Programowanie w języku Java

WYDANIE II

Podejście interdyscyplinarne

Robert Sedgewick • Kevin Wayne

Tytuł oryginału: Introduction to Programming in Java: An Interdisciplinary Approach (2nd Edition)

Tłumaczenie: Paweł Gonera (wstęp, rozdz. 1 – 3, dodatki),
Michał Kępski, Tomasz Walczak (rozdz. 4)

ISBN: 978-83-283-3793-0

Authorized translation from the English language edition, entitled: INTRODUCTION TO PROGRAMMING IN JAVA: AN INTERDISCIPLINARY APPROACH, 2ND EDITION; ISBN 0672337843; by Robert Sedgewick and Kevin Wayne; published by Pearson Education, Inc, publishing as Addison-Wesley Professional. Copyright ©2017 by Pearson Education, Inc.

Ali rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
Polish language edition published by HELION S.A., Copyright © 2018.

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

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

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

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/prjaid.zip

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Spis programów 9

Wstęp 11

1. Elementy programowania 19

1.1. Twój pierwszy program 20
1.2. Wbudowane typy danych 30
1.3. Instrukcje warunkowe i pętle 62
1.4. Tablice 98
1.5. Operacje wejścia i wyjścia 130
1.6. Studium przypadku: losowy surfer sieci WWW 170

2. Funkcje i moduły 189

2.1. Definiowanie funkcji 190
2.2. Biblioteki i klienty 220
2.3. Rekurencja 252
2.4. Studium przypadku: przesiąkanie 286

3. Programowanie obiektowe 311

3.1. Użycie typów danych 312
3.2. Tworzenie typów danych 358
3.3. Projektowanie typów danych 400
3.4. Studium przypadku: symulacja n-ciał 444

4. Algorytmy i struktury danych 457

4.1. Wydajność 458
4.2. Sortowanie i przeszukiwanie 492
4.3. Stosy i kolejki 524
4.4. Tablice symboli 578
4.5. Studium przypadku: zjawisko małego świata 620

Kontekst 661

Słownik 667

Skorowidz 675

API 687

Kup ksi k Pole ksi k

Spis programów

Elementy programowania

Twój pierwszy program

1.1.1. Witaj świecie ... 22
1.1.2. Zastosowanie argumentów wiersza poleceń ... 25

Wbudowane typy danych

1.2.1. Łączenie ciągów znaków ... 35
1.2.2. Mnożenie i dzielenie całkowite ... 38
1.2.3. Równanie kwadratowe ... 40
1.2.4. Rok przestępny ... 43
1.2.5. Rzutowanie w celu uzyskania losowej liczby całkowitej ... 47

Instrukcje warunkowe i pętle

1.3.1. Rzut monetą ... 65
1.3.2. Pierwsza pętla while ... 66
1.3.3. Obliczanie potęg liczby 2 ... 67
1.3.4. Pierwsze zagnieżdżone pętle ... 73
1.3.5. Liczby harmoniczne ... 75
1.3.6. Metoda Newtona ... 76
1.3.7. Konwersja na liczbę binarną ... 78
1.3.8. Symulacja ruiny gracza ... 80
1.3.9. Wyznaczanie czynników pierwszych liczby ... 81

Tablice

1.4.1. Próbkowanie bez zastępowania ... 105
1.4.2. Symulacja kolekcjonera kuponów ... 108
1.4.3. Sito Eratostenesa ... 110
1.4.4. Losowy spacer z samounikaniem ... 117

Operacje wejścia i wyjścia

1.5.1. Generowanie losowej sekwencji ... 132
1.5.2. Interakcyjne dane wejściowe ... 139
1.5.3. Obliczanie średniej wartości ze strumienia liczb ... 140
1.5.4. Prosty filtr ... 144
1.5.5. Standardowy filtr wejścia do rysowania ... 148
1.5.6. Odbijająca się piłka ... 153
1.5.7. Przetwarzanie sygnałów cyfrowych ... 158

Studium przypadku: losowy surfer sieci WWW

1.6.1. Obliczanie macierzy przejść ... 173
1.6.2. Symulacja losowego surfera ... 174
1.6.3. Mieszanie łańcuchów Markowa ... 181

Funkcje i moduły

Definiowanie funkcji

2.1.1. Liczby harmoniczne (kolejne podejście) ... 192
2.1.2. Funkcje Gaussa ... 200
2.1.3. Kolekcjoner kuponów (kolejne podejście) ... 202
2.1.4. Odtwarzanie dźwięku (kolejne podejście) ... 208

Biblioteki i klienty

2.2.1. Biblioteka liczb losowych ... 226
2.2.2. Biblioteka operacji wejścia-wyjścia dla tablic ... 231
2.2.3. Systemy funkcji iteracyjnych ... 233
2.2.4. Biblioteka analizy danych ... 237
2.2.5. Wizualizacja wartości z tablicy ... 238
2.2.6. Próby Bernoulliego ... 240

Rekurencja

2.3.1. Algorytm Euklidesa ... 257
2.3.2. Wieże Hanoi ... 259
2.3.3. Kod Graya ... 264
2.3.4. Grafika rekurencyjna ... 265
2.3.5. Most Browna ... 267
2.3.6. Najdłuższa wspólna podsekwencja ... 274

Studium przypadku: przesiąkanie

2.4.1. Szkielet dla przesiąkania ... 290
2.4.2. Wykrywanie przesiąkania pionowego ... 291
2.4.3. Klient wizualizacji ... 293
2.4.4. Oszacowanie prawdopodobieństwa przesiąkania ... 295
2.4.5. Wykrywanie przesiąkania ... 297
2.4.6. Klient wykresu adaptacyjnego ... 300

Kup ksi k   Pole ksi k

Programowanie obiektowe

Użycie typów danych

3.1.1. Identyfikacja potencjalnego genu 318
3.1.2. Kwadraty Albersa 323
3.1.3. Biblioteka luminancji 326
3.1.4. Konwersja koloru na skalę szarości 329
3.1.5. Skalowanie obrazów 331
3.1.6. Efekt przenikania 332
3.1.7. Łączenie plików 336
3.1.8. Zbieranie z ekranu danych dotyczących kursów walut 338
3.1.9. Dzielenie pliku 339

Tworzenie typów danych

3.2.1. Naładowana cząstka 363
3.2.2. Stoper 366
3.2.3. Histogram 368
3.2.4. Grafika żółwia 371
3.2.5. Spira mirabilis 373
3.2.6. Liczby zespolone 377
3.2.7. Zbiór Mandelbrota 382
3.2.8. Konto giełdowe 385

Projektowanie typów danych

3.3.1. Liczby zespolone (podejście alternatywne) 405
3.3.2. Licznik 408
3.3.3. Wektor przestrzenny 414
3.3.4. Szkic dokumentu 429
3.3.5. Wykrywanie podobieństw 431

Studium przypadku: symulacja n-ciał

3.4.1. Ciało grawitacyjne 446
3.4.2. Symulacja n-ciał 449

Algorytmy i struktury danych

Wydajność

4.1.1. Problem ThreeSum 461
4.1.2. Walidacja hipotezy podwojenia 463

Sortowanie i przeszukiwanie

4.2.1. Wyszukiwanie binarne (gra w 20 pytań) 494
4.2.2. Metoda równego podziału 497
4.2.3. Wyszukiwanie binarne (dla uporządkowanej tablicy) 498
4.2.4. Sortowanie przez wstawianie 505
4.2.5. Program testujący hipotezę podwajania 507
4.2.6. Sortowanie przez scalanie 510
4.2.7. Zliczanie wystąpień 515

Stosy i kolejki

4.3.1. Stos ciągów znaków (implementacja tablicowa) 528
4.3.2. Stos ciągów znaków (implementacja za pomocą listy połączonej) 533
4.3.3. Stos ciągów znaków (zmienny rozmiar tablicy) 536
4.3.4. Stos generyczny 541
4.3.5. Obliczanie wartości wyrażeń 544
4.3.6. Generyczna kolejka FIFO (z wykorzystaniem listy połączonej) 550
4.3.7. Symulacja kolejki M/M/1 554
4.3.8. Symulacja równoważenia obciążenia (ang. load balancing) 561

Tablice symboli

4.4.1. Przeszukiwanie słownika 585
4.4.2. Indeksowanie 587
4.4.3. Tablica mieszająca 592
4.4.4. Binarne drzewo poszukiwań 599
4.4.5. Filtr usuwający duplikaty 606

Studium przypadku: zjawisko małego świata

4.5.1. Typ danych Graph 627
4.5.2. Używanie grafu do odwracania indeksu 630
4.5.3. Klient wyznaczający najkrótsze ścieżki 635
4.5.4. Implementacja wyszukiwania najkrótszych ścieżek 639
4.5.5. Sprawdzanie, czy graf reprezentuje mały świat 644
4.5.6. Graf aktor-aktor 646

Kup książkę Poleć książkę

1.4. Tablice

W TYM PODROZDZIALE PRZEDSTAWIMY KONCEPCJĘ **STRUKTUR DANYCH** ORAZ PIERWSZĄ strukturę danych — **tablicę**. Głównym zastosowaniem tablic jest przechowywanie dużych ilości danych i manipulowanie nimi. Tablice grają ważną rolę w wielu zadaniach przetwarzania danych. Odpowiadają one wektorom i macierzom, które są powszechnie wykorzystywane w nauce i programach naukowych. Przedstawimy podstawowe właściwości tablic w języku Java oraz wiele przykładów ich użyteczności.

1.4.1. Próbkowanie bez zastępowania 105
1.4.2. Symulacja kolekcjonera kuponów 108
1.4.3. Sito Erastonesa 110
1.4.4. Losowy spacer z samounikaniem 117

Programy w tym podrozdziale

Struktura danych jest sposobem organizacji danych w komputerze (zwykle w celu zaoszczędzenia czasu lub przestrzeni). Struktury danych pełnią ważną rolę w programowaniu — cały rozdział 4. tej książki jest poświęcony różnym rodzajom klasycznych struktur danych.

a

a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]

Rysunek 1.24. Tablica a

Tablica jednowymiarowa (lub po prostu **tablica**) jest strukturą danych przechowującą **sekwencję** wartości tego samego typu. Komponenty tej tablicy nazywany **elementami**. Do wskazywania elementów tablicy wykorzystujemy **indeksowanie**: jeżeli w tablicy znajduje się n elementów, numerujemy je kolejno od 0 do n–1, dzięki czemu możemy w jednoznaczny sposób wskazać element, podając jego indeks.

Tablica dwuwymiarowa jest tablicą tablic jednowymiarowych. Ponieważ tablicę jednowymiarową można indeksować za pomocą jednej liczby, to elementy tablicy dwuwymiarowej można indeksować przy użyciu pary liczb: pierwszy indeks określa wiersz, a drugi indeks oznacza kolumnę.

Często, gdy mamy do przetworzenia dużo danych, umieszczamy je na początek w tablicy lub tablicach. Następnie je przetwarzamy, czyli wykorzystujemy indeksowanie, odwołując się do poszczególnych elementów. Mogą być to wyniki egzaminów, ceny na akcji, nukleotydy w łańcuchu DNA czy też znaki w książce. Każdy z tych przykładów wymaga pracy na wielu wartościach tego samego typu. Tego typu aplikacje przedstawimy w czasie omawiania operacji wejścia-wyjścia w podrozdziale 1.5 oraz w studiach przypadków w podrozdziale 1.6. W bieżącym podrozdziale opiszemy podstawowe cechy tablic na podstawie programów, które na początek wypełniają tablice wartościami eksperymentów, a następnie je przetwarzają.

Tablice w języku Java

Utworzenie tablicy w programie Java wymaga wykonania trzech kroków:

- zadeklarowania tablicy,
- utworzenia tablicy,
- zainicjalizowania elementów tablicy.

