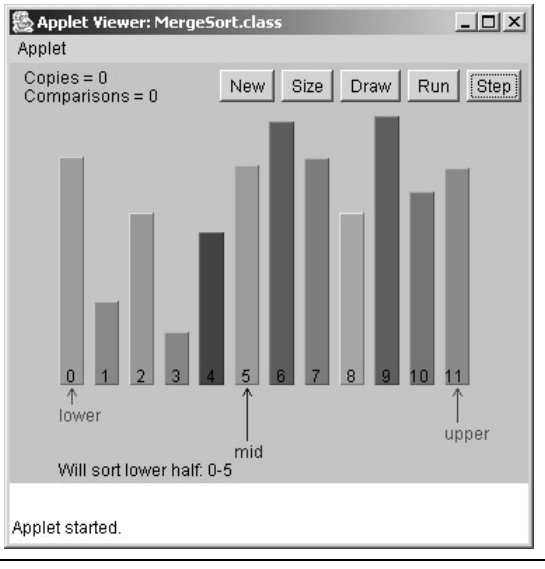
W pierwszej kolejności jednoelementowy zakres 0-0 jest scalany z jednoelementowym zakresem 1-1, tworząc w ten sposób dwuelementowy zakres 0-1. Następnie zakres 0-1 jest scalany z jednoelementowym zakresem 2-2. W ten sposób tworzony jest trójelementowy zakres 0-2. Ten zakres jest z kolei scalany z trójelementowym zakresem 3-5. Proces ten trwa aż do posortowania całej tablicy.

Należy zauważyć, że algorytm sortowania przez scalanie nie scala dwóch niezależnych tablic, zapisując ich elementy w trzeciej, jak to było robione w programie *merge.java*. Zamiast tego scalane są dwa fragmenty tej samej tablicy, w której także są zapisywane wyniki scalenia.

Można się zastanawiać, gdzie są przechowywane wszystkie te „podtablice". Algorytm tworzy w tym celu tablicę roboczą, której wielkość odpowiada wielkości oryginalnej sortowanej tablicy. Wszystkie podtablice są przechowywane w odpowiednich miejscach tej tablicy roboczej. Oznacza to, że podtablice wyznaczone w oryginalnej tablicy są kopiowane w odpowiednie miejsca tablicy roboczej. Po każdym scaleniu zawartość tablicy roboczej jest kopiowana z powrotem do tablicy oryginalnej.

# Applet MergeSort Workshop

Opisywany w tej części rozdziału proces sortowania łatwiej docenić, gdy naocznie można się przekonać, w jaki sposób on działa. W tym celu uruchom aplet *MergeSort Workshop*. Klikając przycisk *Step* (Krok), można wykonywać algorytm krok po kroku. Na rysunku 6.17 został przedstawiony stan algorytmu po wykonaniu trzech pierwszych kroków.




RYSUNEK 6.17. Aplet demonstracyjny sortowania przez scalanie

Strzałki *lower* oraz *upper* oznaczają zakres liczb aktualnie analizowany przez algorytm, z kolei strzałka *mid* wskazuje środek rozpatrywanego zakresu. Początkowo zakres obejmuje całą sortowaną tablicę, a następnie jest dzielony na połowy za każdym razem, gdy metoda `mergesort()` wywołuje sama siebie. Gdy rozpatrywany zakres zawiera jeden element, metoda zostaje zakończona — jest to warunek bazowy algorytmu. W przeciwnym przypadku scalane są dwie podtablice. Aplet wyświetla także komunikaty tekstowe, na przykład: „Will sort lower half: 0-5", informując użytkownika o wykonywanych czynnościach oraz o aktualnie rozpatrywanym zakresie tablicy.

Wiele kroków algorytmu polega na wywołaniu metody `mergesort()` lub zwracaniu wartości. Operacje porównywania oraz kopiowania są wykonywane wyłącznie podczas procesu scalania, kiedy to aplet generuje komunikaty o postaci „Merged 0-0 and 1-1 into workspace". Samego scalenia nie można zaobserwować, gdyż obszar roboczy nie jest prezentowany. Niemniej jednak, po skopiowaniu

odpowiedniego fragmentu obszaru roboczego do oryginalnej tablicy, można zobaczyć wyniki scalenia — słupki w danym zakresie zostaną wyświetlone w kolejności rosnącej.

Na samym początku zostaną posortowane dwa pierwsze słupki, następnie pierwsze trzy słupki, potem pierwsze dwa słupki zakresu 3-4, trzy słupki zakresu 3-5, sześć słupków zakresu 0-5, i tak dalej, zgodnie z sekwencją przedstawioną na rysunku 6.16. W końcu wszystkie słupki zostaną posortowane.

Klikając przycisk *Run* (Uruchom), można zażądać wykonania całego algorytmu. Proces ten można jednak w dowolnej chwili przerwać, klikając przycisk *Step*. Po przerwaniu wykonywania algorytmu można go realizować krokowo — naciskając przycisk *Step* — jak również wznowić wykonywanie „ciągłe" — klikając ponownie przycisk *Run*.

Podobnie jak w innych apletach demonstrujących algorytmy sortowania, także i w tym kliknięcie przycisku *New* (Nowe) powoduje zapisanie w tablicy nowej grupy nieposortowanych liczb oraz przełącza wypełnianie tablicy liczbami losowymi bądź liczbami zapisanymi w kolejności malejącej. Przycisk *Size* (Wielkość) umożliwia zmianę ilości sortowanych liczb — dostępne ilości to 12 oraz 100.

Wyjątkowo pouczające jest obserwowanie działania algorytmu podczas sortowania 100 liczb zapisanych początkowo w kolejności malejącej. Wyświetlane wyniki bardzo wyraźnie pokazują, jak kolejne zakresy są sortowane niezależnie do siebie, a następnie scalane, oraz jak zakresy stają się coraz to większe.

## Program mergeSort.java

W dalszej części rozdziału zostanie przedstawiony cały program *mergeSort.java*. W pierwszej kolejności przeanalizujemy jednak metodę, która realizuje algorytm sortowania przez scalanie. Oto jej kod:

```
private void recMergeSort(long[] workSpace, int lowerBound,
                                             int upperBound)
   {
   if(lowerBound == upperBound)            // jeśli zakres zawiera 1 elem.,
      return;                              // nie ma sensu sortować
   else
      {                                    // znajdujemy punkt środkowy
      int mid = (lowerBound+upperBound) / 2;
                                           // sortujemy lewą połówkę
      recMergeSort(workSpace, lowerBound, mid);
                                           // sortujemy prawą połówkę
      recMergeSort(workSpace, mid+1, upperBound);
                                           // scalamy posortowane połówki
      merge(workSpace, lowerBound, mid+1, upperBound);
      }  // koniec klauzuli else
   }  // koniec metody recMergeSort()
```

Jak można się przekonać, oprócz przypadku bazowego, kod metody zawiera wyłącznie cztery inne instrukcje. Jednak z nich wyznacza punkt środkowy analizowanego zakresu, dwie kolejne są rekurencyjnymi wywołaniami metody `recMergeSort()` (po jednym dla każdej połówki analizowanego zakresu), a ostatnią jest wywołanie metody `merge()` scalającej posortowane połówki zakresu. Przypadek bazowy zachodzi, gdy analizowany zakres zawiera tylko jeden element (czyli gdy `lowerBound ==upperBound`); w takiej sytuacji działanie metody zostanie bezzwłocznie zakończone.

W programie *mergeSort.java* użytkownik klasy ma dostęp jedynie do metody `mergeSort()`. Metoda ta tworzy tablicę `workSpace[]`, a następnie wywołuje rekurencyjną metodę `recMergeSort()`, która jest odpowiedzialna za faktyczne posortowanie tablicy. Stworzenie tablicy roboczej na potrzeby sortowania jest realizowane w metodzie `mergeSort()`. Gdyby tablica ta była tworzona w metodzie `recMergeSort()`, to każde rekurencyjne wywołanie tej metody powodowałoby ponowne utworzenie tablicy. Oczywiście byłoby to bardzo nieefektywne.

Metoda `merge()` przedstawiona wcześniej w programie *merge.java* (patrz listing 6.5) operowała na trzech oddzielnych tablicach: dwóch tablicach źródłowych oraz jednej tablicy wynikowej. Metoda `merge()` wykorzystana w programie *mergeSort.java* operuje na jednej tablicy — polu `theArray` klasy `DArray`. Argumentami wywołania tej metody są liczby określające punkt początkowy dolnej połowy podtabeli, punkt początkowy górnej połowy podtabeli oraz punkt końcowy górnej połowy podtabeli. Na podstawie tych informacji metoda oblicza wielkości obu fragmentów analizowanego obszaru tabeli.

Pełny kod programu *mergeSort.java* został przedstawiony na listingu 6.6. Program ten wykorzystuje klasę `DArray` — odmianę klasy implementującej tablice przedstawionej w rozdziale 2., wzbogaconą o metody `recMergeSort()` oraz `mergeSort()`. Metoda `main()` programu tworzy tablicę, zapisuje w niej 12 elementów, wyświetla jej początkową zawartość, następnie wywołuje metodę `mergeSort()`, po czym ponownie wyświetla zawartość tablicy.

LISTING 6.6
Program mergeSort.java

```
// mergeSort.java
// Program demonstruje rekurencyjne sortowanie przez scalanie
// Aby uruchomić program: C>java MergeSortApp
////////////////////////////////////////////////////////////////
class DArray
   {
   private long[] theArray;          // odwołanie do tablicy theArray
   private int nElems;               // ilość elementów
   //--------------------------------------------------------------
   public DArray(int max)            // konstruktor
      {
      theArray = new long[max];      // stworzenie tablicy
      nElems = 0;
      }
   //--------------------------------------------------------------
   public void insert(long value)    // zapisanie elementu w tablicy
      {
      theArray[nElems] = value;      // wstawiamy element
      nElems++;                      // inkrementujemy ilość elementów
      }
   //--------------------------------------------------------------
   public void display()             // wyświetlenie zawartości tablicy
      {
      for(int j=0; j<nElems; j++)             // dla każdego elementu,
         System.out.print(theArray[j] + " "); // wyświetlamy go
      System.out.println("");
      }
   //--------------------------------------------------------------
   public void mergeSort()                    // wywoływana przez main()
      {                                       // tworzymy obszar roboczy
```

```
      long[] workSpace = new long[nElems];
      recMergeSort(workSpace, 0, nElems-1);
      }
   //-------------------------------------------------------------
   private void recMergeSort(long[] workSpace, int lowerBound,
                                               int upperBound)
      {
      if(lowerBound == upperBound)             // jeśli zakres zawiera 1 elem.,
         return;                               // nie ma sensu sortować
      else
         {                                     // znajdujemy punkt środkowy
         int mid = (lowerBound+upperBound) / 2;
                                               // sortujemy lewą połówkę
         recMergeSort(workSpace, lowerBound, mid);
                                               // sortujemy prawą połówkę
         recMergeSort(workSpace, mid+1, upperBound);
                                               // scalamy posortowane połówki
         merge(workSpace, lowerBound, mid+1, upperBound);
         }  // koniec klauzuli else
      }  // koniec metody recMergeSort()
   //-------------------------------------------------------------
   private void merge(long[] workSpace, int lowPtr,
                           int highPtr, int upperBound)
      {
      int j = 0;                               // indeks obszaru roboczego
      int lowerBound = lowPtr;
      int mid = highPtr-1;
      int n = upperBound-lowerBound+1;         // ilość elementów

      while(lowPtr <= mid && highPtr <= upperBound)
         if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
         else
            workSpace[j++] = theArray[highPtr++];

      while(lowPtr <= mid)
         workSpace[j++] = theArray[lowPtr++];

      while(highPtr <= upperBound)
         workSpace[j++] = theArray[highPtr++];

      for(j=0; j<n; j++)
         theArray[lowerBound+j] = workSpace[j];
      }  // koniec metody merge()
   //-------------------------------------------------------------
   }  // koniec class DArray
////////////////////////////////////////////////////////////////
class MergeSortApp
   {
   public static void main(String[] args)
      {
```

```
      int maxSize = 100;            // wielkość tablicy
      DArray arr;                   // odwołanie do tablicy
      arr = new DArray(maxSize);    // tworzymy tablicę

      arr.insert(64);               // wstawiamy elementy
      arr.insert(21);
      arr.insert(33);
      arr.insert(70);
      arr.insert(12);
      arr.insert(85);
      arr.insert(44);
      arr.insert(3);
      arr.insert(99);
      arr.insert(0);
      arr.insert(108);
      arr.insert(36);

      arr.display();                // wyświetlamy elementy

      arr.mergeSort();              // sortujemy tablicę

      arr.display();                // ponownie wyświetlamy elementy
      }  // koniec metody main()
   }  // koniec class MergeSortApp
////////////////////////////////////////////////////////////////
```

Wyniki wykonania tego programu przedstawiają jedynie nieposortowaną oraz posortowaną zawartość tablicy:

```
64 21 33 70 12 85 44 3 99 0 108 36
0 3 12 21 33 36 44 64 70 85 99 108
```

Umieszczając w kodzie metody `recMergeSort()` dodatkowe instrukcje, można by generować komunikaty informujące o czynnościach, jakie program wykonuje podczas sortowania. Poniższy przykład pokazuje, jak mogłyby wyglądać takie informacje, generowane podczas sortowania tablicy zawierającej 4 liczby `{64, 21, 33, 70}` (tablicę tę można sobie wyobrazić jako dolną połówkę tablicy z rysunku 6.15).

```
Wywołanie 0-3
   Sortowana będzie dolna połówka 0-3
   Wywołanie 0-1
      Sortowana będzie dolna połówka 0-1
         Wywołanie 0-0
         Przypadek bazowy - zwracane 0-0
      Sortowana będzie górna połówka 0-1
         Wywołanie 1-1
         Przypadek bazowy - zwracane 1-1
      Scalanie połówek w zakres 0-1
   Zwracane 0-1                    theArray=21 64 33 70
   Sortowana będzie górna połówka 0-3
```

```
   Wywołanie 2-3
      Sortowana będzie dolna połówka 2-3
         Wywołanie 2-2
         Przypadek bazowy - zwracane 2-2
      Sortowana będzie górna połówka 2-3
         Wywołanie 3-3
         Przypadek bazowy - zwracane 3-3
      Scalanie połówek w zakres 2-3
   Zwracane 2-3                    theArray=21 64 33 70
   Scalanie połówek w zakres 0-3
Zwracane 0-3                    theArray=21 33 64 70
```

Mniej więcej te same informacje zwróciłby aplet demostracyjny *MergeSort*, gdyby był w stanie sortować jedynie cztery elementy. Analiza powyższych wyników oraz porównanie ich z kodem metody `recMergeSort()` i rysunkiem 6.15 umożliwi dokładne zrozumienie zasady działania omawianego algorytmu sortowania.

# Efektywność działania algorytmu sortowania przez scalanie

Zgodnie z podanymi wcześniej informacjami, efektywność działania algorytmu sortowania przez scalanie wynosi $O(N * \log N)$. Ale skąd to wiadomo? Przekonajmy się, w jaki sposób można określić, ile razy podczas działania algorytmu dane będą musiały zostać skopiowane oraz ile razy trzeba je będzie porównywać. Zakładamy, że właśnie kopiowanie danych oraz ich porównywanie są operacjami zajmującymi najwięcej czasu oraz że rekurencyjne wywołania metod i zwracanie wartości przez metodę nie powoduje znaczących narzutów czasowych.

## Ilość kopiowań

Przeanalizujmy rysunek 6.15. Każda komórka umieszczona poniżej górnego wiersza liczb reprezentuje daną skopiowaną z tablicy do obszaru roboczego.

Dodając do siebie ilość wszystkich komórek widocznych na rysunku 6.15 (w siedmiu ponumerowanych etapach), można się przekonać, że posortowanie 8 elementów wymaga wykonania 24 operacji kopiowania. $\text{Log}_2 8$ ma wartość 3, a zatem $8 * \log_2 8$ daje 24. Wynika stąd, że dla tablicy zawierającej 8 elementów, ilość operacji kopiowania jest proporcjonalna do wartości $N * \log_2 N$.

Na analizowany proces sortowania można jednak spojrzeć w inny sposób. Otóż posortowanie 8 liczb wymaga 3 *poziomów*, a każdy z nich składa się z 8 operacji kopiowania. Poziom oznacza w tym przypadku wszystkie operacje kopiowania do podtablicy o takiej samej wielkości. Na pierwszym poziomie wykorzystywane są cztery podtablice 2-elementowe, na drugim poziomie — dwie podtablice 4-elementowe, a na trzecim poziomie — jednak podtablica 8-elementowa. Każdy poziom zawiera 8 elementów. A zatem także przy takim podejściu wykonywanych jest $3 * 8$, czyli 24 operacje kopiowania.

Analizując fragmenty rysunku 6.15, można się przekonać, że do posortowania 4 elementów koniecznych jest 8 operacji kopiowania (etapy 1, 2 oraz 3), a do posortowania 2 elementów — dwie operacje kopiowania. Identyczne obliczenia pozwalają określić ilość operacji kopiowania, jakie należy wykonać w celu posortowania większych tablic. Podsumowanie tych informacji zostało przedstawione w tabeli 6.4.

TABELA 6.4.
Ilość operacji wykonywanych w przypadku gdy N jest potęgą liczby 2

| N | log2N | Ilość operacji kopiowania do obszaru roboczego | Łączna ilość operacji kopiowania | Ilość porównań Maksimum (minimum) |
|---|---|---|---|---|
| 2 | 1 | 2 | 4 | 1 (1) |
| 4 | 2 | 8 | 16 | 5 (4) |
| 8 | 3 | 24 | 48 | 17 (12) |
| 16 | 4 | 64 | 128 | 49 (32) |
| 32 | 5 | 160 | 320 | 129 (80) |
| 64 | 6 | 384 | 768 | 321 (192) |
| 128 | 7 | 896 | 1792 | 769 (448) |

W rzeczywistości sortowane liczby nie są kopiowane jedynie do obszaru roboczego, lecz także z powrotem do oryginalnej tablicy. Oznacza to, że ilość operacji kopiowania jest dwukrotnie większa, co zostało odzwierciedlone w kolumnie „Łączna ilość operacji kopiowania". Ostatnia kolumna tabeli 6.4 zawiera informacje dotyczące ilości porównań; to zagadnienie zostanie opisane w dalszej części rozdziału.

Nieco trudniej jest obliczyć ilość operacji kopiowania oraz ilość porównań w przypadku gdy *N* nie jest wielokrotnością liczby 2. Wyniki uzyskiwane w tych przypadkach mieszczą się w granicach uzyskiwanych dla liczb stanowiących potęgi liczby 2. Na przykład dla 12 elementów koniecznych jest 88 operacji kopiowania, a dla 100 elementów — 1344 operacje kopiowania.