Kup ksi k Pole ksi k

Aby zadeklarować tablicę, należy określić jej nazwę oraz typ danych, jaki będzie przechowywać. Żeby ją utworzyć, należy określić jej **wielkość** (liczbę elementów). By ją zainicjalizować, należy przypisać wartość do każdego z elementów. Przykładowo poniższy kod tworzy tablicę n elementów typu double i inicjalizuje je wartością 0.0:

```
double[] a;                      // Deklaracja tablicy.
a = new double[n];               // Tworzenie tablicy.
for (int i = 0; i < n; i++)      // Inicjalizowanie tablicy.
   a[i] = 0.0;
```

Pierwsza instrukcja jest **deklaracją tablicy**. Wygląda ona jak deklaracja zmiennej odpowiedniego typu prostego, ale ma dodatkowo nawiasy klamrowe po nazwie typu, które wskazują na deklarację tablicy. Druga instrukcja powoduje **utworzenie tablicy**; wykorzystuje słowo kluczowe new do przydzielenia pamięci mieszczącej podaną liczbę elementów. Operacja ta nie jest potrzebna dla zmiennych typów prostych, ale niezbędna dla pozostałych typów danych w języku Java (podrozdział 3.1). Za pomocą pętli for przypisujemy wartość 0.0 do każdego z n elementów tablicy. Do elementu tablicy odwołujemy się, umieszczając indeks w nawiasach kwadratowych po nazwie tablicy: kod a[i] pozwala na odwołanie się do elementu i tablicy a[] (w tekście książki używamy notacji a[] do wskazania, że zmienna a jest tablicą, ale w kodzie Java nie używamy a[]).

Oczywistą zaletą stosowania tablic jest możliwość zdefiniowania wielu zmiennych bez jawnego ich nazywania. Jeżeli np. chcemy przetwarzać osiem zmiennych typu double, można zadeklarować je za pomocą:

```
double a0, a1, a2, a3, a4, a5, a6, a7;
```

i odwoływać się do nich za pomocą a0, a1, a2 itd. Nazywanie kilkunastu zmiennych w ten sposób jest nużące, a nazywanie milionów niemożliwe do zrobienia. Zamiast tego można wykorzystać tablicę i zadeklarować n zmiennych za pomocą instrukcji double[] a = new double[n] i odwoływać się do nich w taki sposób: a[0], a[1], a[2] itd. Teraz zdefiniowanie kilkunastu czy milionów zmiennych jest równie łatwe. Co więcej, ponieważ możliwe jest użycie zmiennej (lub wyrażenia obliczanego w czasie działania programu) jako indeksu tablicy, to możliwe jest przetwarzanie dowolnej liczby elementów za pomocą jednej pętli. Powinieneś myśleć o każdym z elementów tablicy jak o osobnej zmiennej, której można użyć w wyrażeniu lub po lewej stronie instrukcji przypisania.

W pierwszym przykładzie użyjemy tablicy do reprezentowania **wektora**. Wektory będziemy omawiać szczegółowo w podrozdziale 3.3; na ten moment potraktujmy wektor jako sekwencję liczb rzeczywistych. **Iloczyn skalarny** dwóch wektorów (tej samej długości) jest sumą iloczynów odpowiednich elementów. Iloczyn skalarny dwóch wektorów reprezentowanych jako tablice jednowymiarowe x[] oraz y[] o długości 3 to wyrażenie x[0]*y[0] + x[1]*y[1] + x[2]*y[2]. Uogólniając, powiedzmy, że jeżeli każda tablica ma długość n, poniższy kod pozwala wyliczyć ich iloczyn skalarny:

```
double sum = 0.0;
for (int i = 0; i < n; i++)
   sum += x[i]*y[i];
```

Kup ksi k Pole ksi k

Tabela 1.29. Tabela śladu obliczania iloczynu skalarnego

i	x[i]	y[i]	x[i]*y[i]	sum
				0.00
0	0.30	0.50	0.15	0.15
1	0.60	0.10	0.06	0.21
2	0.10	0.40	0.04	0.25
				0.25

Łatwość realizacji takich obliczeń powoduje, że tablice są naturalnym wyborem dla wielu zastosowań.

W TABELI 1.30 UMIESZCZONE SĄ PRZYKŁADY KODU przetwarzającego tablice; więcej przykładów znajduje się dalej w tej książce, ponieważ tablice pełnią ważną rolę w przetwarzaniu danych w wielu aplikacjach. Przed analizą bardziej skomplikowanych przykładów przedstawiamy kilka ważnych aspektów programowania z wykorzystaniem tablic.

Tabela 1.30. Typowy kod przetwarzający tablice (dla tablicy a[] z n wartości double)

tworzenie tablicy z wartościami losowymi

```
double[] a = new double[n];
for (int i = 0; i < n; i++)
   a[i] = Math.random();
```

wyświetlanie wartości tablicy, po jednej w wierszu

```
for (int i = 0; i < n; i++)
   System.out.println(a[i]);
```

znajdowanie maksymalnej wartości w tablicy

```
double max = Double.NEGATIVE_INFINITY;
for (int i = 0; i < n; i++)
   if (a[i] > max) max = a[i];
```

obliczanie średniej z wartości w tablicy

```
double sum = 0.0;
for (int i = 0; i < n; i++)
   sum += a[i];
double average = sum / n;
```

odwracanie kolejności wartości w tablicy

```
for (int i = 0; i < n/2; i++)
{
   double temp = a[i];
   a[i] = a[n-1-i];
   a[n-i-1] = temp;
}
```

kopiowanie sekwencji wartości do innej tablicy

```
double[] b = new double[n];
for (int i = 0; i < n; i++)
   b[i] = a[i];
```

Indeksowanie od zera

Pierwszym elementem tablicy `a[]` jest `a[0]`, drugim elementem jest `a[1]` itd. Być może bardziej naturalne dla nas jest odwoływanie się do pierwszego elementu poprzez `a[1]`, do drugiego poprzez `a[2]` itd., ale zaczynanie indeksowania od zera jest konwencją wykorzystywaną w większości nowoczesnych języków programowania. Brak stosowania się do tej konwencji często prowadzi do błędów przesunięcia o jeden, które są bardzo trudne do uniknięcia i znalezienia, więc miej się na baczności!

Długość tablicy

Gdy utworzymy tablicę w języku Java, jej długość jest stała. Jednym z powodów konieczności jawnego tworzenia tabel w czasie działania programu jest fakt, że kompilator Java „nie wie” w czasie kompilacji, ile przestrzeni powinien zarezerwować na tablicę

Kup ksi k Pole ksi k

(ponieważ jej długość może być znana dopiero w czasie działania programu). Nie musisz jawnie przydzielać pamięci na zmienne typu `int` czy `double`, ponieważ ich wielkość jest stała i znana w czasie kompilacji. Możesz użyć kodu `a.length` do odczytania długości tablicy `a[]`. Zauważ, że ostatni element tablicy `a[]` to zawsze `a[a.length-1]`. Dla ułatwienia często przechowujemy długość tablicy w zmiennej całkowitej `n`.

Domyślna inicjalizacja tablicy

Aby nie tworzyć niepotrzebnego kodu, często korzystamy z **domyślnej inicjalizacji tablicy** przez język Java do zadeklarowania i zainicjalizowania tablicy za pomocą jednej instrukcji. Przykładowo poniższa instrukcja jest odpowiednikiem kodu z początku podpunktu „Tablice w języku Java":

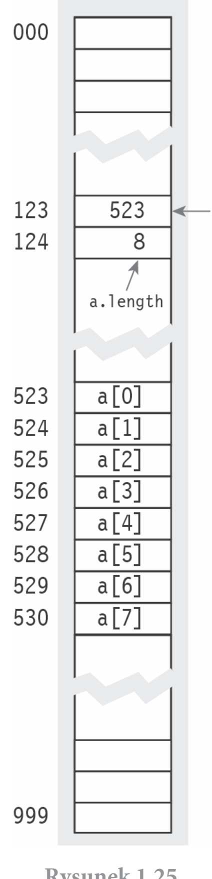
```
double[] a = new double[n];
```

Kod znajdujący się po lewej stronie znaku równości jest deklaracją, natomiast kod z prawej strony jest odpowiedzialny za utworzenie tablicy. Pętla `for` jest niepotrzebna w tym przypadku, ponieważ Java zawsze inicjalizuje elementy tablicy każdego typu prostego wartością zero (dla typów numerycznych) lub `false` (dla typu `boolean`). Java automatycznie inicjalizuje elementy tablicy typu `String` (i typów innych niż proste) specjalną wartością `null`, którą przedstawimy w rozdziale 3.

Reprezentacja w pamięci

Tablice są tak podstawową strukturą danych, że w niemal wszystkich komputerach mają bezpośrednie odwzorowanie w systemach obsługi pamięci. Elementy tablicy są przechowywane kolejno w pamięci, dzięki czemu można szybko odnaleźć dowolną wartość. W zasadzie możemy traktować pamięć komputera jak jedną gigantyczną tablicę. W nowoczesnych komputerach pamięć jest implementowana sprzętowo jako sekwencja lokalizacji pamięci, z których każda jest dostępna za pomocą jej indeksu. Kiedy mówimy o pamięci komputera, indeks lokalizacji zwykle nazywamy **adresem**. Możemy uważać nazwę tablicy — np. `a` — za adres pierwszego elementu tablicy `a[0]`. Na potrzeby ilustracji załóżmy, że pamięć komputera składa się z 1000 wartości o adresach od `000` do `999` (ten uproszczony model pomija fakt, że elementy tablicy mogą zajmować różną ilość pamięci, w zależności od ich typu, ale na razie możemy zignorować ten szczegół). Teraz załóżmy, że tablica ośmioelementowa jest zapisana w pamięci w lokalizacjach od 523 do 530. W takiej sytuacji Java zapisze adres pamięci (indeks) pierwszego elementu tablicy w innej lokalizacji pamięci, wraz z długością tablicy. Adres taki nazywamy **wskaźnikiem**; ma on za zadanie **wskazywać na** odpowiednią lokalizację w pamięci. Gdy użyjemy `a[i]`, kompilator wygeneruje kod odwołujący się do żądanej wartości poprzez dodanie indeksu `i` do adresu pamięci tablicy `a[]`. Przykładowo `a[4]` wygeneruje kod maszynowy odczytujący wartość z lokalizacji pamięci `523 + 4 = 527`. Dostęp do elementu `i` tablicy jest bardzo efektywną operacją, ponieważ wymaga tylko dodania dwóch wartości całkowitych, a następnie odwołania się do pamięci — to tylko dwie operacje elementarne.

Kup ksi k Pole ksi k



Rysunek 1.25. Reprezentacja w pamięci

Przydział pamięci

Gdy używamy słowa kluczowego `new` do utworzenia tablicy, Java rezerwuje odpowiedni fragment pamięci do przechowywania podanej liczby elementów. Proces ten jest nazywany **przydziałem pamięci**. To ten sam proces, który jest wymagany dla wszystkich zmiennych używanych w programie (ale dla typów prostych nie stosujemy słowa kluczowego `new`, ponieważ Java „wie", ile pamięci należy przydzielić). Skupimy się teraz na tym procesie, ponieważ my ponosimy odpowiedzialność za utworzenie tablicy przed dostępem do jej elementów. Jeżeli nie spełnisz tego warunku, w czasie wykonania otrzymasz błąd **niezainicjalizowanej zmiennej**.

Kontrola granic

Jak już wspominaliśmy, programując z użyciem tablic, należy zachować uwagę. Naszym zadaniem jest używanie prawidłowych indeksów przy odwoływaniu się do elementów tablicy. Jeżeli utworzyłeś tablicę o długości `n` i używasz indeksu, którego wartość jest mniejsza od `0` lub większa od `n – 1`, program zostanie przerwany przez wyjątek `ArrayIndexOutOfBo` ↪`undsException`. (W wielu językach programowania takie przykłady **przepełnienia bufora** nie są sprawdzane przez system. Te niesygnalizowane błędy mogą prowadzić do koszmarów debugowania, a nierzadko pozostają niezauważone w gotowym programie. Możesz być zaskoczony, ale tego typu błędy mogą być wykorzystane przez hakera do przejęcia kontroli nad systemem, nawet Twoim własnym, w celu rozsiewania wirusów, kradzieży tożsamości lub innych szkodliwych działań). Komunikat błędu generowany przez Javę może początkowo wydawać się irytujący, ale jest to niewielka cena za możliwość zbudowania bezpieczniejszego programu.