## Ilość porównań

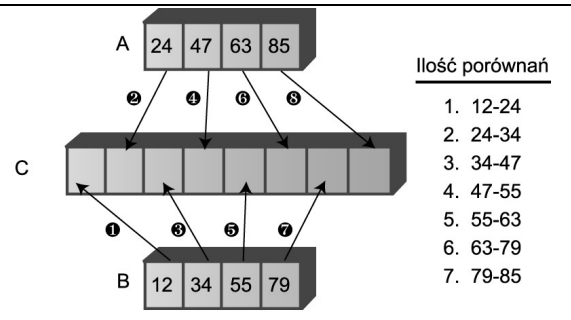
W przypadku algorytmu sortowania przez scalanie ilość porównań, jakie należy wykonać w celu posortowania tablicy, zawsze jest nieco mniejsza od ilości koniecznych operacji kopiowania. O ile mniejsza? Zakładając, że ilość sortowanych elementów jest wielokrotnością liczby 2, dla każdej niezależnej operacji scalania maksymalna ilość porównań jest zawsze o jeden mniejsza od ilości scalanych elementów. Z kolei minimalna ilość porównań odpowiada połowie ilości scalanych elementów. O prawdziwości tych obliczeń można się przekonać, analizując rysunek 6.18, przedstawiający dwie sytuacje, jakie mogą zajść podczas scalania dwóch 4-elementowych tablic.

W pierwszym przypadku elementy są rozmieszczone naprzemiennie, a do scalenia tablic należy wykonać siedem porównań. W drugim przypadku wszystkie elementy pierwszej tablicy są mniejsze od wszystkich elementów drugiej tablicy, a zatem do scalenia obu tablic konieczne będą tylko cztery porównania.

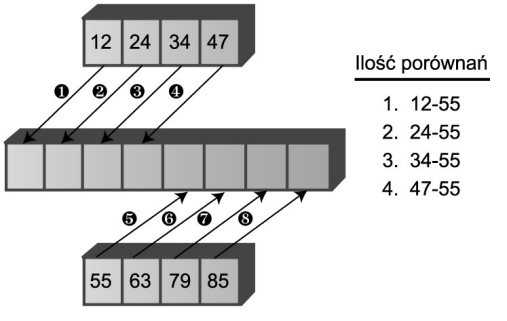
Podczas każdego sortowania wykonywanych jest wiele operacji scalania, a zatem należy zsumować ilości porównań wykonywanych w ramach poszczególnych operacji scalania. Odwołując się ponownie do rysunku 6.15, można się przekonać, że posortowanie ośmiu elementów wymaga wykonania siedmiu operacji scalania. Ilości scalanych elementów oraz odpowiadające im ilości porównań zostały przedstawione w tabeli 6.5.

Maksymalna ilość porównań, jakie należy wykonać podczas każdej operacji scalania, jest o jeden mniejsza od ilości scalanych elementów. Po dodaniu odpowiednich liczb dla wszystkich wykonywanych operacji scalania okazuje się, że maksymalna ilość porównań wynosi 17.

Z kolei minimalna ilość porównań zawsze jest o połowę mniejsza od ilości scalanych elementów, co, po zsumowaniu wszystkich operacji scalania, daje w rezultacie wynik 12. W taki sam sposób zostały wyznaczone wyniki podane w kolumnie „Ilość porównań" tabeli 6.4. Faktyczna ilość porównań, jakie należy wykonać w celu posortowania konkretnej tablicy, zależy od kolejności, w jakiej są zapisane jej elementy; niemniej jednak zawsze będzie to liczba leżąca pomiędzy przedstawioną powyżej wartością minimalną i maksymalną.




a) najgorszy z możliwych przypadków




b) najlepszy z możliwych przypadków

RYSUNEK 6.18. Maksymalna i minimalna ilość porównań

TABELA 6.5.
Porównania wykonywane podczas sortowania 8 elementów

| Numer kroku | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Ilość sumaryczna |
|---|---|---|---|---|---|---|---|---|
| Ilość scalanych elementów (*N*) | 2 | 2 | 4 | 2 | 2 | 4 | 8 | 24 |
| Maksymalna ilość porównań (*N* – 1) | 1 | 1 | 3 | 1 | 1 | 3 | 7 | 17 |
| Minimalna ilość porównań (*N* / 2) | 1 | 1 | 2 | 1 | 1 | 2 | 4 | 12 |

# Eliminacja rekurencji

Niektóre algorytmy można zaimplementować w sposób rekurencyjny, a inne nie. Jak można się było przekonać, rekurencyjne metody `triangle()` oraz `factorial()` można zaimplementować przy użyciu pętli, a co więcej, rozwiązanie takie będzie bardziej efektywne niż użycie rekurencji. Niemniej jednak, wiele algorytmów działających według zasady „dziel i zwyciężaj”, takich jak na przykład sortowanie przez scalanie, działa bardzo dobrze, wykorzystując właśnie rekurencję.

Często zdarza się, że algorytm można łatwo wyobrazić sobie jako metodę rekurencyjną, jednak po jego implementacji okazuje się, że takie rozwiązanie jest mało efektywne. W takich przypadkach warto przekształcić algorytm rekurencyjny na odpowiadający mu algorytm, który nie wykorzystuje rekurencji. Takie przekształcenie algorytmu rekurencyjnego często może bazować na wykorzystaniu stosu.

# Rekurencja i stosy

Istnieje bardzo bliski związek pomiędzy rekurencją i stosami. W rzeczywistości przeważająca większość kompilatorów implementuje mechanizmy rekurencyjne właśnie w oparciu o stosy. Zgodnie z informacjami podanymi wcześniej, w momencie wywoływania metody kompilator zapisuje na stosie argumenty przekazywane do metody oraz adres powrotu (określający miejsce, gdzie zostanie przekazane sterowanie po jej zakończeniu), po czym przekazuje sterowanie do metody. Gdy metoda kończy działanie, informacje zapisane na stosie przed jej uruchomieniem zostają ze stosu usunięte. Argumenty metody znikają, a sterowanie jest przekazywane pod wskazany adres powrotu.

# Symulowanie metod rekurencyjnych

W tej części rozdziału zostanie przedstawiona metoda przekształcania metod rekurencyjnych na metody bazujące na wykorzystaniu stosu. Czy Czytelnik pamięta rekurencyjną metodę `triangle()` przedstawioną w pierwszej części tego rozdziału? Oto jej kod:

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

Algorytm ten zostanie teraz podzielony na poszczególne czynności, z których każda będzie reprezentowana przez osobną klauzulę `case` w instrukcji `switch` (podobnego podziału metody można także dokonać, stosując instrukcję `goto` dostępną w C++ oraz innych językach programowania, jednak w Javie instrukcja ta nie istnieje).

Instrukcja `switch` została umieszczona w metodzie o nazwie `step()`. Każde wywołanie tej metody powoduje wykonanie jednej z klauzul `case` umieszczonych w instrukcji `switch`. Cykliczne wywoływanie metody `step()` spowoduje w końcu wykonanie wszystkich operacji algorytmu.

Przedstawiona powyżej metoda `triangle()` wykonuje dwa rodzaje operacji. Pierwszą z nich jest wykonanie obliczeń arytmetycznych koniecznych do wyznaczenia wartości liczby trójkątnej. Wymaga ona sprawdzenia, czy wartość `n` jest równa `1` oraz dodanie `n` do wartości zwróconej przez rekurencyjne wywołanie metody. Niemniej jednak metoda `triangle()` wykonuje także inne operacje, konieczne do jej poprawnego wykonania; można do nich zaliczyć przekazywanie sterowania, pobieranie argumentów oraz zwracanie wartości wynikowej. Operacji tych nie można zauważyć, analizując kod metody, jednak występują one we wszystkich tworzonych metodach. Poniżej zostały ogólnie opisane czynności wykonywane podczas wywoływania metody:

- W momencie wywoływania metody jej argumenty oraz adres powrotu są zapisywane na stosie.
- Metoda może uzyskać dostęp do argumentów, pobierając je z wierzchołka stosu.
- Bezpośrednio przed zakończeniem działania metody ze stosu pobierany jest adres powrotu, a następnie zarówno ten adres, jak i argumenty wywołania metody są usuwane ze stosu.

Program *stackTriangle.java* zawiera trzy klasy: `Params`, `StackX` oraz `StackTriangleApp`. Klasa `Param` reprezentuje adres powrotu oraz argument wywołania metody — `n`; obiekty tej klasy są zapisywane na stosie. Klasa `StackX` przypomina klasy implementujące stosu przedstawione w innych rozdziałach tej książki; wyróżnia ją jedynie to, iż umożliwia operowanie na obiektach klasy `Params`. Klasa `StackTriangleApp` zawiera cztery metody: `main()`, `recTriangle()`, `step()` oraz znaną już metodę `getInt()` służącą do pobierania danych liczbowych.

Metoda `main()` prosi użytkownika o podanie liczby, następnie wywołuje metodę `recTriangle()` (której zadaniem jest wyznaczenie liczby trójkątnej odpowiadającej liczbie podanej w wywołaniu), a w końcu wyświetla wyniki.

Metoda `recTriangle()` tworzy obiekt `StackX` i przypisuje właściwości `codePart` wartość `1`. Pozostałą częścią metody jest pętla `while`, która cyklicznie wywołuje metodę `step()`. Pętla ta zostanie zakończona dopiero w momencie, gdy metoda `step()` zwróci wartość `true`, co nastąpi w przypadku wykonania szóstej klauzuli `case` — czyli punktu zakończenia metody. Metoda `step()` jest w zasadzie dużą instrukcją `switch`, której klauzule `case` odpowiadają fragmentom kodu oryginalnej metody `triangle()`. Kod programu *stackTriangle.java* został przedstawiony na listingu 6.7.