Ustawianie wartości tablicy w czasie kompilacji

Gdy do przechowania w tablicy mamy niewielką liczbę wartości, możemy zadeklarować, utworzyć i zainicjalizować tablicę przez umieszczenie wartości w nawiasach klamrowych, rozdzielając je przecinkami. Przykładowo w programie operującym na kartach do gry możemy użyć poniższego kodu:

```
String[] SUITS = { "Trefl", "Karo", "Kier", "Pik" };
String[] RANKS =
{
   "2", "3", "4", "5", "6", "7", "8", "9", "10",
   "Walet", "Dama", "Król", "As"
};
```

Teraz możemy skorzystać z tych dwóch tablic do wyświetlania losowych nazw kart, np. *Dama Karo*:

```
int i = (int) (Math.random() * RANKS.length);
int j = (int) (Math.random() * SUITS.length);
System.out.println(RANKS[i] + " " + SUITS[j]);
```

Kup ksi k Pole ksi k

Kod ten wykorzystuje mechanizm wprowadzony w podrozdziale 1.2, który pozwala na wygenerowanie losowych indeksów, następnie użytych do wybrania ciągów znaków z dwóch tablic. Jeżeli wartości wszystkich elementów tablicy są znane (i nie jest to zbyt duża tablica), użycie tej metody inicjalizacji jest sensowne — po prostu umieść wszystkie wartości w nawiasach klamrowych po prawej stronie znaku równości w deklaracji tablicy. Powoduje to utworzenie tablicy, więc słowo kluczowe `new` nie jest potrzebne.

Ustawianie wartości tablicy w czasie działania programu

W większości przypadków chcemy obliczyć wartości, które mają być zapisane w tablicy. Wtedy możemy użyć nazwy tablicy z indeksem tak samo, jak używamy nazwy zmiennej po lewej stronie instrukcji przypisania. Przykładowo poniższy kod inicjalizuje tablicę o długości 52, reprezentującą talię kart, wykorzystując wcześniej zdefiniowane tablice:

```
String[] deck = new String[RANKS.length * SUITS.length];
for (int i = 0; i < RANKS.length; i++)
   for (int j = 0; j < SUITS.length; j++)
      deck[SUITS.length*i + j] = RANKS[i] + " " + SUITS[j];
```

Po wykonaniu tego kodu możemy wyświetlić zawartość `deck[]` w kolejności od `deck[0]` do `deck[51]` i otrzymamy następujący wynik:

```
2 Trefl
2 Karo
2 Kier
2 Pik
3 Trefl
3 Karo
...
As Kier
As Pik
```

Wymiana dwóch wartości w tablicy

Często spotykaną sytuacją jest konieczność wymiany wartości dwóch elementów tablicy. Kontynuujmy zatem nasz przykład z talią kart. Poniższy kod wymienia karty pod indeksami `i` oraz `j`, stosując ten sam idiom, jaki zastosowaliśmy w pierwszym przykładzie użycia przypisania w podrozdziale 1.2:

```
String temp = deck[i];
deck[i] = deck[j];
deck[j] = temp;
```

Jeżeli np. użyjemy tego kodu, przypisując 1 do `i` oraz 4 do `j`, w tablicy `deck[]` z poprzedniego przykładu w `deck[1]` znajdzie się *3 Trefl*, a w `deck[4]` znajdzie się *2 Kier*. Możesz również sprawdzić, że kod ten nie zmienia zawartości tablicy, jeżeli `i` oraz `j` są równe. Korzystając z tego kodu, możemy być pewni, że zmienia on **porządek** wartości w tablicy, ale nie **zbiór** wartości tablicy.

Kup ksi k Pole ksi k

Tasowanie tablicy

Poniższy kod tasuje naszą talię kart:

```
int n = deck.length;
for (int i = 0; i < n; i++)
{
   int r = i + (int) (Math.random() * (n-i));
   String temp = deck[i];
   deck[i] = deck[r];
   deck[r] = temp;
}
```

Zaczynając od lewej do prawej, wybieramy losowo kartę między `deck[i]` a `deck[n-1]` (każda karta jest równie prawdopodobna) i wymieniamy ją z `deck[i]`. Kod ten jest bardziej złożony, niż może się wydawać: po pierwsze, upewniamy się, że karty w talii po potasowaniu są takie same jak przed potasowaniem, ponieważ używamy znanego mechanizmu wymiany. Po drugie, upewniamy się, że tasowanie jest losowe przez wybór z niewybranych jeszcze kart.

Próbkowanie bez zastępowania

We wielu sytuacjach chcemy uzyskać losową próbkę ze zbioru w taki sposób, że każdy składnik zbioru występuje w próbce nie więcej niż jeden raz. Przykładem takiego losowania jest wyciąganie numerowanych piłeczek pingpongowych z koszyka lub rozdawanie kart z talii. Program `Sample` (listing 1.4.1) ilustruje sposób próbkowania z użyciem podstawowych operacji wykorzystywanych w tasowaniu. Program odczytuje z wiersza poleceń argumenty `m` i `n`, a następnie tworzy **permutację** o długości `n` (zbiór liczb całkowitych od `0` do `n`–1)), której pierwsze `m` elementów tworzy losową próbkę. Dołączona tabela śladu przedstawia zawartość tablicy `perm[]` na końcu iteracji głównej pętli (dla wykonania programu o wartościach `m` i `n` — odpowiednio — 6 i 16).

i	r	perm[]															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	9	9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15
1	5	9	5	2	3	4	1	6	7	8	0	10	11	12	13	14	15
2	13	9	5	13	3	4	1	6	7	8	0	10	11	12	2	14	15
3	5	9	5	13	1	4	3	6	7	8	0	10	11	12	2	14	15
4	11	9	5	13	1	11	3	6	7	8	0	10	4	12	2	14	15
5	8	9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15
		9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15

Rysunek 1.26. Tabela śladu wykonania java Sample 6 16

Jeżeli wartości `r` są wybrane w taki sposób, że każda wartość w danym zakresie jest równie prawdopodobna, to po zakończeniu procesu elementy od `perm[0]` do `perm[m-1]` są jednolitą próbką losową (mimo że niektóre wartości mogły być przenoszone

Kup ksi k Pole ksi k

Listing 1.4.1. Próbkowanie bez zastępowania

```
public class Sample
{
   public static void main(String[] args)
   {  // Wyświetla losową próbkę m liczb całkowitych
      // od 0 … n-1 (bez powtórzeń).
      int m = Integer.parseInt(args[0]);
      int n = Integer.parseInt(args[1]);
      int[] perm = new int[n];
      // Inicjalizowanie perm[].
      for (int j = 0; j < n; j++)
         perm[j] = j;
      // Pobranie próbki.
      for (int i = 0; i < m; i++)
      {  // Wymiana perm[i] z losowym elementem po prawej.
         int r = i + (int) (Math.random() * (n - i));
         int t = perm[r];
         perm[r] = perm[i];
         perm[i] = t;
      }
      // Wyświetlenie próbki.
      for (int i = 0; i < m; i++)
         System.out.print(perm[i] + " ");
      System.out.println();
   }
}
```

m	wielkość próbki
n	zakres
perm[]	permutacje od 0 do n-1

Program odczytuje z wiersza poleceń dwa argumenty `m` i `n`, a następnie tworzy próbkę `m` liczb od `0` do n–1. Proces ten jest przydatny nie tylko w loteriach, ale również w aplikacjach naukowych. Jeżeli pierwszy argument jest równy drugiemu, wynikiem jest losowa permutacja liczb całkowitych od `0` do n–1. Jeżeli pierwszy argument jest większy od drugiego, program zostanie przerwany wyjątkiem `ArrayOutOfBoundsException`.

```
% java Sample 6 16
9 5 13 1 11 8
% java Sample 10 1000
656 488 298 534 811 97 813 156 424 109
% java Sample 20 20
6 12 9 8 13 19 0 2 4 5 18 1 14 16 17 3 7 11 10 15
```

wiele razy), ponieważ każdemu elementowi próbki jest przypisywana wartość losowa z dotychczas niepróbkowanych wartości. Ważnym powodem jawnego obliczania permutacji jest możliwość wyświetlenia losowej próbki **dowolnej** tablicy przez użycie elementów permutacji jako indeksów tabeli. Jest to atrakcyjną alternatywą dla faktycznego przenoszenia elementów tablicy, ponieważ w pewnych zastosowaniach muszą one pozostać w określonym porządku (np. firma chce uzyskać losową próbkę z listy klientów przechowywanej w porządku alfabetycznym).

Aby zobaczyć, jak działa ten mechanizm, przeanalizujmy losowanie kart do pokera z tablicy `deck[]` utworzonej w opisany wcześniej sposób. Możemy użyć kodu z programu `Sample` dla `n = 52` i `m = 5` i zamienić w instrukcji `System.out.print()` `perm[i]` na `deck[perm[i]]` (zamieniając ją na `println()`), w wyniku czego otrzymamy następujący wynik:

```
3 Trefl
Walet Kier
6 Pik
As Trefl
10 Karo
```

Tego typu próbkowanie jest powszechnie stosowane jako podstawa dla opracowań statystycznych przy ankietowaniu, badaniach naukowych i wielu innych zastosowaniach, w których chcemy uzyskać wnioski dotyczące dużej populacji przez analizowanie losowej próbki.

Wstępnie wyliczone wartości

Jednym z prostych zastosowań tablic jest przechowywanie obliczonych wartości do późniejszego zastosowania. Dla przykładu załóżmy, że piszemy program wykonujący obliczenia z użyciem niewielkiej ilości liczb harmonicznych (listing 1.3.5). Efektywnym podejściem jest zapisanie tych wartości w tablicy w następujący sposób:

```
double[] harmonic = new double[n];
for (int i = 1; i < n; i++)
   harmonic[i] = harmonic[i-1] + 1.0/i;
```

Następnie możesz użyć w kodzie `harmonic[i]`, gdy potrzebujesz odwołać się do `i`-tej liczby harmonicznej. Wstępne obliczenia tego typu są przykładem **kompromisu czasu i miejsca**: inwestując miejsce (do zapisania wartości), możemy zaoszczędzić czas (ponieważ nie musimy ich ponownie obliczać). Metoda ta nie jest efektywna, jeżeli potrzebujesz dużej liczby wartości, ale jest bardzo efektywna, kiedy potrzebujesz wiele razy niewielkiej liczby wartości.

Upraszczanie powielanego kodu

Przykładem innego prostego zastosowania tablic może być uproszczenie poniższego fragmentu kodu, który wyświetla nazwę miesiąca na podstawie jego numeru (1 dla stycznia, 2 dla lutego itd.):

```
if      (m ==  1) System.out.println("Sty");
else if (m ==  2) System.out.println("Lut");
else if (m ==  3) System.out.println("Mar");
else if (m ==  4) System.out.println("Kwi");
else if (m ==  5) System.out.println("Maj");
else if (m ==  6) System.out.println("Cze");
else if (m ==  7) System.out.println("Lip");
else if (m ==  8) System.out.println("Sie");
else if (m ==  9) System.out.println("Wrz");
else if (m == 10) System.out.println("Paź");
else if (m == 11) System.out.println("Lis");
else if (m == 12) System.out.println("Gru");
```

Kup ksi k Pole ksi k

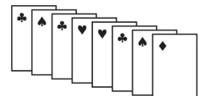
Możemy również użyć instrukcji `switch`, ale znacznie bardziej zwartą alternatywą jest zastosowanie tablicy ciągów składającej się z nazw poszczególnych miesięcy:

```
String[] MONTHS =
{
   "","Sty", "Lut", "Mar", "Kwi", "Maj", "Cze",
      "Lip", "Sie", "Wrz", "Paź", "Lis", "Gru"
};
System.out.println(MONTHS[m]);
```

Technika ta jest szczególnie użyteczna, kiedy musisz korzystać z nazw miesięcy za pomocą ich numerów w kilku miejscach w programie. Zwróć uwagę, że rozmyślnie zmarnowaliśmy jeden slot tablicy (element 0), aby `MONTHS[1]` odpowiadało styczniowi.

PO PRZEDSTAWIENIU TYCH PODSTAWOWYCH DEFINICJI I PRZYKŁADÓW MOŻEMY ZADEMONSTROWAĆ dwie aplikacje rozwiązujące klasyczne problemy i ilustrujące wagę tablic w efektywnych obliczeniach. W obu przykładach centralną rolę pełni wyrażenie do indeksowania tablicy, pozwalające na wykonanie obliczeń, które do tej pory nie były możliwe.

Kolekcjoner kuponów



Rysunek 1.27. Zbieranie kuponów

Załóżmy, że masz talię kart i jedna po drugiej wybierasz losowo karty (z zastępowaniem). Ile kart musisz wybrać, zanim zobaczysz po jednej karcie z każdego koloru? Ile kart musisz wybrać, zanim zobaczysz każdą z wartości? Są to przykłady znanego problemu **kolekcjonera kuponów**. Załóżmy, że firma handlowa wypuszcza karty kolekcjonerskie składające się z `n` różnych kart. Ile kart musisz zebrać, aby uzyskać wszystkie `n` możliwości, przy założeniu, że prawdopodobieństwo uzyskania każdej karty jest takie samo?

Zbieranie kuponów nie jest sztucznym problemem. Przykładowo naukowcy często chcą wiedzieć, czy sekwencja występująca w naturze ma taką samą charakterystykę jak sekwencja losowa. Jeżeli tak, fakt ten może być interesujący, jeżeli nie, mogą być konieczne dalsze badania w celu wyszukania interesujących wzorców. Tego typu testy są wykorzystywane przez naukowców w celu zdecydowania, które części genomu są warte analizy. Jedną z metod efektownego testowania, czy sekwencja jest naprawdę losowa, jest **test kolekcjonera kuponów**: porównaj liczbę elementów potrzebnych do przeanalizowania, zanim zostaną znalezione wszystkie wartości określonego typu w jednorodnie losowej sekwencji. Program `CouponCollector` (listing 1.4.2) jest programem symulującym ten proces i ilustrującym użycie tablic. Odczytuje on z wiersza poleceń argument `n` i generuje sekwencję całkowitych liczb losowych pomiędzy `0` a `n`–1 za pomocą instrukcji `(int)(Math.random() * n)` — patrz listing 1.2.5. Każda liczba reprezentuje kartę; dla każdej z nich chcemy wiedzieć, czy już ją widzieliśmy. Aby można było to stwierdzić, używamy tablicy `isCollected[]`, w której indeksem jest numer karty. Element `isCollected[i]` ma wartość `true`, jeżeli widzieliśmy kartę `i`, lub `false`, jeżeli jej nie widzieliśmy. Gdy otrzymujemy nową kartę reprezentowaną przez liczbę `r`, sprawdzamy, czy ją widzieliśmy, przez odczytanie wartości `isCollected[r]`. Kod śledzi liczbę widzianych rodzajów kart oraz liczbę wygenerowanych kart i wyświetla ostatnią wartość, gdy wcześniejsza osiągnie `n`.

Listing 1.4.2. Symulacja kolekcjonera **kuponów**

```
public class CouponCollector
{
   public static void main(String[] args)
   {
      // Generowanie losowej wartości z zakresu [0..n) do momentu znalezienia każdej z nich.
      int n = Integer.parseInt(args[0]);
      boolean[] isCollected = new boolean[n];
      int count = 0;
      int distinct = 0;
      while (distinct < n)
      {
         // Generowanie kolejnego kuponu.
         int r = (int) (Math.random() * n);
         count++;
         if (!isCollected[r])
         {
            distinct++;
            isCollected[r] = true;
         }
      } // Znaleziono n różnych kuponów.
      System.out.println(count);
   }
}
```

`n`	liczba wartości kuponów (od 0 do n-1)
`isCollected[i]`	czy kupon `i` został zebrany?
`count`	liczba wygenerowanych kuponów
`distinct`	liczba różnych kuponów
`r`	losowy kupon

Program ten odczytuje z wiersza poleceń argument `n` i symuluje zbieranie kuponów przez wygenerowanie liczby losowej z zakresu od `0` do `n–1`, do momentu zebrania każdej z możliwych wartości.

```
% java CouponCollector 1000
6583
% java CouponCollector 1000
6477
% java CouponCollector 1000000
12782673
```

Jak zwykle, najlepszym sposobem na zrozumienie programu jest analiza tabeli śladu z wartościami zmiennych w czasie typowego wywołania. Łatwo możemy dodać do programu `CouponCollector` kod generujący tabelę śladu zawierającą wartości na końcu pętli `while`. Na rysunku 1.28 użyliśmy `F` dla wartości `false` i `T` dla wartości `true`, aby tabela śladu była czytelniejsza. Śledzenie programu korzystającego z dużych tablic może być wyzwaniem: gdy mamy w naszym programie tablicę o długości `n`, reprezentuje ona `n` zmiennych, więc musimy wyświetlić je wszystkie. Śledzenie programów korzystających z `Math.random()` może być również wyzwaniem, ponieważ przy każdym wykonaniu programu otrzymujemy inne tabele śladu. Konieczne jest uważne sprawdzanie relacji pomiędzy zmiennymi. W naszym przypadku `distinct` zawsze zawiera liczbę wartości `true` w tablicy `isCollected[]`.

Bez użycia tablic możemy nie być w stanie zasymulować procesu zbierania kuponów dla dużych wartości n; przy użyciu tablic jest to proste do wykonania. W tej książce przedstawimy jeszcze wiele przykładów takich procesów.

r	isCollected[]						distinct	count
	0	1	2	3	4	5		
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

Rysunek 1.28. Tabela śladu dla typowego wykonania java CouponCollector 6

Sito Eratostenesa

Liczby pierwsze pełnią ważną rolę w matematyce i programowaniu, w tym w kryptografii. **Liczba pierwsza** jest liczbą całkowitą większą od 1, której jedynymi dodatnimi dzielnikami jest 1 i ona sama. Funkcja zliczająca liczby pierwsze $\pi(n)$ zwraca liczbę liczb pierwszych mniejszych lub równych n. Przykładowo $\pi(25) = 0$, ponieważ pierwsze 9 liczb pierwszych to 2, 3, 5, 7, 11, 13, 17, 19 oraz 23. Funkcja ta pełni ważną rolę w teorii liczb.

Jednym z podejść do zliczania liczb pierwszych jest użycie takich programów jak `Factors` (listing 1.3.9). Możemy go zmodyfikować, aby ustawiał wartość logiczną na `true`, jeżeli liczba jest pierwsza, lub na `false`, jeżeli nie jest (zamiast wyświetlania dzielników), a następnie umieścić ten kod w pętli, która zwiększa licznik dla każdej liczby pierwszej. Podejście to jest efektywne dla małych `n`, ale staje się zbyt powolne, gdy `n` wzrasta.

Program `PrimeSieve` (listing 1.4.3) odczytuje z wiersza poleceń wartość całkowitą `n` i wyznacza liczbę liczb pierwszych za pomocą techniki nazywanej **sitem Eratostenesa**. Program wykorzystuje tablicę wartości `boolean isPrime[]` do zaznaczania liczb pierwszych. Naszym celem jest ustawienie w `isPrime[i]` wartości `true`, jeżeli `i` jest liczbą pierwszą, a w przeciwnym razie wartości `false`. Sito działa w następujący sposób: początkowo wszystkim elementom tablicy przypisujemy wartość `true`, co stanowi informację, że nie znaleziono dzielników danej liczby. Następnie powtarzamy następujące kroki do momentu, gdy `i <= n/i`:

- znajdujemy następną najmniejszą liczbę całkowitą `i`, dla której nie znaleziono podzielników,
- pozostawiamy `isPrime[i]` na `true`, ponieważ `i` nie ma mniejszych podzielników,
- ustawiamy na `false` w elementach `isPrime[]` dla wszystkich wielokrotności `i`.

Gdy zakończy się zagnieżdżona pętla `for`, `isPrime[i]` ma wartość `true` wyłącznie wtedy, gdy `i` jest liczbą pierwszą. Za pomocą jeszcze jednego przebiegu przez tabelę możemy zliczyć liczby pierwsze mniejsze lub równe `n`.

Kup ksi k Pole ksi k

Listing 1.4.3. Sito Eratostenesa

```
public class PrimeSieve
{
   public static void main(String[] args)
   { // Wyświetla liczbę liczb pierwszych <= n.
      int n = Integer.parseInt(args[0]);
      boolean[] isPrime = new boolean[n + 1];
      for (int i = 2; i <= n; i++)
         isPrime[i] = true;
      for (int i = 2; i <= n / i; i++)
      {
         if (isPrime[i])
         { // Zaznaczenie wielokrotności i jako niepierwsze.
            for (int j = i; j <= n / i; j++)
               isPrime[i * j] = false;
         }
      }
      // Zliczanie liczb pierwszych.
      int primes = 0;
      for (int i = 2; i <= n; i++)
         if (isPrime[i]) primes++;
      System.out.println(primes);
   }
}
```

`n`	argument
`isPrime[i]`	czy i jest pierwsza?
`primes`	licznik liczb pierwszych

Program ten odczytuje argument `n` *z wiersza poleceń i zlicza liczbę liczb pierwszych mniejszych lub równych* `n`. *W tym celu wykorzystuje tablicę wartości* `boolean isPrime[]`, *w której element o indeksie* `i` *ma wartość* `true`, *jeżeli* `i` *jest liczbą pierwszą, i* `false` *w przeciwnym przypadku. Na razie ustawiamy wszystkie wartości tablicy na* `true`, *co wskazuje, że początkowo nie stwierdziliśmy, że którakolwiek z liczb nie jest pierwsza. Następnie ustawiamy wartość* `false` *we wszystkich indeksach, o których wiemy, że nie są pierwsze (ponieważ są wielokrotnościami znanych liczb pierwszych). Jeżeli* `a[i]` *nadal ma wartość* `true` *po ustawieniu wszystkich wielokrotności mniejszych liczb na* `false`, *to wiemy, że* `i` *jest liczbą pierwszą. Warunkiem zakończenia w drugiej pętli* `for` *jest* `i <= n/i` *zamiast zwykłego* `i <= n`, *ponieważ żadna liczba nieposiadająca dzielników mniejszych niż* `n/i` *nie posiada dzielników większych niż* `n/i`, *więc nie trzeba ich szukać. To usprawnienie pozwala na uruchamianie programu dla większych wartości* `n`.

```
% java PrimeSieve 25
9
% java PrimeSieve 100
25
% java PrimeSieve 1000000000
50847534
```

Tak jak zwykle, w łatwy sposób możemy dodać do kodu instrukcje wyświetlające tabelę śladu. Przy programie, takim jak `PrimeSieve`, trzeba zachować ostrożność — zawiera on zagnieżdżone instrukcje `for-if-for`, więc musisz zwracać uwagę na nawiasy klamrowe, aby umieścić kod we właściwym miejscu. Zwróć uwagę, że zatrzymujemy się, gdy `i > n/i`, podobnie jak w `Factors`.

i	isPrime[]																							
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
3	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
5	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F
	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

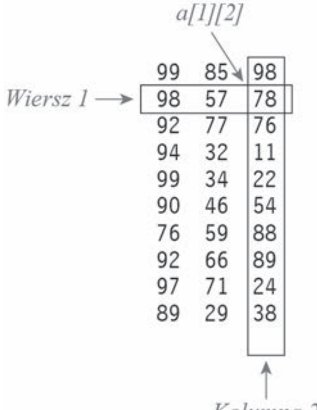
Rysunek 1.29. Tabela śladu dla java PrimeSieve 25

Przy użyciu `PrimeSieve` można obliczyć $\pi(n)$ dla dużych n, ograniczonych przede wszystkim przez maksymalną długość tablicy dozwolonej przez Javę. Jest to kolejny przykład kompromisu czasu i miejsca. Programy, takie jak `PrimeSieve`, pełnią ważną rolę jako pomoc dla matematyków przy opracowywaniu teorii liczb, która ma wiele ważnych zastosowań.

Tablice dwuwymiarowe

Wygodnym sposobem na zapisywanie informacji jest użycie tabeli liczb zorganizowanych w postaci prostokąta i odwoływanie się do **wierszy** i **kolumn** w tabeli. Przykładowo nauczyciel może utworzyć tabelę, w której wiersze odpowiadają studentom, a kolumny — wynikom egzaminów, naukowiec może zbudować tabelę z danymi eksperymentów, w której wiersze odpowiadają eksperymentom, a kolumny — różnym wynikom. Programista może przygotować rysunek do wyświetlenia przez utworzenie tabeli pikseli przyjmujących różne wartości skali szarości lub koloru.