LISTING 6.7.
Program stackTriangle.java

```
// stackTriangle.java
// Program wyznacza liczby trójkątne, zastępując rekurencję stosem
// Aby uruchomić program: C>java StackTriangleApp
import java.io.*;                    // for I/O
////////////////////////////////////////////////////////////////
class Params                         // parametry zapisywane na stosie
   {
   public int n;
   public int returnAddress;

   public Params(int nn, int ra)
      {
      n=nn;
      returnAddress=ra;
      }
   }  // koniec klasy Params
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;                  // wielkość tablicy StackX
   private Params[] stackArray;
   private int top;                      // wierzchołek stosu
//--------------------------------------------------------------
   public StackX(int s)                  // konstruktor
      {
      maxSize = s;                       // określamy wielkość tablicy
      stackArray = new Params[maxSize];  // tworzymy tablicę
      top = -1;                          // na razie tablica jest pusta
      }
//--------------------------------------------------------------
   public void push(Params p)   // umieszczenie elementu na wierzchołku stosu
      {
```

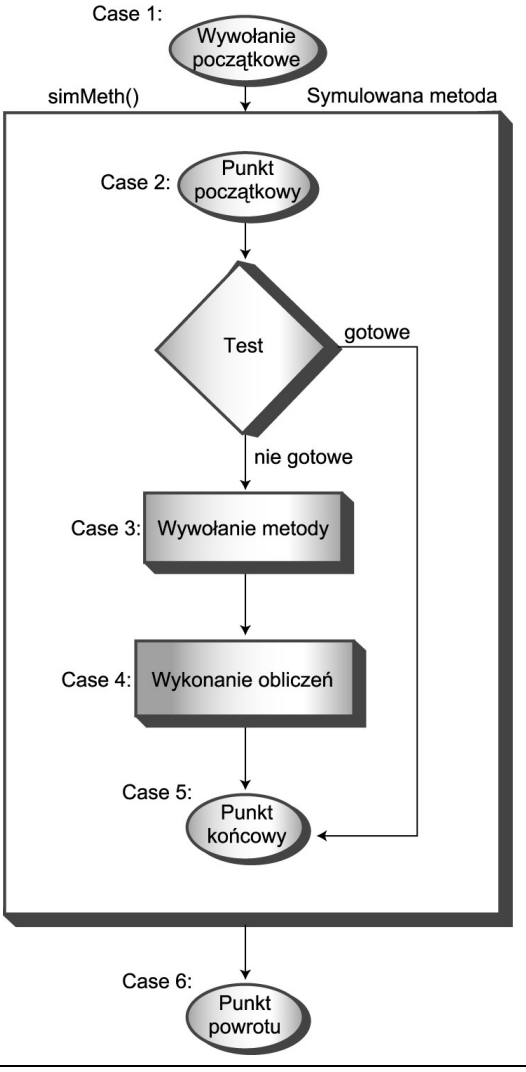
```
      stackArray[++top] = p;    // inkrementujemy indeks wierzchołka, wstawiamy element
      }
//--------------------------------------------------------------
   public Params pop()          // pobranie elementu z wierzchołka stosu
      {
      return stackArray[top--]; // pobierany element, dekrementujemy indeks wierzchołka
      }
//--------------------------------------------------------------
   public Params peek()         // odczyt elementu z wierzchołka stosu
      {                         // bez usuwania elementu
      return stackArray[top];
      }
//--------------------------------------------------------------
   }  // koniec klasy StackX
////////////////////////////////////////////////////////////////
class StackTriangleApp
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;
   static int codePart;
   static Params theseParams;
//--------------------------------------------------------------
   public static void main(String[] args) throws IOException
      {
      System.out.print("Podaj liczbe: ");
      theNumber = getInt();
      recTriangle();
      System.out.println("Odpowiadajaca jej liczba trojkatna="+theAnswer);
      }  // end main()
//--------------------------------------------------------------
   public static void recTriangle()
      {
      theStack = new StackX(10000);
      codePart = 1;
      while( step() == false)   // wywołujemy step() aż do zwrócenia true
         ;                      // instrukcja pusta
      }
//--------------------------------------------------------------
   public static boolean step()
      {
      switch(codePart)
         {
         case 1:                            // wywołanie początkowe
            theseParams = new Params(theNumber, 6);
            theStack.push(theseParams);
            codePart = 2;
            break;
         case 2:                            // "wejście" do metody
            theseParams = theStack.peek();
            if(theseParams.n == 1)          // sprawdzenie warunku
               {
```

```
            theAnswer = 1;
            codePart = 5;                  // koniec
            }
         else
            codePart = 3;                  // wywołanie rekurencyjne
         break;
      case 3:                              // wywołanie metody
         Params newParams = new Params(theseParams.n - 1, 4);
         theStack.push(newParams);
         codePart = 2;                     // wejście do metody
         break;
      case 4:                              // obliczenia
         theseParams = theStack.peek();
         theAnswer = theAnswer + theseParams.n;
         codePart = 5;
         break;
      case 5:                                    // wyjście z metody
         theseParams = theStack.peek();
         codePart = theseParams.returnAddress;  // (4 lub 6)
         theStack.pop();
         break;
      case 6:                                    // zakończenie działania metody
         return true;
      }  // koniec instrukcji switch
   return false;
   }  // koniec metody triangle
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   }  // koniec klasy StackTriangleApp
////////////////////////////////////////////////////////////////
```

Powyższy program wyznacza wartość liczb trójkątnych, podobnie jak program *triangle.java* przedstawiony na początku tego rozdziału (na listingu 6.1). Oto przykładowe wyniki jego wykonania:

```
Podaj liczbe 100
Odpowiadajaca jej liczba trojkatna=5050
```

Rysunek 6.19 przedstawia relację pomiędzy poszczególnymi klauzulami `case` oraz fragmentami algorytmu.




RYSUNEK 6.19. Klauzule case oraz metoda step()

Choć cały przedstawiany program symuluje jedną metodę, nie nosi żadnej nazwy, gdyż w rzeczywistości nie jest on prawdziwą metodą. Tej symulowanej metodzie nadamy umowną nazwę `simMeth()`. Jej początkowe wywołanie (w klauzuli `case 1`) zapisuje na stosie wartość podaną przez użytkownika oraz wartość wynikową 6, po czym przechodzi do punktu początkowego (klauzula `case 2`).

W punkcie początkowym (klauzula `case 2`) metoda `simMeth()` sprawdza, czy przekazany argument ma wartość 1. W tym celu odczytywana jest wartość elementu znajdującego się na wierzchołku stosu (bez jego usuwania). Jeśli argument ma wartość 1, zachodzi przypadek bazowy i sterowanie przekazywane jest do punktu końcowego (klauzula `case 5`). W przeciwnym wypadku, czyli gdy wartość przekazanego argumentu jest różna od 1, metoda rekurencyjnie wywołuje sama siebie (klauzula `case 3`). To „rekurencyjne" wywołanie polega na umieszczeniu na stosie wartości `n - 1` i adresu powrotu wynoszącego 4 oraz przejścia do punktu początkowego metody (klauzula `case 2`).

Podczas zakańczania rekurencyjnego wywołania metody metoda `simMeth()` dodaje argument `n` do wartości zwróconej przez wywołanie rekurencyjne. W końcu rekurencyjne wywołanie metody

zostanie zakończone (klauzula `case 5`). W tym momencie ze stosu jest usuwany obiekt `Params` umieszczony na jego wierzchołku; obiekt ten nie jest już do niczego potrzebny.

Adres powrotu, podany w początkowym wywołaniu, miał wartość 6. A zatem, w momencie zakańczania działania metody `simMeth()`, sterowanie jest przekazywane do klauzuli `case 6`. Umieszczony w niej kod zwraca wartość `true`, co powoduje zakończenie pętli `while` w metodzie `recTriangle()`.

Warto zwrócić uwagę, iż w powyższym opisie metody `simMeth()` używane są takie zwroty jak: *argument*, *wywołanie rekurencyjne* oraz *adres powrotu*. Należy pamiętać, że w tym kontekście opisują one jedynie symulowane cechy, a nie faktyczne informacje i operacje wykonywane przez program.

Gdyby w poszczególnych klauzulach `case` zostały umieszczone instrukcje wyświetlające informacje o czynnościach wykonywanych przez metodę `simMeth()`, to wygenerowane wyniki mogłyby przypominać te przedstawione poniżej:

```
Podaj liczbe: 4
case 1. theAnswer=0  Stos:
case 2. theAnswer=0  Stos: (4, 6)
case 3. theAnswer=0  Stos: (4, 6)
case 2. theAnswer=0  Stos: (4, 6) (3, 4)
case 3. theAnswer=0  Stos: (4, 6) (3, 4)
case 2. theAnswer=0  Stos: (4, 6) (3, 4) (2, 4)
case 3. theAnswer=0  Stos: (4, 6) (3, 4) (2, 4)
case 2. theAnswer=0  Stos: (4, 6) (3, 4) (2, 4) (1, 4)
case 5. theAnswer=1  Stos: (4, 6) (3, 4) (2, 4) (1, 4)
case 4. theAnswer=1  Stos: (4, 6) (3, 4) (2, 4)
case 5. theAnswer=3  Stos: (4, 6) (3, 4) (2, 4)
case 4. theAnswer=3  Stos: (4, 6) (3, 4)
case 5. theAnswer=6  Stos: (4, 6) (3, 4)
case 4. theAnswer=6  Stos: (4, 6)
case 5. theAnswer=10 Stos: (4, 6)
case 6. theAnswer=10 Stos:
Odpowiadajaca jej liczba trojkatna=10
```

Cyfra wyświetlana przy słowie „case” określa aktualnie wykonywany fragment kodu. Wyświetlana jest także zawartość stosu (składająca się z obiektów `Params` zawierających argument `n` oraz adres powrotu). Metoda `simMeth()` jest rozpoczynana i zakańczana cztery razy (czemu odpowiadają klauzule `case 2` oraz `case 5`). Dopiero w momencie zakańczania tej metody obliczone wyniki zostają zapisywane w zmiennej `theAnswer`.