Abstrakcją matematyczną takiej tabeli jest **macierz**, a konstrukcją w języku Java — **tablica dwuwymiarowa**. Prawdopodobnie napotkałeś już mnóstwo zastosowań macierzy i tablic dwuwymiarowych zarówno w nauce, inżynierii, jak i technikach obliczeniowych, a w książce tej przedstawimy jeszcze wiele takich przykładów. Podobnie jak w przypadku wektorów i tablic jednowymiarowych, wiele z najważniejszych zastosowań wymaga przetwarzania dużej ilości danych, więc odłożymy przedstawianie takich zastosowań do momentu omówienia mechanizmów wejścia i wyjścia w podrozdziale 1.5.



Rysunek 1.30. Anatomia tablicy dwuwymiarowej

Rozszerzenie tablicy Java o obsługę dwóch wymiarów jest bardzo proste. Aby odwołać się do elementu w wierszu `i` oraz kolumnie `j` w tablicy dwuwymiarowej `a[][]`, wykorzystujemy notację `a[i][j]`; do deklaracji tablicy dwuwymiarowej dodajemy jeszcze jedną parę nawiasów kwadratowych. W celu utworzenia tablicy określamy liczbę wierszy oraz kolumn po nazwie typu (obie wartości w nawiasach kwadratowych) w następujący sposób:

```
double[][] a = new double[m][n];
```

Tablicę taką nazywamy tablicą m na n. Zgodnie z konwencją, pierwszym wymiarem jest liczba wierszy, a drugim — liczba kolumn. Podobnie jak w tablicach jednowymiarowych, Java inicjalizuje wszystkie elementy wartością 0, a tablice logiczne — wartością false.

Domyślna inicjalizacja tablicy

Domyślna inicjalizacja tablicy dwuwymiarowej jest przydatna, ponieważ pozwala uniknąć pisania większej ilości kodu niż w przypadku tablic jednowymiarowych. Poniższy kod jest odpowiednikiem jednowierszowego idiomu tworzenia i inicjalizacji, którego użyliśmy powyżej:

```
double[][] a;
a = new double[m][n];
for (int i = 0; i < m; i++)
{  // Inicjalizacja i-tego wiersza.
   for (int j = 0; j < n; j++)
      a[i][j] = 0.0;
}
```

Kod ten jest nadmiarowy przy inicjalizacji elementów tablicy dwuwymiarowej zerami, ale takie zagnieżdżone pętle for są niezbędne do zainicjalizowania tablicy innymi wartościami. Jak widać, kod jest analogiczny do używanego w celu odwołania się do wszystkich elementów tablicy lub ich modyfikacji.

Wyświetlanie

Zagnieżdżone pętle for są wykorzystywane w wielu operacjach przetwarzania tablic dwuwymiarowych. Aby np. wyświetlić tablicę m na n w postaci tabelarycznej, możemy użyć następującego kodu:

```
for (int i = 0; i < m; i++)
{  // Wyświetlenie i-tego wiersza.
   for (int j = 0; j < n; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

Jeżeli chcesz, możesz dodać do kodu wyświetlanie indeksów wierszy i kolumn (ćwiczenie 1.4.6). Programiści Java zwykle formatują wyświetlane tablice dwuwymiarowe w taki sposób, że indeksy wierszy biegną od góry do dołu, licząc od 0, a indeksy kolumn biegną od lewej do prawej, licząc od 0.

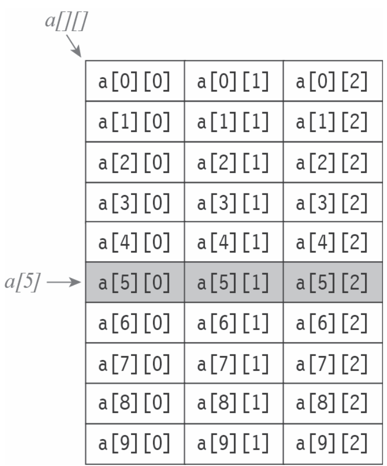
Reprezentacja w pamięci

Java reprezentuje tablice dwuwymiarowe jako tablice tablic. Oznacza to, że tablica dwuwymiarowa z m wierszami i n kolumnami jest w rzeczywistości tablicą o długości m, w której każdy element jest tablicą jednowymiarową o długości n. W przypadku dwuwymiarowej tablicy a[][] możesz użyć kodu a[i] do odwołania się do wiersza i (który jest tablicą jednowymiarową), ale nie ma analogicznego sposobu na odwołanie się do kolumny j.

Kup ksi k Pole ksi k

Ustawianie wartości w czasie kompilacji

Metoda inicjalizacji wartości tablicy w czasie kompilacji wynika bezpośrednio z jej reprezentacji w języku Java. Tablica dwuwymiarowa jest tablicą wierszy, więc każdy wiersz jest inicjalizowany jako tablica jednowymiarowa. Aby zainicjalizować tablicę dwuwymiarową, umieszczamy w nawiasach klamrowych listę składników inicjalizujących wiersze, oddzielając je od siebie przecinkami. Każdy składnik tej listy jest listą elementów wierszy oddzielonych od siebie przecinkami.



Rysunek 1.31. Tablica 10 na 3

```
double[][] a =
{
   { 99.0, 85.0, 98.0, 0.0 },
   { 98.0, 57.0, 79.0, 0.0 },
   { 92.0, 77.0, 74.0, 0.0 },
   { 94.0, 62.0, 81.0, 0.0 },
   { 99.0, 94.0, 92.0, 0.0 },
   { 80.0, 76.5, 67.0, 0.0 },
   { 76.0, 58.5, 90.5, 0.0 },
   { 92.0, 66.0, 91.0, 0.0 },
   { 97.0, 70.5, 66.5, 0.0 },
   { 89.0, 89.5, 81.0, 0.0 },
   {  0.0,  0.0,  0.0, 0.0 }
};
```

Rysunek 1.32. Inicjalizacja tablicy double 11 na 4 w czasie kompilacji

Arkusze kalkulacyjne

Jednym z najbardziej znanych zastosowań tablic są arkusze kalkulacyjne pozwalające na tworzenie tablic liczb. Przykładowo nauczyciel mający `m` studentów i `n` wyników testów dla każdego studenta może utworzyć tablicę $(m+1)$ na $(n+1)$, rezerwując ostatnią kolumnę na średnią każdego ze studentów i ostatni wiersz na średnią wyników testu. Mimo że tego typu obliczenia zwykle realizujemy w specjalizowanych aplikacjach, warto przeanalizować kod arkusza jako wprowadzenie do przetwarzania tablic. Aby obliczyć średnią ocenę dla każdego studenta (wartość średnią w wierszu), sumujemy elementy każdego wiersza i dzielimy je przez n. Kod przetwarza elementy macierzy wiersz po wierszu. Podobnie, aby obliczyć średnią ocenę z testu (wartość średnią kolumny), sumujemy elementy każdej kolumny i dzielimy wynik przez m. W tym przypadku kod przetwarza elementy macierzy kolumna po kolumnie.

Średnia wiersza w kolumnie n

n = 3

m = 10

99.0	85.0	98.0	94.0
98.0	57.0	79.0	78.0
92.0	77.0	74.0	81.0
94.0	62.0	81.0	79.0
99.0	94.0	92.0	95.0
80.0	76.5	67.0	74.5
76.0	58.5	90.5	75.0
92.0	66.0	91.0	83.0
97.0	70.5	66.5	78.0
89.0	89.5	81.0	86.5
91.6	73.6	82.0	

$\frac{92 + 77 + 74}{3}$

Średnia kolumny w wierszu m

$\frac{85 + 57 + \ldots + 89.5}{10}$

Obliczenie średnich w wierszach

```
for (int i = 0; i < m; i++)
{  // Obliczenie średniej dla wiersza i
   double sum = 0.0;
   for (int j = 0; j < n; j++)
      sum += a[i][j];
   a[i][n] = sum / n;
}
```

Obliczenie średnich w kolumnach

```
for (int j = 0; j < n; j++)
{  // Obliczenie średniej dla kolumny j
   double sum = 0.0;
   for (int i = 0; i < m; i++)
      sum += a[i][j];
   a[m][j] = sum / m;
}
```

Rysunek 1.33. Typowe obliczenia w arkuszu kalkulacyjnym

Kup ksi k Pole ksi k

```
a[][]                a[1][2]
.70 .20 .10
.30 .60 .10
.50 .10 .40

b[][]                b[1][2]
.20 .30 .50
.10 .20 .10
.10 .30 .40

c[][]                c[1][2]
.90 .50 .60
.40 .80 .20
.60 .40 .80
```

Rysunek 1.34. Dodawanie macierzy

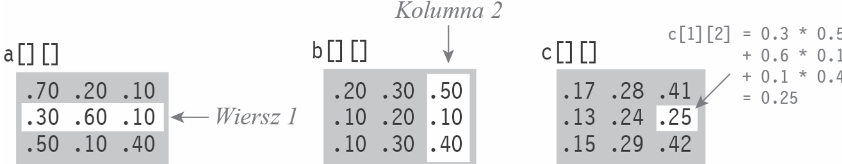
Operacje na macierzach

Typowym zastosowaniem tablic dwuwymiarowych w nauce i inżynierii jest reprezentacja za ich pomocą macierzy, a następnie wykonywanie różnych operacji matematycznych na operandach macierzowych. Również w tym przypadku operacje takie są najczęściej wykonywane w specjalizowanych aplikacjach, jednak warto rozumieć wykonywane przez nie obliczenia. Przykładowo w pokazany poniżej sposób można **dodać** do siebie dwie macierze n na n:

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      c[i][j] = a[i][j] + b[i][j];
```

W podobny sposób można **pomnożyć** dwie macierze. Być może uczyłeś się o mnożeniu macierzy, ale jeżeli nie przypominasz sobie metody, poniższy kod Java mnoży dwie macierze kwadratowe zgodnie z matematyczną definicją tej operacji. Każdy element `c[i][j]` w iloczynie `a[][]` i `b[][]` jest wyliczany przez obliczenie iloczynu skalarnego wiersza `i` z `a[][]` oraz kolumny `j` z `b[][]`.

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++)
{
   for (int j = 0; j < n; j++)
   {
      // Iloczyn skalarny wiersza i oraz kolumny j.
      for (int k = 0; k < n; k++)
         c[i][j] += a[i][k]*b[k][j];
   }
}
```



Rysunek 1.35. Mnożenie macierzy

Specjalne przypadki mnożenia macierzy

Szczególnie interesujące są dwa przypadki mnożenia macierzy. W tych specjalnych przypadkach jeden z wymiarów jednej macierzy wynosi 1, więc może być ona uważana za wektor. Mamy więc **mnożenie macierzy i wektora**, w którym mnożymy macierz m na n z wektorem (macierzą n na 1), otrzymując w wyniku wektor `m` na 1 (każdy element wyniku jest iloczynem skalarnym odpowiedniego wiersza w tablicy z wektorem). Drugi przypadek to **mnożenie wektora i macierzy**, w którym mnożymy wektor

Kup ksi k Pole ksi k

wiersza (macierz 1 na *m*) przez macierz *m* na *n*, uzyskując wektor wyniku 1 na *n* (każdy element wyniku jest iloczynem skalarnym operandu wektora i odpowiedniej kolumny w macierzy).

Operacje te zapewniają zwięzły sposób na wyrażanie wielu obliczeń na macierzach. Przykładowo obliczenie średniej wierszy dla arkusza z *m* wierszami i *n* kolumnami jest odpowiednikiem mnożenia macierzy i wektora, w którym wektor kolumny zawiera *n* elementów równych 1/*n*. Podobnie obliczenie średniej wartości w kolumnach jest odpowiednikiem mnożenia wektora i macierzy, w którym wektor kolumny zawiera *m* elementów równych 1/*m*. Do mnożenia wektorów i macierzy wrócimy w kontekście ważnej aplikacji przedstawianej na końcu tego rozdziału.

Mnożenie macierzy i wektora a[][]*x[] = b[]

```
for (int i = 0; i < m; i++)
{  // Iloczyn skalarny wiersza i oraz x[].
   for (int j = 0; j < n; j++)
      b[i] += a[i][j]*x[j];
}
```

```
     a[][]                      b[]
  [99  85  98]                 [94]
  [98  57  78]                 [77]
  [92  77  76]        x[]      [81]
  [94  32  11]                 [45]
  [99  34  22]       [.33]     [51]  <- Średnie
  [90  46  54]       [.33]     [63]     wierszy
  [76  59  88]       [.33]     [74]
  [92  66  89]                 [82]
  [97  71  24]                 [64]
  [89  29  38]                 [52]
```

Mnożenie wektora i macierzy y[]*a[][] = c[]

```
for (int j = 0; j < n; j++)
{  // Iloczyn skalarny y[] oraz kolumny j.
   for (int i = 0; i < m; i++)
      c[j] += y[i]*a[i][j];
}
```

```
y[] [ .1 .1 .1 .1 .1 .1 .1 .1 .1 .1 ]

a[][] [99  85  98]
      [98  57  78]
      [92  77  76]
      [94  32  11]
      [99  34  22]
      [90  46  54]
      [76  59  88]
      [92  66  89]
      [97  71  24]
      [89  29  38]

c[]   [92  55  57]  <- Średnie
                       kolumn
```

Rysunek 1.36. Mnożenie macierzy wektora oraz wektora i macierzy

Tablice postrzępione

Nie istnieje wymaganie, aby wszystkie wiersze w tablicy dwuwymiarowej miały taką samą długość — tablica z wierszami o różnej długości jest nazywana **tablicą postrzępioną** (przykładowe zastosowanie znajdziesz w ćwiczeniu 1.4.34). Możliwość wykorzystania tablic poszarpanych wymaga większej uwagi przy tworzeniu kodu przetwarzającego tablicę. Przykładowo poniższy kod wyświetla zawartość tablicy postrzępionej:

```
for (int i = 0; i < a.length; i++)
{
   for (int j = 0; j < a[i].length; j++)
      System.out.print(a[i][j] + " ");
    System.out.println();
}
```

Kod ten sprawdza Twoją wiedzę na temat tablic w języku Java, więc powinieneś poświęcić nieco czasu na jego przeanalizowanie. W książce tej zwykle używamy tablic kwadratowych lub prostokątnych, których wymiary są określane za pomocą zmiennych `m` lub `n`. Kod wykorzystujący `a[i].length` to jasny sygnał, że tablica jest postrzępiona.

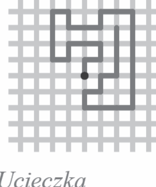
Tablice wielowymiarowe

Stosowaną wcześniej notację można rozszerzyć, aby napisać kod wykorzystujący dowolną liczbę wymiarów. Możemy np. zadeklarować i zainicjalizować tablicę trójwymiarową:

```
double[][][] a = new double[n][n][n];
```

a następnie odwoływać się do jej elementów w kodzie za pomocą `a[i][j][k]`.

TABLICE DWUWYMIAROWE SĄ NATURALNĄ REPREZENTACJĄ DLA MACIERZY POWSZECHNIE stosowanych w nauce, matematyce i inżynierii. Oferują one również naturalny sposób organizacji dużych ilości danych — są one kluczowym elementem w arkuszu kalkulacyjnym i innych aplikacjach obliczeniowych. Poprzez użycie współrzędnych kartezjańskich tablice dwu- i trójwymiarowe są również podstawą modelowania świata. W książce przedstawimy je we wszystkich trzech obszarach zastosowań.



Ucieczka

Rysunek 1.37. Spacery z samounikaniem

Przykład: losowy spacer z samounikaniem

Załóżmy, że pozostawimy psa w środku dużego miasta, którego ulice tworzą znany wzór siatki. Załóżmy też, że mamy `n` ulic północ-południe i `n` ulic wschód-zachód, wszystkie są od siebie jednakowo oddalone i przecinając się, tworzą wzór nazywany **kratą**. Pies, kiedy próbuje uciec z miasta, dokonuje losowych wyborów na każdym skrzyżowaniu, ale korzystając z węchu, wie że musi unikać wcześniej odwiedzonych miejsc. Możliwe jest, że pies utknie i nie będzie miał innej możliwości niż użycie odwiedzonego skrzyżowania. Jaka jest szansa, że zdarzy się taka sytuacja? Ten problem jest prostym przykładem znanego modelu nazywanego **losowym spacerem z samounikaniem**, który ma ważne zastosowania naukowe m.in. przy badaniu polimerów oraz w mechanice statystycznej. Możesz np. zauważyć, że proces ten modeluje stopniowo rosnący łańcuch materiału, aż do osiągnięcia kresu możliwości wzrostu. Aby lepiej zrozumieć te procesy, naukowcy starają się zrozumieć właściwości spacerów z samounikaniem.

Prawdopodobieństwo ucieczki psa jest zależne od wielkości miasta. Przy niewielkim mieście 5 na 5 łatwo się przekonać, że pies ucieknie. A jakie jest prawdopodobieństwo ucieczki, gdy miasto jest duże? Występują tu również inne ciekawe parametry. Jaka jest np. średnia długość trasy psa? Jak często pies jest w odległości jednego skrzyżowania od ucieczki? Tego typu właściwości są ważne we wspomnianych zastosowaniach.

Program `SelfAvoidingWalk` (listing 1.4.4) jest symulacją wykorzystującą dwuwymiarową tablicę typu `boolean`, której każdy element reprezentuje skrzyżowanie. Wartość `true` wskazuje, że pies odwiedził skrzyżowanie; `false`, że skrzyżowanie nie było odwiedzone. Trasa zaczyna się w środku i wykonywane są losowe kroki w nieodwiedzonych wcześniej kierunkach, do momentu gdy zajdziemy w ślepy zaułek lub uciekniemy, będąc na granicy. Dla uproszczenia kod jest napisany w taki sposób, że jeżeli losowy wybór kieruje nas w odwiedzonym kierunku, to nie jest podejmowana żadna akcja w nadziei, że któryś kolejny losowy wybór skieruje nas do nowego miejsca (co jest pewne, ponieważ kod sprawdza wystąpienie ślepego zaułka i w takim przypadku kończy pętlę).

Kup ksi k Pole ksi k

Listing 1.4.4. Losowy spacer z samounikaniem

```
public class SelfAvoidingWalk
{
   public static void main(String[] args)
   {  // Wykonanie trials losowych spacerów
      // z samounikanem w kracie n na n.
      int n = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);
      int deadEnds = 0;
      for (int t = 0; t < trials; t++)
      {
         boolean[][] a = new boolean[n][n];
         int x = n/2, y = n/2;
         while (x > 0 && x < n-1 && y > 0 && y < n-1)
         {  // Sprawdzenie ślepego zaułka i wykonanie losowego ruchu.
            a[x][y] = true;
            if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1])
            { deadEnds++; break; }
            double r = Math.random();
            if (r < 0.25) { if (!a[x+1][y]) x++; }
            else if (r < 0.50) { if (!a[x-1][y]) x--; }
            else if (r < 0.75) { if (!a[x][y+1]) y++; }
            else if (r < 1.00) { if (!a[x][y-1]) y--; }
         }
      }
      System.out.println(100*deadEnds/trials + "% ślepych zaułków");
   }
}
```

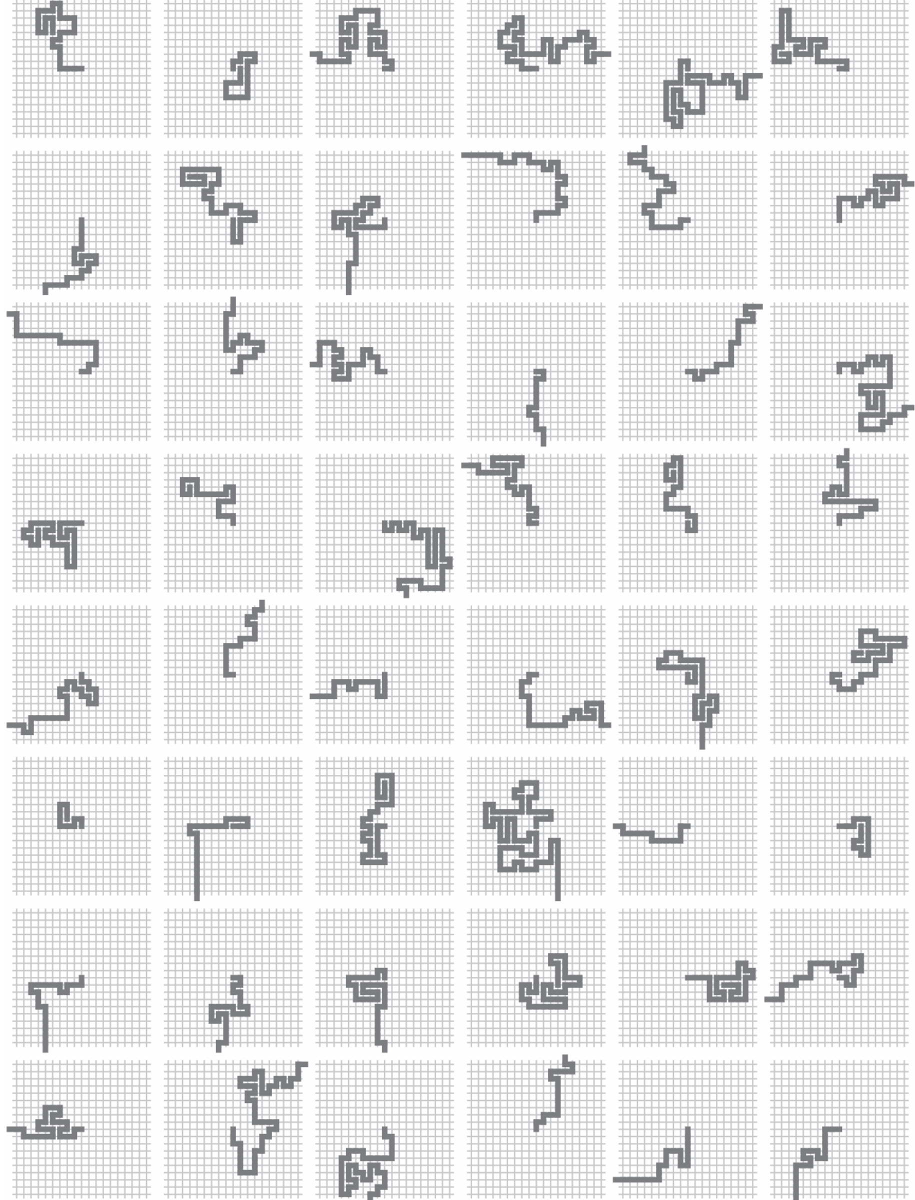
`n`	wielkość kraty
`trials`	liczba prób
`deadEnd`	liczba prób skutkujących ślepym zaułkiem
`a[][]`	odwiedzone skrzyżowanie
`x, y`	bieżąca pozycja
`r`	liczba losowa z zakresu (0, 1)

Program ten odczytuje z wiersza poleceń argumenty n oraz `trials` i wykonuje `trials` spacerów z samounikaniem na kracie n na n. Dla każdego spaceru tworzy tablicę typu `boolean`, zaczyna spacer od środka i kontynuuje do momentu osiągnięcia ślepego zaułka lub granicy. Wynikiem obliczenia jest procent ślepych zaułków. Zwiększenie liczby eksperymentów zwiększa dokładność.