## Czego to dowodzi?

Program *stackTriangle.java* przedstawiony na listingu 6.7 jest przykładem programu, który, w sposób mniej lub bardziej systematyczny, przekształca algorytm rekurencyjny w rozwiązanie wykorzystujące stos. Sugeruje to, że w podobny sposób można przekształcić dowolny program wykorzystujący rekurencję. I faktycznie jest to możliwe.

Wkładając nieco dodatkowej pracy, można systematycznie zmodyfikować przedstawiony program w celu poprawienia efektywności jego działania. Modyfikacje te mogą nawet doprowadzić do całkowitego zlikwidowania instrukcji `switch`.

W rzeczywistości znacznie bardziej praktycznym rozwiązaniem jest przemyślenie i zaimplementowanie algorytmu od podstaw, z wykorzystaniem stosu, a nie rekurencji. Listing 6.8 pokazuje jak w takim przypadku może się zmienić kod metody `triangle()`.

LISTING 6.8.
Program stackTriangle2.java

```
// stackTriangle2.java
// Program wyznacza liczby trójkątne, wykorzystując stos, a nie rekurencję
// Aby uruchomić program: C>java StackTriangle2App
import java.io.*;                       // konieczne do operacji wejścia/wyjścia
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;                 // wielkość tablicy stosu
   private int[] stackArray;
   private int top;                     // wierzchołek stosu
//--------------------------------------------------------------
   public StackX(int s)                 // konstruktor
      {
      maxSize = s;
      stackArray = new int[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(int p)              // umieszczenie elementu na wierzchołku stosu
      { stackArray[++top] = p; }
//--------------------------------------------------------------
   public int pop()                     // pobranie elementu z wierzchołka stosu
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public int peek()                    // odczytanie wartości elementu z wierzchołka
      { return stackArray[top]; }
//--------------------------------------------------------------
   public boolean isEmpty()             // true jeśli stos jest pusty
      { return (top == -1); }
//--------------------------------------------------------------
   }  // koniec klasy StackX
////////////////////////////////////////////////////////////////
class StackTriangle2App
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Podaj liczbe: ");
      System.out.flush();
      theNumber = getInt();
      stackTriangle();
      System.out.println("Odpowiadajaca jej liczba trojkatna="+theAnswer);
      }  // end main()
```

```
//--------------------------------------------------------------
   public static void stackTriangle()
      {
      theStack = new StackX(10000);    // stworzenie stosu

      theAnswer = 0;                   // inicjalizacja wartości wynikowej

      while(theNumber > 0)             // aż do momentu gdy n wynosi 1
         {
         theStack.push(theNumber);     // zapis na stosie
         --theNumber;                  // dekrementacja liczby
         }
      while( !theStack.isEmpty() )     // aż do momentu gdy stos jest pusty,
         {
         int newN = theStack.pop();    // pobranie wartości,
         theAnswer += newN;            // dodanie wartości do odpowiedzi
         }
      }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // koniec klasy StackTriangle2App
////////////////////////////////////////////////////////////////
```

W metodzie `stackTriangle()` zostały umieszczone dwie krótkie pętle `while`, zastępujące całą metodę `step()` wykorzystywaną w programie *stackTriangle.java*. Oczywiście, analizując ten program, można się przekonać, że istnieje możliwość rezygnacji z wykorzystania stosu i sprowadzenia całego programu do prostej pętli `while`. Niemniej jednak, w bardziej złożonych algorytmach stos musi być wykorzystywany.

Bardzo często należy przeprowadzać eksperymenty, aby przekonać się, które z rozwiązań — rekurencyjne, wykorzystujące stos, czy też bazujące na prostej pętli — jest najbardziej efektywne w danej sytuacji.

# Niektóre interesujące zastosowania rekurencji

W tej części rozdziału zostaną pokrótce przedstawione niektóre inne, interesujące zastosowania rekurencji. Wziąwszy pod uwagę różnorodność przedstawionych tu przykładów, można dojść do wniosku, że rozwiązania rekurencyjne mogą się przydawać w wielu niespodziewanych sytuacjach. W tej części

rozdziału zostaną przeanalizowane trzy zagadnienia: podnoszenie liczby do potęgi, rozwiązanie „problemu plecakowego" oraz wybieranie członków ekspedycji alpinistycznej. We wszystkich tych przypadkach zostaną przedstawione wyłącznie ogólne pojęcia związane z danym zagadnieniem, natomiast implementacja programu będzie stanowić ćwiczenie, które Czytelnik może wykonać we własnym zakresie.

## Podnoszenie liczby do potęgi

Bardziej zaawansowane kalkulatory kieszonkowe pozwalają na podnoszenie liczb do dowolnej potęgi. Zazwyczaj na klawiaturze takich kalkulatorów można znaleźć przycisk *x^y*, gdzie znak ^ oznacza, że liczba *x* jest podnoszona do potęgi *y*. W jaki sposób można wykonać potęgowanie, w przypadku gdy kalkulator nie będzie dysponował odpowiednim przyciskiem? Można przyjąć, że podniesienie liczby z do potęgi y wymaga pomnożenia liczby x przez siebie samą y razy. A zatem, jeśli x wynosi 2, a y wynosi 8 ($2^8$), to wymagane byłoby wykonanie następującego działania: 2 ∗ 2 ∗ 2 ∗ 2 ∗ 2 ∗ 2 ∗ 2 ∗ 2. Niemniej jednak w przypadkach, gdy wartość y jest bardzo duża, rozwiązanie to może się okazać męczące. Czy istnieje zatem jakiś lepszy sposób?

Jednym z możliwych rozwiązań jest zmiana podejścia do problemu i mnożenie przez siebie iloczynów liczby 2, a nie uzyskanego wyniku razy 2. Rozpatrzmy przykład podnoszenia liczby 2 do potęgi 8 ($2^8$). W tym przypadku konieczne jest pomnożenie przez siebie ośmiu liczb 2. Załóżmy, że zaczynamy od działania 2 ∗ 2 = 4. Jego wykonanie spowoduje, że dwie liczby 2 zostaną już wykorzystane, pozostaje jednak kolejnych 6 liczb. Jednak w tym momencie pojawia się nowa wartość, która bierze udział w działaniu — jest nią wartość 4. Można zatem spróbować pomnożyć 4 ∗ 4, co daje 16 i wykorzystuje już 4 liczby 2 (ponieważ każda liczba 4 to dwie pomnożone przez siebie liczby 2). A zatem należy wykorzystać jeszcze cztery liczby 2, jednak w tym momencie można już operować na wartości 16, a mnożenie 16 ∗ 16 wykorzystuje dokładnie osiem liczb 2 (gdyż każda liczba 16 to cztery pomnożone liczby 2).

A zatem udało się określić wartość $2^8$, wykonując jedynie trzy mnożenia, a nie siedem. Oznacza to, że złożoność takiego algorytmu wynosi $O(\log N)$, a nie $O(N)$.

Czy można zaimplementować ten proces w formie algorytmu, nadającego się do wykonywania na komputerze? Schemat jego działania opiera się na równaniu $x^y = (x^2)^{y/2}$. W przedstawionym przykładzie $2^8 = (2^2)^{8/2}$, czyli $2^8 = (2^2)^4$. Równanie to jest poprawne, gdyż potęgowanie potęgi daje te same wyniki co pomnożenie wykładników.

Zakładamy jednak, że nasz komputer nie jest w stanie podnosić liczb do potęgi, a zatem nie możemy obliczyć wartości $(2^2)^4$. Sprawdźmy zatem, czy można przekształcić to wyrażenie do postaci wykorzystującej wyłącznie mnożenie. Sztuczka polega na tym, aby rozpocząć obliczenia od zastąpienia wartości $2^2$ nową zmienną.

Załóżmy, że $2^2 = a$. W takim przypadku $2^8$ wynosi $(2^2)^4$ czyli $a^4$. Niemniej jednak, bazując na początkowym równaniu, $a^4$ można zapisać jako $(a^2)^2$; a zatem $2^8 = (a^2)^2$.

Ponownie wykorzystujemy nową zmienną — *c* — zastępując nią $a^2$. Tym razem $(c)^2$ można zapisać jako $(c^2)^1$. Także i to wyrażenie jest równe $2^8$.

W ten sposób uzyskaliśmy problem, który można rozwiązać, wykorzystując wyłącznie mnożenie — wystarczy pomnożyć *c* razy *c*.

Ten schemat postępowania można zaimplementować w postaci metody rekurencyjnej pozwalającej na wyznaczanie potęg. Metodzie tej można, na przykład, nadać nazwę `power()`. Argumentami wywołania tej metody są liczby *x* oraz *y*, a metoda zwraca wartość $x^y$. Zmiennymi pomocniczymi, takimi jak *a* oraz *c* nie trzeba się przejmować, gdyż podczas każdego rekurencyjnego wywołania

metody, argumentom *x* oraz *y* są przypisywane nowe wartości. Konkretnie rzecz biorąc, argumentowi *x* przypisywana jest wartość *x* * *x*, a argumentowi *y* — wartość *y* / 2. Poniżej została przedstawiona sekwencja argumentów i zwracanych wartości dla argumentów początkowych wynoszących odpowiednio *x* = 2 oraz *y* = 8:

```
x=2, y=8
x=4, y=4
x=16, y=2
x=256, y=1
Wynik 256, x=256, y=1
Wynik 256, x=16, y=2
Wynik 256, x=4, y=4
Wynik 256, x=2, y=8
```

Metoda zwraca wartość wynikową w momencie, gdy argument *y* uzyska wartość 1. Gdy to nastąpi, wartość wynikowa jest przekazywana bez żadnych modyfikacji aż na sam początek sekwencji wywołań metody.

Przedstawiony przykład zakłada, że wartość y zawsze jest parzysta (nawet po kolejnych dzieleniach). Jednak nie jest to jedyny możliwy przypadek. Poniżej pokazano, w jaki sposób można zmodyfikować algorytm, tak aby obsługiwał także nieparzyste wartości wykładnika (*y*). Realizując kolejne rekurencyjne wywołania metody, należy używać dzielenia całkowitego lub pomijać wszelkie ułamki, które mogą wystąpić podczas dzielenia wartości *y* przez 2. Natomiast później, podczas procesu kończenia kolejnych rekurencyjnych wywołań metody, w przypadku gdy wartość *y* jest nieparzysta, zwrócony wynik należy dodatkowo pomnożyć razy *x*. Oto sekwencja wywołań realizowana podczas wyznaczania wartości $3^{18}$:

```
x=3, y=18
x=9, y=9
x=81, y=4
x=6561, y=2
x=43046721, y=1
Wynik 43046721, x=43046721, y=1
Wynik 43046721, x=6561, y=2
Wynik 43046721, x=81, y=4
Wynik 387420489, x=9, y=9 // y jest nieparzyste, więc wynik mnożymy razy x
Wynik 387420489, x=3, y=18
```

# Problem plecakowy

Jest to klasyczny problem informatyczny. W najprostszej postaci polega on na podejmowaniu prób umieszczenia w plecaku elementów o różnej wadze, tak aby sumaryczna waga plecaka osiągnęła pewną określoną wartość. Nie ma wymogu, by w plecaku zostały umieszczone wszystkie elementy.

Na przykład, załóżmy, że sumaryczna waga plecaka ma wynosić 20 kilogramów, a do dyspozycji mamy elementy o wagach 11, 8, 7, 6 oraz 5 kilogramów. W przypadku niewielkiej ilości elementów, ludzie całkiem dobrze radzą sobie z rozwiązywaniem tego problemu, sprawdzając kolejne możliwości. Dlatego Czytelnik zapewne nie będzie mieć większych problemów z określeniem, że zamierzoną wartość 20 można uzyskać dodając liczby 8, 7 oraz 5.

Jednak, aby to komputer rozwiązał ten problem, konieczne będzie podanie mu bardziej szczegółowych instrukcji. Poniżej został przedstawiony algorytm rozwiązania:

(1) Jeśli w jakimkolwiek momencie realizacji procesu suma wag wybranych elementów zrówna się z wagą docelową, należy zakończyć działanie.
(2) Początkowo wybierany jest pierwszy element. Wagi pozostałych elementów muszą zrównać się z wagą docelową, pomniejszoną o wagę pierwszego wybranego elementu. Jest to nowa waga docelowa.
(3) Kolejno należy wypróbować wszystkie dostępne kombinacje pozostałych elementów. Należy jednak zauważyć, że w rzeczywistości wcale nie trzeba sprawdzać wszystkich kombinacji, gdyż sumowanie można zakończyć w momencie, gdy sumaryczna waga wybranych elementów przekracza wagę docelową.
(4) Jeśli nie uda się odnaleźć kombinacji elementów o zadanej wadze, to należy odrzucić pierwszy element i rozpocząć cały proces od początku, wybierając element kolejny.
(5) W podobny sposób należy rozpocząć cały proces sprawdzania, wybierając na początku trzeci, czwarty oraz kolejne elementy, aż do momentu przeanalizowania całego zbioru dostępnych elementów. Sprawdzenie wszystkich możliwości będzie oznaczać, że poszukiwane rozwiązanie nie istnieje.

W przedstawionym powyżej przykładzie rozpocznijmy od wybrania elementu o wadze 11. Teraz suma wag pozostałych elementów ma wynosić 9 (20 minus 11). Spośród dostępnych elementów wybieramy ten o wadze 8. Sumaryczna waga wybranych elementów jest jednak mniejsza od wagi docelowej. Wyznaczamy zatem nową wagę docelową, wynoszącą 1 (9 minus 8). Wybieramy element o wadze 7, lecz jest on zbyt ciężki, więc sprawdzamy pozostałe elementy o wagach 6 i 5, jednak także one są zbyt ciężkie. W tym momencie zbiór dostępnych elementów został wyczerpany, a zatem wiadomo, że nie istnieje kombinacja zawierająca element o wadze 8, której waga sumaryczna będzie wynosić 9. Następnie wybieramy element o wadze 7, a zatem poszukiwana nowa waga docelowa wynosi 2 (9 minus 7). Na tej samej zasadzie wykonywane są kolejne czynności przedstawione poniżej:

```
Elementy: 11, 8, 7 , 6, 5
=============================================
11            // Waga docelowa = 20, 11 to zbyt mało
11, 8         // Waga docelowa = 9, 8 to zbyt mało
11, 8, 7      // Waga docelowa = 1, 7 to zbyt dużo
11, 8, 6      // Waga docelowa = 1, 6 to zbyt dużo
11, 8, 5      // Waga docelowa = 1, 5 to zbyt dużo. Brak elementów
11, 7         // Waga docelowa = 9, 7 to zbyt mało
11, 7, 6      // Waga docelowa = 2, 6 to zbyt dużo
11, 7, 5      // Waga docelowa = 2, 5 to zbyt dużo. Brak elementów
11, 6         // Waga docelowa = 9, 6 to zbyt mało
11, 6, 5      // Waga docelowa = 3, 5 to zbyt dużo. Brak elementów
11, 5         // Waga docelowa = 9, 5 to zbyt mało. Brak elementów
8             // Waga docelowa = 20, 8 to zbyt mało
8, 7          // Waga docelowa = 12, 7 to zbyt mało
8, 7, 6       // Waga docelowa = 5, 6 to zbyt dużo
8, 7, 5       // Waga docelowa = 5, 5 pasuje! Udało się!
```

Jak można się domyślić, rekurencyjna metoda mogłaby wybrać pierwszy element ze zbioru dostępnych elementów, a następnie, jeśli jego waga jest mniejsza od sumarycznej wagi docelowej, wywołać samą siebie, by sprawdzić sumę wag pozostałych dostępnych elementów.

# Kombinacje: Wybieranie zespołu

W matematyce *kombinacja* jest wybraną grupą elementów, których kolejność nie ma znaczenia. Na przykład, załóżmy, że istnieje grupa pięciu alpinistów oznaczonych jako A, B, C, D oraz E. Spośród nich trzeba wybrać grupę trzech, którzy zdobędą stromą i oblodzoną górę Mount Anaconda. Niemniej jednak zachodzą obawy, jak poszczególni członkowie ekipy będą ze sobą współpracować i z tego względu zdecydowaliśmy się sporządzić listę wszystkich możliwych zespołów (czyli wszystkie możliwe kombinacje trzech alpinistów). Wkrótce jednak zmieniliśmy decyzję i teraz chcemy, aby to program komputerowy wygenerował za nas listę zespołów. Taki program wyświetliłby listę 10 możliwych kombinacji:

```
ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE
```

W jaki sposób można by napisać taki program? Okazuje się, że istnieje eleganckie, rekurencyjne rozwiązanie tego problemu. Wiąże się ono z podzieleniem wszystkich kombinacji na dwie grupy: tych które rozpoczynają się od alpinisty A, oraz wszystkich pozostałych. Załóżmy, że zadanie wybrania drużyny trzech osób spośród grupy pięcioosobowej zapiszemy skrótowo (5, 3). Następnie załóżmy, że wielkość grupy oznaczymy przez *n*, a ilość osób w zespole — przez *k*. W takim przypadku nasze twierdzenie ma postać:

$$(n, k) = (n - 1, k - 1) + (n - 1, k)$$

W przedstawionym przykładzie, wymagającym wybrania trzech osób spośród pięciu, uzyskujemy:

$$(5, 3) = (4, 2) + (4, 3)$$

W ten sposób większy problem został podzielony na dwa mniejsze. Zamiast wybierania spośród grupy pięciu osób, przeprowadzane są dwie operacje wyboru z grupy czterech osób. Pierwsza z nich sprowadza się do wyznaczenia wszystkich sposobów wybrania dwóch osób spośród czterech, a druga — do wyznaczenia wszystkich sposobów wybrania trzech osób z grupy czterech osób.

Istnieje sześć sposobów wybrania dwóch osób z grupy czterech osób. A zatem wyrażenie (4, 2), które nazwiemy „wyrażeniem lewym", daje sześć następujących kombinacji:

```
BC, BD, BE, CD, CE, DE
```

Brakującym członkiem grupy jest alpinista A, a zatem, aby uzyskać zespół trzyosobowy, dopisujemy A do uzyskanych wyników:

```
ABC, ABD, ABE, ACD, ACE, ADE
```

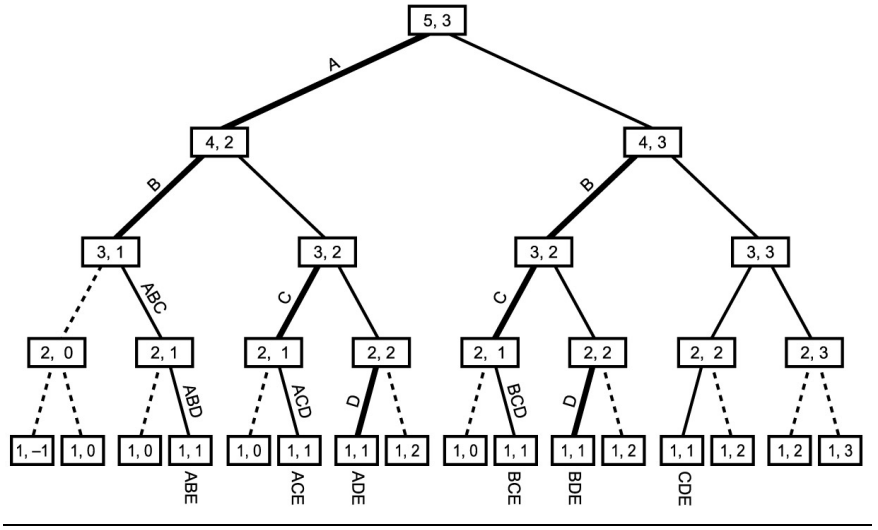
Istnieją cztery sposoby wybrania trzech osób z grupy czterech osób. A zatem wyrażenie (4, 3), które nazwiemy „wyrażeniem prawym", daje cztery kombinacje:

```
BCD, BCE, BDE, CDE
```

Dodając kombinacje uzyskane z wyrażenia prawego z kombinacjami uzyskanymi z wyrażenia lewego, dostajemy dziesięć kombinacji stanowiących rozwiązanie problemu.

Ten sam proces rozbioru można zastosować także do wyznaczenia kombinacji dla obu grup składających się z czterech osób. I tak, (4, 2) to (3, 1) + (3, 2). Jak widać, wykorzystanie rekurencji jest naturalnym sposobem rozwiązania tego problemu.

Przedstawiony tu problem można sobie wyobrazić jako drzewo, w którym w pierwszym wierszu znajduje się wyrażenie (5, 3), w drugim — wyrażenia (4, 3) oraz (4, 2), i tak dalej; przy czym poszczególne węzły drzewa odpowiadają rekurencyjnym wywołaniom metody. Wygląd takiego drzewa dla problemu wyboru trzech osób z grupy pięciu — (5, 3) — został przedstawiony na rysunku 6.20.




RYSUNEK 6.20. Wybieranie trzyosobowego zespołu z grupy pięciu osób

Przypadkiem bazowym omawianego algorytmu są kombinacje nie mające żadnego sensu — takie, w których wybierany zespół lub wielkość grupy wynosi 0, lub w których wielkość wybieranego zespołu jest większa od ilości osób w całej grupie. Kombinacja (1, 1) jest poprawna, lecz jej dalszy rozbiór nie ma sensu. Na przedstawionym rysunku linie kropkowane przedstawiają przypadki bazowe, które powodują zakończenie metody.

Głębokość rekurencyjnych wywołań odpowiada ilości członków grupy. I tak: wierzchołek w górnym wierszu odpowiada alpiniście A, dwa wierzchołki w drugim wierszu odpowiadają alpiniście B i tak dalej. Jeśli grupa liczy pięć osób, drzewo będzie miało pięć poziomów głębokości.

Schodząc „w dół" drzewa, należy pamiętać sekwencję „odwiedzanych" członków grupy. Oto w jaki sposób można to zrealizować: w przypadku wywoływania metody odpowiadającej lewemu wyrażeniu, aktualny wierzchołek jest zapamiętywany poprzez dodanie jego litery do sekwencji. Wywołania odpowiadające lewym wyrażeniom oraz litery dodawane do sekwencji zostały przedstawione na rysunku pogrubioną linią. W czasie powrotu „do góry" konieczne będzie usuwanie kolejnych liter z sekwencji.

Aby zapamiętać wszystkie sekwencje, można je wyświetlać podczas działania algorytmu. Podczas realizacji wywołań odpowiadających lewym wyrażeniom nie są wyświetlane żadne wyniki. Jednak podczas realizacji wywołań odpowiadających prawym wyrażeniom, sekwencja jest sprawdzana, a w przypadku gdy aktualny wierzchołek jest poprawny i gdy dodanie do niego jednej osoby dopełni wybierany zespół, wierzchołek zostaje dodany do sekwencji, a wyznaczony w ten sposób zespół jest wyświetlany.

# Podsumowanie

- Metoda rekurencyjna cyklicznie wywołuje samą siebie, za każdym razem przekazując w wywołaniu inne argumenty.
- Przekazanie argumentu o pewnej wartości powoduje, że metoda kończy działanie bez kolejnego wywołania samej siebie. Sytuacja ta jest nazywana przypadkiem bazowym.
- Gdy najbardziej „wewnętrzne" wywołanie metody zostaje zakończone, rozpoczyna się proces zakańczania kolejnych wywołań, który w końcu doprowadza do zakończenia początkowego wywołania.
- Liczba trójkątna to suma danej liczby oraz wszystkich liczb mniejszych od niej (w tym kontekście *liczba* oznacza *liczbę całkowitą*). Na przykład liczba trójkątna odpowiadająca liczbie 4 ma wartość 10, gdyż 4 + 3 + 2 + 1 wynosi 10.
- Silnia liczby jest iloczynem tej liczby oraz wszystkich liczb mniejszych od niej. Na przykład silnia 4 wynosi 4 ∗ 3 ∗ 2 ∗ 1, czyli 24.
- Zarówno liczby trójkątne, jak i silnie można obliczać za pomocą metod rekurencyjnych lub zwyczajnej pętli.
- Anagram słowa (wszystkie możliwe kombinacje *n* liter tworzących słowo) można wyznaczyć rekurencyjnie poprzez cykliczne przesuwanie wszystkich liter słowa i wyznaczanie anagramu wszystkich *n* – 1 liter słowa z pominięciem jego skrajnej lewej litery.
- Wyszukiwanie binarne można przeprowadzić rekurencyjnie, sprawdzając, w której połówce posortowanego zakresu znajduje się poszukiwany klucz, a następnie wykonując te same czynności dla wyznaczonej połówki.
- Zagadka Wież Hanoi składa się z trzech kolumn oraz dowolnej ilości dysków.
- Zagadkę Wież Hanoi można rozwiązać rekurencyjnie, przenosząc całe poddrzewo dysków, z wyjątkiem największego z nich, na kolumnę pomocniczą, a następnie przenosząc ostatni, największy dysk na kolumnę docelową.
- Scalenie dwóch posortowanych tablic oznacza utworzenie trzeciej tablicy, zawierającej wszystkie elementy scalanych tablicy, które dodatkowo są odpowiednio posortowane.
- W algorytmie sortowania przez scalanie jednoelementowe podtablice stanowiące fragmenty całej tablicy są scalane w podtablice dwuelementowe, które następnie są scalane w podtablice czteroelementowe, i tak dalej. Proces ten jest kontynuowany do momentu posortowania całej tablicy.
- Algorytm sortowania przez scalanie ma złożoność rzędu $O(N * \log N)$.
- Algorytm ten wykorzystuje przestrzeń roboczą, której wielkość odpowiada wielkości sortowanej tablicy.
- Rekurencyjne metody, wyznaczające liczby trójkątne, silnie, anagramy oraz realizujące wyszukiwanie binarne, zawierają tylko jedno rekurencyjne wywołanie samych siebie (w kodzie metody realizującej wyszukiwanie binarne są, co prawda, umieszczone dwa rekurencyjne wywołania, jednak podczas realizacji metody wykonywane jest tylko jedno z nich).
- W przypadku rekurencyjnego algorytmu rozwiązującego problem Wież Hanoi oraz algorytmu sortowania przez scalanie, metoda zawiera dwa rekurencyjne wywołania samej siebie.
- Każdy problem, który można rozwiązać rekurencyjnie, można także rozwiązać z wykorzystaniem stosu.
- Niektóre rozwiązania rekurencyjne mogą być nieefektywne. W takim przypadkach można je zastąpić rozwiązaniami wykorzystującymi zwyczajną pętlę lub stos.

# Pytania

Przedstawione poniżej pytania, należy potraktować jako test, który pozwoli Czytelnikowi sprawdzić wiadomości. Odpowiedzi na postawione pytania można znaleźć w dodatku C.

(1) Jeśli w programie *triangle.java* (listing 6.1) użytkownik wpisze **10**, to jaka jest maksymalna ilość „kopii" metody `triangle()` (a w zasadzie ilość kopii argumentów tej metody)?
(2) Gdzie są przechowywane kopie argumentów, o których była mowa pytaniu 1.?
   (a) W zmiennej zdefiniowanej w metodzie `triangle()`.
   (b) We właściwościach klasy `TriangleApp`.
   (c) W zmiennej zdefiniowanej w metodzie `getString()`.
   (d) Na stosie.
(3) Załóżmy, że w pytaniu 1. użytkownik wpisał wartość 10. Jaka będzie wartość `n`, w momencie gdy metoda `triangle()` zwróci wartość inną niż 1?
(4) Załóżmy, że zachodzi ta sama sytuacja, która została opisana w pytaniu 1. Jaka będzie wartość `n`, w chwili gdy metoda `triangle()` przekaże wartość do metody `main()`?
(5) Prawda czy fałsz: wartości zwracane przez metodę `triangle()` są przechowywane na stosie.
(6) W programie *anagram.java* (listing 6.2) na pewnym etapie rekurencyjnej sekwencji wywołań metoda `doAnagram()` operuje na łańcuchu znaków „`led`". Na jakich literach będzie operować kolejna, rekurencyjnie wywołana „wersja" metody?
(7) Przedstawione przykłady (a konkretnie program *orderedArray.java* z listingu 2.4 wykorzystujący pętlę oraz rekurencyjny program *binarySearch.java* z listingu 6.3) pokazały, że wywołania rekurencyjne można umieszczać wewnątrz pętli. Które z poniższych stwierdzeń *nie* jest prawdziwe?
   (a) Oba ze wspomnianych programów dzieliły tablicę, na której operowały, na połowę.
   (b) Jeśli klucz nie został odnaleziony, to program wykorzystujący pętlę kończy się, gdyż zostanie przekroczony zakres tablicy, natomiast program rekurencyjny — ze względu na dotarcie do najniższego poziomu rekurencyjnych wywołań.
   (c) Jeśli klucz zostanie odnaleziony, to w programie wykorzystującym pętlę kończy się działanie całej metody, natomiast w programie rekurencyjnym — tylko jednego poziomu wywołania.
   (d) W rekurencyjnej wersji metody przeszukiwany zakres należy określić przy użyciu argumentów; natomiast w programie wykorzystującym pętlę nie ma takiej konieczności.
(8) W algorytmie wyszukiwania binarnego, który nie wykorzystuje rekurencji, używana jest pętla. Co zastępuje tę pętlę w metodzie `recFind()` w programie *binarySearch.java* (listing 6.3)?
   (a) Metoda `recFind()`.
   (b) Argumenty metody `recFind()`.
   (c) Rekurencyjne wywołanie metody `recFind()`.
   (d) Wywołanie metody `recFind()` umieszczone w metodzie `main()`.
(9) Program *binarySearch.java* jest przykładem _____ rozwiązywania problemu.
(10) Co staje się coraz mniejsze podczas wykonywania kolejnych rekurencyjnych wywołań metody `recFind()`?
(11) Co staje się coraz mniejsze podczas wykonywania kolejnych rekurencyjnych wywołań metody w programie *towers.java* (listing 6.4)?
(12) W algorytmie stosowanym w programie *towers.java*
   (a) Są wykorzystywane „drzewa" będące urządzeniami służącymi do przechowywania danych.
   (b) Mniejsze dyski są potajemnie umieszczane pod większymi dyskami.
   (c) Kolumna docelowa i pomocnicza są zamieniane.
   (d) Przenoszony jest jeden najmniejszy dysk, a następnie cały stos większych dysków.

(13) Które z poniższych stwierdzeń dotyczących metody `merge()` z programu *merge.java* (listing 6.5) *nie* jest prawdziwe:
   (a) Algorytm zaimplementowany w tej metodzie może obsługiwać tablice różnej wielkości.
   (b) Konieczne jest przeszukanie tablicy docelowej, w celu określenia miejsca, gdzie należy wstawić kolejny element.
   (c) Metoda nie jest rekurencyjna.
   (d) Metoda ciągle pobiera najmniejszy element, niezależnie od tego, w jakiej tablicy jest on zapisany.

(14) Wadą algorytmu sortowania przez scalanie jest:
   (a) Fakt, iż nie jest to algorytm rekurencyjny.
   (b) Wykorzystanie znacznych ilości pamięci.
   (c) Efektywność, która jest większa niż w sortowaniu przez wstawianie, lecz znacznie mniejsza niż w algorytmie quicksort;
   (d) Duża złożoność algorytmu i problemy z jego implementacją.

(15) Oprócz pętli, rekurencję często można także zastępować _____.

# Eksperymenty

Wykonanie poniższych eksperymentów umożliwi dokładne zrozumienie zagadnień omawianych w tym rozdziale. Eksperymenty te nie wymagają pisania jakichkolwiek programów.

(1) W programie *triangle.java* (listing 6.1) należy usunąć przypadek bazowy (fragmenty kodu: `if (n==1)`, `return 1;` oraz `else`), a następnie uruchomić program i sprawdzić, co się stanie.
(2) Korzystając z apletu *Towers Workshop*, należy rozwiązać zagadkę Wież Hanoi, operując na siedmiu lub większej ilości dysków.
(3) Należy zmodyfikować metodę `main()` programu *mergeSort.java* (listing 6.6) w taki sposób, aby tablica mogła być wypełniana setkami tysięcy losowych liczb. Po wprowadzeniu modyfikacji należy uruchomić program w celu posortowania tablicy i porównać szybkość jego działania z algorytmami sortowania przedstawionymi w rozdziale 3., „Proste algorytmy sortowania”.

# Projekty programów

Napisanie programów stanowiących rozwiązanie projektów przedstawionych w tej części rozdziału może utrwalić zrozumienie zagadnień omawianych w tym rozdziale i pokazać, w jaki sposób zagadnienia te mogą być stosowane w praktyce.

(6.1) Załóżmy, że kupiliśmy tani kieszonkowy komputer PC i odkryliśmy, że jego procesor nie jest w stanie wykonywać operacji mnożenia — jedynie dodawanie. W celu rozwiązania tego problemu napisaliśmy rekurencyjną metodę `mult()`, która wykonuje mnożenie liczby *x* i *y*, dodając liczbę *x* do siebie samej *y* razy. Napisz tę metodę oraz metodę `main()`, w której będzie ona wywoływana. Czy operacja dodawania jest wykonywana w momencie rekurencyjnego wywoływania metody, czy też w chwili gdy metoda zwraca wynik?
(6.2) W rozdziale 8., „Drzewa binarne” zostaną przedstawione zagadnienia związane z drzewami binarnymi, w których każda gałąź (potencjalnie) ma dokładnie dwie podgałęzie. Gdyby takie drzewo narysować na ekranie, to w górnym wierszu znalazłaby się jedna gałąź, w następnym

wierszu — dwie, a w kolejnych: 4, 8 16 i tak dalej. Poniżej został przedstawiony wygląd takiego drzewa o szerokości 16 znaków:

```
--------X-------
----X-------X---
--X---X---X---X-
-X-X-X-X-X-X-X-X
XXXXXXXXXXXXXXXX
```

Należy zauważyć, ze dolny wiersz powinien być przesunięty o pół znaku w prawo, jednak wyświetlając drzewo w trybie tekstowym nic na to nie można poradzić.
Takie drzewo można wyświetlić, wykorzystując rekurencyjną metodę `makeBranches()`, do której przekazywane są argumenty `left` oraz `right`, określające krańce poziomego zakresu wyświetlania gałęzi. W momencie pierwszego wywołania metody argument `left` przyjmuje wartość 0, a wartością argumentu `right` jest liczba znaków w wierszu (łącznie ze znakami minusa) pomniejszona o 1. Znak `X` jest wyświetlany w samym środku tego zakresu. Następnie metoda dwukrotnie rekurencyjnie wywołuje samą siebie. Pierwsze z tych wywołań obsługuje lewą połówkę przekazanego zakresu, a drugi — prawą. Wykonywanie metody kończy się, gdy przekazany do niej zakres jest zbyt mały. Prawdopodobnie warto umieszczać wszystkie minusy i `X` w jednej tablicy, a następnie wyświetlać całą jej zawartość, posługując się pewną metodą (na przykład o nazwie `display()`). Napisz metodę `main()`, która będzie wyświetlać drzewo, wykorzystując w tym celu metody `makeBranches()` oraz `display()`. Metoda `main()` powinna dawać możliwość określania szerokości wiersza (na przykład 32, 64 lub dowolnej innej). Dodatkowo należy zagwarantować, że tablica przechowująca wyświetlane wiersze nie będzie większa niż to konieczne. Jak jest zależność pomiędzy ilością wierszy drzewa (w przedstawionym drzewie jest ich pięć) a ich szerokością?

(6.3) Zaimplementuj rekurencyjną metodę podnoszącą liczbę do potęgi, działającą zgodnie z założeniami przedstawionymi w części rozdziału, zatytułowanej „Podnoszenie liczby do potęgi”. Napisz rekurencyjną metodę `power()` oraz metodę `main()`, która umożliwiałaby jej przetestowanie.

(6.4) Napisz program rozwiązujący zagadnienie pakowania plecaka, który umożliwiałby operowanie na plecaku o dowolnej pojemności oraz dowolnej grupie umieszczanych w nim elementów. Należy założyć, że wagi elementów są zapisywane w tablicy. Podpowiedź — argumentami wywołania rekurencyjnej metody `knapsack()` są: waga docelowa oraz indeks określający fragment tablicy, w którym są zapisane pozostałe analizowane elementy.

(6.5) Zaimplementuj rekurencyjny algorytm rozwiązujący problem wygenerowania wszystkich możliwych grup z pewnego zbioru elementów (*n* elementów wybieranych k razy). Napisz rekurencyjną metodę `showTeams()` oraz metodę `main()`, która będzie prosić użytkownika od podanie wielkości zbioru oraz ilości wybieranych elementów, a następnie przekazywać te informacje w wywołaniu metody `showTeams()`. Z kolei metoda `showTeams()` ma wyświetlać wszystkie wygenerowane kombinacje.