```
% java SelfAvoidingWalk 5 100
0% ślepych zaułków
% java SelfAvoidingWalk 20 100
36% ślepych zaułków
% java SelfAvoidingWalk 40 100
80% ślepych zaułków
% java SelfAvoidingWalk 80 100
98% ślepych zaułków
```

```
% java SelfAvoidingWalk 5 1000
0% ślepych zaułków
% java SelfAvoidingWalk 20 1000
32% ślepych zaułków
% java SelfAvoidingWalk 40 1000
70% ślepych zaułków
% java SelfAvoidingWalk 80 1000
95% ślepych zaułków
```

Zwróć uwagę, że kod korzysta z inicjalizacji tablicy wartością `false` przez środowisko Java. Wykorzystuje on również ważną technikę programowania, w której wyrażenie kończące pętlę `while` jest **strażnikiem** nieprawidłowej instrukcji w ciele pętli. W tym przypadku warunek kontynuacji pętli `while` służy jako strażnik dostępu poza granicą tablicy w ciele pętli. Odpowiada to testowi, czy pies uciekł z miasta. Wewnątrz pętli spełniony test ślepego zaułka powoduje wykonanie instrukcji `break` i przerwania pętli.



Rysunek 1.38. Losowe spacery z samounikaniem na kracie 21 na 21

Jak można się przekonać na podstawie przykładowych uruchomień zilustrowanych na rysunku 1.38, niemal pewne jest, że w dużym mieście pies utknie w ślepym zaułku. Jeżeli chcesz dowiedzieć się więcej na temat spacerów z samounikaniem, możesz znaleźć kilka sugestii w ćwiczeniach. Przykładowo w wersji trójwymiarowej tego problemu niemal pewne jest, że pies ucieknie. Choć jest to intuicyjny wynik potwierdzony przez nasze eksperymenty, to opracowanie matematycznego modelu wyjaśniającego działanie

Kup ksi k Pole ksi k

spacerów z samounikaniem jest nadal otwartym problemem. Pomimo rozległych badań nikt nie zna zwięzłego matematycznego wyrażenia wyznaczającego prawdopodobieństwo ucieczki, średnią długość trasy lub jakiegokolwiek inny ważny parametr.

Podsumowanie

Tablice są czwartym podstawowym elementem (po przypisaniu, warunkach i pętlach) znajdującym się w niemal każdym języku programowania. Jak się można przekonać na podstawie przykładowych zaprezentowanych programów, możesz pisać programy korzystające z tych konstrukcji, które rozwiązują wszystkie rodzaje problemów.

Tablice są ważne dla wielu programów, a podstawowe operacje, jakie tu omawialiśmy, służą jako podstawa do rozwiązywania wielu zadań programistycznych. Gdy nie korzystamy jawnie z tablic (a często tak będzie), będziemy używać ich niejawnie, ponieważ wszystkie komputery mają pamięć, która jest koncepcyjnie odpowiednikiem tablicy.

Podstawową cechą, jaką tablice dodają do naszych programów, jest potencjalne ogromne zwiększenie rozmiaru **stanu** programu. Stan programu może być zdefiniowany jako dane potrzebne do zrozumienia działania programu. W programach bez tablic, jeżeli znasz wartości zmiennych i następną wykonywaną instrukcję, możesz określić, co program będzie robił dalej. Gdy śledzimy program, właśnie śledzimy jego stan. Gdy jednak program korzysta z tablic, to liczba wartości może być zbyt duża (a każda może być zmieniona w każdej instrukcji), aby można było je efektywnie śledzić. Różnica ta powoduje, że pisanie programów z tablicami jest trudniejsze niż bez nich.

Tablice reprezentują wektory i macierze, więc są one bezpośrednio używane w obliczeniach dotyczących wielu podstawowych problemów naukowych i inżynierskich. Tablice zapewniają spójną notację do manipulowania dużą ilością danych w jednorodny sposób, więc pełnią ważną rolę w aplikacjach wymagających przetwarzania takich ilości danych.

Kup ksi k Pole ksi k

Pytania i odpowiedzi

P. Niektórzy programiści korzystają z a[] zamiast int[] przy deklaracji tablic. Jaka to różnica?

O. W języku Java obie konstrukcje są dozwolone i wymienne. W pierwszy sposób deklaruje się tablice w języku C. Drugi jest preferowanym stylem w języku Java, ponieważ typ zmiennej int[] jednoznacznie wskazuje, że jest to **tablica liczb całkowitych**.

P. Dlaczego indeksy tablicy zaczynają się od 0, a nie od 1?

O. Konwencja ta pochodzi z programowania niskopoziomowego, gdzie adres elementu tablicy jest obliczany przez dodanie indeksu do adresu początku tablicy. Zaczynając od indeksu 1, będziemy albo marnować miejsce na początku tablicy, albo czas na wykonanie dodatkowego zmniejszenia adresu o 1.

P. Co się stanie, gdy użyję ujemnych wartości do indeksowania tablicy?

O. To samo, co przy użyciu zbyt dużego indeksu. Gdy program będzie próbował indeksować tablicę za pomocą wartości niemieszczącej się w przedziale od 0 do długości tablicy minus 1, Java wygeneruje błąd ArrayIndexOutOfBoundsException.

P. Czy pozycje przy inicjalizacji tablicy muszą być literałami?

O. Nie, pozycje inicjalizacji tablicy mogą być dowolnymi wyrażeniami (wymaganego typu), nawet jeżeli wartości te nie są znane w czasie kompilacji. Przykładowo poniższy kod inicjalizuje tablicę dwuwymiarową z użyciem argumentu wiersza poleceń theta:

```
double theta = Double.parseDouble(args[0]);
double[][] rotation =
{
   { Math.cos(theta), -Math.sin(theta) },
   { Math.sin(theta), Math.cos(theta) },
};
```

P. Czy jest różnica między tablicą znaków a wartością String?

O. Tak, np. możesz zmienić pojedyncze znaki w char[], a nie można tego zrobić w String. Typ String przedstawimy dokładnie w podrozdziale 3.1.

P. Co się stanie, gdy porównamy dwie tablice za pomocą (a == b)?

O. Wyrażenie to przyjmie wartość true wtedy i tylko wtedy, gdy a[] i b[] będą wskazywały na tę samą tablicę (adres w pamięci), a nie na taką samą sekwencję wartości. Niestety, najczęściej nie jest to oczekiwane działanie. Zamiast tego powinieneś użyć pętli do porównania odpowiadających sobie elementów.

P. Co się stanie, gdy użyję tablicy w instrukcji przypisania, takiej jak a = b?

O. Instrukcja przypisania spowoduje, że zmienna a będzie odwoływała się do tej samej tablicy, co b — nie spowoduje skopiowania wartości z tablicy b do a, jak można by tego oczekiwać. Dla przykładu przeanalizujmy następujący fragment kodu:

Kup ksi k Pole ksi k

```
int[] a = { 1, 2, 3, 4 };
int[] b = { 5, 6, 7, 8 };
a = b;
a[0] = 9;
```

Po instrukcji przypisania `a = b` mamy `a[0]` równe 5, `a[1]` równe 6 itd., jak się tego spodziewaliśmy. Tablice zawierają takie same sekwencje wartości. Jednak nie są to tablice **niezależne**. Przykładowo ostatnia instrukcja nie tylko zmieni `a[0]` na 9, ale również element `b[0]` będzie miał wartość 9. Jest to jedna z kluczowych różnic między typami prostymi (takimi jak `int` czy `double`) a typami złożonymi (takimi jak tablice). Do tej subtelnej (ale podstawowej) różnicy wrócimy przy okazji przekazywania tablic do funkcji w podrozdziale 2.1 oraz przy okazji typów referencyjnych w podrozdziale 3.1.

P. Jeżeli `a[]` jest tablicą, dlaczego `System.out.println(a)` wyświetla wartość, taką jak `@f62373`, zamiast sekwencji wartości z tablicy?

O. Dobre pytanie. Wyświetlany jest adres tablicy w pamięci (w postaci szesnastkowej), co niestety nie jest tym, czego najczęściej chcemy.

P. Na jakie jeszcze pułapki powinienem zwracać uwagę przy korzystaniu z tablic?

O. Trzeba pamiętać, że Java **automatycznie** inicjalizuje tablice przy ich tworzeniu, więc **tworzenie tablicy zajmuje czas proporcjonalny do jej długości**.

Kup ksi k Pole ksi k

Ćwiczenia

1.4.1. Napisz program, który deklaruje, tworzy i inicjalizuje tablicę `a[]` o długości `1000` i odwołaj się do `a[1000]`. Czy program się skompiluje? Co się stanie przy jego uruchomieniu?

1.4.2. Opisz i wyjaśnij, co się stanie, gdy spróbujesz skompilować program z poniższą instrukcją:

```
int n = 1000;
int[] a = new int [n*n*n*n];
```

1.4.3. Mając dwa wektory długości `n`, które są reprezentowane za pomocą tablic jednowymiarowych, napisz fragment kodu obliczający **odległość euklidesową** pomiędzy nimi (pierwiastek z sumy kwadratów różnic pomiędzy odpowiadającymi sobie elementami).

1.4.4. Napisz fragment kodu, który odwraca kolejność wartości w jednowymiarowej tablicy ciągów znaków. Nie twórz innej tablicy do przechowywania wyniku. **Wskazówka**: użyj kodu do wymiany wartości dwóch elementów.

1.4.5. Jaki błąd kryje się w poniższym fragmencie kodu?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

1.4.6. Napisz fragment kodu wyświetlający zawartość dwuwymiarowej tablicy `boolean`, używając `*` do reprezentowania wartości `true` i spacji do reprezentowania wartości `false`. Dodaj indeksy wierszy i kolumn.

1.4.7. Co wyświetli poniższy fragment kodu?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(a[i]);
```

1.4.8. Jakie wartości poniższy kod umieści w `a[]`?

```
int n = 10;
int[] a = new int[n];
a[0] = 1;
a[1] = 1;
for (int i = 2; i < n; i++)
   a[i] = a[i-1] + a[i-2];
```

1.4.9. Co wyświetli poniższy fragment kodu?

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a == b);
```

Kup ksi k Pole ksi k

1.4.10. Napisz program `Deal`, który odczyta z wiersza poleceń wartość całkowitą `n` i wyświetli `n` rozdanych rąk pokerowych (po pięć kart) z potasowanej talii, oddzielając je pustymi wierszami.

1.4.11. Napisz program `HowMany`, który odczytuje dowolną liczbę wartości z wiersza poleceń i wyświetla ich liczbę.

1.4.12. Napisz program `DiscreteDistribution`, który odczytuje z wiersza poleceń dowolną liczbę liczb całkowitych i wyświetla liczbę `i` z prawdopodobieństwem proporcjonalnym do wartości `i`-tego argumentu wiersza poleceń.

1.4.13. Napisz fragment kodu tworzący dwuwymiarową tablice `b[][]`, która jest kopią istniejącej tablicy dwuwymiarowej `a[][]`, przy następujących założeniach:

a. `a[][]` jest kwadratowa,
b. `a[][]` jest prostokątna,
c. `a[][]` może być postrzępiona.

Rozwiązanie dla punktu *b* powinno działać dla punktu *a*, a rozwiązanie dla *c* powinno działać zarówno dla *b*, jak i *a*, a kod powinien być stopniowo coraz bardziej złożony.

1.4.14. Napisz fragment kodu, który wyświetla *transpozycję* (zamienione wiersze z kolumnami) dwuwymiarowej tablicy prostokątnej. Dla przykładowej tablicy arkusza kalkulacyjnego użytej wcześniej w tekście kod ten powinien wyświetlić następujący wynik:

```
99 98 92 94 99 90 76 92 97 89
85 57 77 32 34 46 59 66 71 29
98 78 76 11 22 54 88 89 24 38
```

1.4.15. Napisz fragment kodu wykonujący transpozycję kwadratowej tablicy **w miejscu**, bez tworzenia drugiej tablicy.

1.4.16. Napisz program odczytujący liczbę całkowitą `n` z wiersza poleceń i tworzący tablicę `boolean a[][]` `n` na `n` taką, że `a[i][j]` ma wartość `true`, gdy `i` oraz `j` są względnie pierwsze (nie mają wspólnego podzielnika), a w przeciwnym razie ma wartość `false`. Użyj rozwiązania ćwiczenia 1.4.6 do wyświetlenia tablicy. **Wskazówka**: skorzystaj z odsiewania.

1.4.17. Zmień fragment kodu arkusza kalkulacyjnego, aby obliczał średnią **ważoną** dla wierszy, gdzie wagi każdego z egzaminów są zapisane w jednowymiarowej tablicy `weights[]`. Aby np. ostatni z trzech egzaminów miał podwójną wagę, należy użyć:

```
double[] weights = { 0.25, 0.25, 0.50 };
```

Zauważ, że suma wag powinna dawać 1.

Kup ksi k Pole ksi k

1.4.18. Napisz fragment kodu mnożący dwie macierze **prostokątne**, a nie kwadratowe. **Uwaga**: aby prawidłowo obliczyć iloczyn skalarny, liczba kolumn w pierwszej macierzy musi być równa liczbie wierszy w drugiej macierzy. Wyświetl komunikat błędu, jeżeli wymiary tablic nie spełniają tego warunku.

1.4.19. Napisz program mnożący dwie kwadratowe macierze `boolean`, używając operacji `or` zamiast `+` i `and` zamiast `*`.

1.4.20. Zmień program `SelfAvoidingWalk` (listing 1.4.4), aby oprócz prawdopodobieństwa ślepego zaułka obliczał i wyświetlał średnią długość tras. Rejestruj osobno średnie długości tras ucieczki i tras prowadzących do ślepego zaułka.

1.4.21. Zmień program `SelfAvoidingWalk`, aby obliczał i wyświetlał średni obszar najmniejszego prostokąta obejmującego trasę do ślepego zaułka.

Kup ksi k Pole ksi k

Ćwiczenia kreatywne

1.4.22. Symulacja kostki. Poniższy kod oblicza dokładne prawdopodobieństwo rozkładu sum dwóch kostek:

```
int[] frequencies = new int[13];
for (int i = 1; i <= 6; i++)
   for (int j = 1; j <= 6; j++)
      frequencies[i+j]++;
double[] probabilities = new double[13];
for (int k = 1; k <= 12; k++)
   probabilities[k] = frequencies[k] / 36.0;
```

Wartość `probabilities[k]` jest prawdopodobieństwem wystąpienia sumy kostek równej `k`. Uruchom eksperymenty kontrolujące poprawność tego obliczenia przez symulowanie `n` rzutów kostką i przechowanie częstotliwości wystąpienia każdej wartości sumy dwóch liczb losowych pomiędzy 1 a 6. Jaka musi być wartość `n`, aby wyniki empiryczne zgadzały się z wyznaczonymi z dokładnością do trzech liczb po przecinku.

1.4.23. Najdłuższy płaskowyż. Mając tablicę liczb całkowitych, wyznacz długość i położenie najdłuższej sekwencji identycznych liczb, dla których wartości elementów bezpośrednio przed nią i bezpośrednio po niej są mniejsze.

1.4.24. Empiryczna kontrola tasowania. Wykonaj eksperymenty obliczeniowe pozwalające na sprawdzenie, czy nasz kod tasujący działa w oczekiwany sposób. Napisz program `ShuffleTest`, który odczytuje z wiersza poleceń dwie wartości całkowite `m` oraz `n` i wykonuje `n` tasowań tablicy o długości `m`, która przed każdym tasowaniem była zainicjalizowana za pomocą wyrażenia `a[i] = i`, a następnie wyświetla tabelę `m` na `m` w taki sposób, że w wierszu `i` znajduje się liczba wystąpień `i` na pozycji `j`, dla wszystkich `j`. Wszystkie wartości w wynikowej tablicy powinny być zbliżone do `n/m`.

1.4.25. Złe tasowanie. Załóżmy, że w kodzie tasującym losowałeś liczbę z zakresu od `0` do `n-1` zamiast między `i` a `n-1`. Pokaż, że wynikowy porządek **nie jest równie prawdopodobny** dla `n`! możliwości. Uruchom kod z powyższego ćwiczenia w tej wersji.

1.4.26. Miksowanie muzyki. Ustaw swój odtwarzacz muzyki w tryb miksowania. Odtwarza on każdy z `n` utworów, zanim zacznie od nowa. Napisz program szacujący prawdopodobieństwo, że nie usłyszysz żadnego z dwóch kolejnych utworów (czyli utwór 3. nie będzie odtworzony po 2., utwór 10. nie będzie odtworzony po 9. itd.).

1.4.27. Minima w permutacjach. Napisz program, który odczytuje z wiersza poleceń liczbę `n`, losuje losową permutację długości `n`, wyświetla permutację, a następnie wyświetla liczbę minimów od lewej do prawej (liczba razy, gdy element jest do tej pory najmniejszy). Następnie napisz program, który odczytuje z wiersza poleceń dwie liczby `m` i `n`, losuje `m` losowych permutacji długości `n`, a następnie wyświetla średnią liczbę minimów od lewej do prawej w wygenerowanych permutacjach. **Dodatkowa premia**: określ hipotezę na temat liczby minimów od lewej do prawej w permutacji długości `n` jako funkcję `n`.

Kup książkę Poleć książkę

1.4.28. Odwracanie permutacji. Napisz program, który odczytuje permutację liczb losowych od `0` do `n–1` z `n` argumentów wiersza poleceń, i wyświetl permutację odwrotną (jeżeli permutacja jest tablicą `a[]`, to jej odwrotnością jest tablica `b[]` taka, że `a[b[i]] = b[a[i]] = i`). Upewnij się, że na wejściu jest podana prawidłowa permutacja.

1.4.29. Macierz Hadamarda. Macierz Hadamarda $H(n)$ wielkości n na n jest macierzą logiczną o takiej właściwości, że każde dwa wiersze różnią się o dokładnie $n/2$ wartości (ta właściwość jest przydatna przy projektowaniu kodów korekcji błędów). $H(1)$ jest macierzą 1 na 1 z jednym elementem wartości `true`, a dla $n > 1$ $H(2n)$ uzyskuje się przez złączenie czterech kopii $H(n)$ w większy kwadrat, a następnie odwrócenie wszystkich wartości w prawej dolnej kopii n na n, jak jest to pokazane na poniższym przykładzie (gdzie jak zwykle `T` reprezentuje `true`, a `F` reprezentuje `false`:

$H(1)$	$H(2)$	$H(4)$
T	T T	T T T T
	T F	T F T F
		T T F F
		T F F T

Napisz program, który odczytuje z wiersza poleceń argument `n` i wyświetla $H(n)$. Załóż, że `n` jest potęgą 2.

1.4.30. Plotki. Alicja organizuje przyjęcie dla n gości, w tym Bogdana. Bogdan rozgłasza plotkę na temat Alicji, przekazując ją jednemu z gości. Osoba słysząca plotkę po raz pierwszy natychmiast przekazuje ją innemu gościowi, wybranego w sposób jednorodny z pozostałych osób na przyjęciu, poza Alicją i osobą, od której usłyszała plotkę. Jeżeli osoba (w tym Bogdan) usłyszy plotkę po raz kolejny, nie będzie jej dalej przekazywała. Napisz program szacujący prawdopodobieństwo, że wszyscy na przyjęciu (poza Alicją) usłyszą plotkę, zanim przestanie być przekazywana. Oblicz również oczekiwaną liczbę osób, które usłyszały plotkę.

1.4.31. Wyznaczanie liczb pierwszych. Porównaj program `PrimeSieve` z programem, który wykorzystywaliśmy do demonstracji instrukcji `break` na końcu podrozdziału 1.3. Jest to klasyczny przykład kompromisu przestrzeni i czasu: program `PrimeSieve` jest szybki, ale wymaga tablicy `boolean` długości n; drugie z podejść wymaga tylko dwóch liczb całkowitych, ale jest wyraźnie wolniejsze. Oszacuj rząd tej różnicy przez znalezienie wartości n, dla której drugie podejście zakończy obliczenia w tym samym czasie, co `java PrimeSieve 1000000`.

1.4.32. Saper. Napisz program, który odczytuje trzy argumenty wiersza poleceń, `m`, `n` i `p`, a następnie generuje tablicę `boolean` `m` na `n`, w której każdy element jest zajęty z prawdopodobieństwem `p`. W grze „Saper” zajęte komórki reprezentują minę, a puste —

Kup ksi k Pole ksi k

bezpieczne miejsce. Wyświetl tabelę, używając gwiazdki dla min oraz kropki dla bezpiecznych miejsc. Następnie utwórz dwuwymiarową tablicę liczb całkowitych z liczbą sąsiednich bomb (powyżej, poniżej, po lewej, po prawej i po przekątnych).

```
* * . . .        * * 1 0 0
. . . . .        3 3 2 0 0
. * . . .        1 * 1 0 0
```

Napisz program tak, aby miał możliwie mało przypadków specjalnych, korzystając z tablicy `boolean` wielkości (m+2) na (n+2).

1.4.33. Wyszukiwanie powtórzeń. Mając tablice długości `n`, o wartościach pomiędzy 1 a `n`, napisz fragment kodu określający, czy w tablicy znajdują się powtórzone wartości. Możesz użyć dodatkowej tablicy (ale nie musisz zachowywać zawartości danej tablicy).

1.4.34. Długość spaceru z samounikaniem. Załóżmy, że nie istnieje ograniczenie wielkości siatki. Wykonaj eksperymenty do określenia średniej długości trasy.

1.4.35. Trójwymiarowy spacer z samounikaniem. Wykonaj eksperymenty weryfikujące twierdzenie, że prawdopodobieństwo napotkania ślepego zaułka jest równe 0 dla trójwymiarowej wersji spaceru z samounikaniem i oblicz średnią długość ścieżki dla różnych wartości n.

1.4.36. Losowi spacerowicze. Załóżmy, że n losowych spacerowiczów zaczyna spacer od środka siatki n na n, przesuwając się o jeden krok w lewo, w prawo, w górę lub w dół, z jednakowym prawdopodobieństwem wybrania każdego kierunku. Napisz program, który pomoże sformułować i przetestować hipotezę na temat liczby kroków potrzebnych do odwiedzenia wszystkich komórek.

1.4.37. Brydż. W brydżu czterem osobom rozdaje się po 13 kart. Ważną statystyką jest rozkład liczby kart w każdym z kolorów. Co jest bardziej prawdopodobne, 5–3–3–2, 4–4–3–2 czy 4–3–3–3?

1.4.38. Problem urodzin. Załóżmy, że do pokoju wchodzą kolejno osoby aż do momentu, gdy dwie z nich będą miały urodziny w tym samym dniu. Ile średnio osób będzie musiało wejść do pokoju, zanim znajdziemy dopasowanie? Wykonaj eksperymenty w celu oszacowania tych wartości. Zakładamy, że urodziny są jednorodnie rozłożone i reprezentowane przez liczby od 0 do 364.

1.4.39. Kolekcjoner kuponów. Wykonaj eksperymenty sprawdzające poprawność matematycznie wyznaczonej, oczekiwanej liczby kuponów potrzebnych do zebrania n wartości, wynoszącej $n\ H_n$, gdzie H_n jest n-tą liczbą harmoniczną. Jeżeli np. uważnie obserwujesz karty przy stole do gry w oko (i krupier ma wystarczającą liczbę posortowanych talii), to będziesz musiał poczekać 235 rozdań, aby zobaczyć każdą z kart.

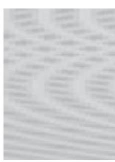
Kup książkę Poleć książkę

1.4.40. Tasowanie z przerzucaniem. Napisz program do tasowania talii *n* kart za pomocą modelu Gilberta–Shannona–Reedsa dla tasowania z przerzucaniem. Na początek wygeneruj liczbę losową *r* zgodnie z rozkładem dwumianowym: rzuć monetą *n* razy i niech *r* będzie liczbą orłów. Teraz podziel talię na dwa stosy: pierwszy zawiera pierwsze *r* kart, a drugi pozostałe $n - r$ kart. Aby dokończyć tasowanie, bierz kolejno kartę z góry jednego ze stosów kart i umieszczaj ją na dole nowego. Jeżeli w pierwszym stosie pozostało n_1 kart, a w drugim n_2, wybierz następną kartę z pierwszego stosu z prawdopodobieństwem $n_1 / (n_1 + n_2)$, a z drugiego z prawdopodobieństwem $n_2 / (n_1 + n_2)$. Sprawdź, ile razy należy wykonać takie tasowanie talii 52 kart, aby uzyskać (niemal) jednorodnie potasowaną talię.

1.4.41. Rozkład dwumianowy. Napisz program, który odczyta z wiersza poleceń argument `n` i utworzy dwuwymiarową poszarpaną tablicę `a[][]` taką, że `a[n][k]` zawiera prawdopodobieństwo uzyskania dokładnie `k` orłów przy `n`-krotnym rzucie monetą. Liczby te są nazywane **rozkładem dwumianowym**: jeżeli pomnożysz każdy element wiersza *i* przez 2^n, uzyskasz **współczynniki dwumianowe** — współczynniki x^k w $(x+1)^n$ — ustawione w **trójkąt Pascala**. Aby je obliczyć, rozpocznij od `a[n][0] = 0.0` dla wszystkich `n` oraz `a[1][1] = 1.0`, a następnie obliczaj wartości w kolejnych wierszach od lewej do prawej, za pomocą wyrażenia `a[n][k] = (a[n-1][k] + a[n-1][k-1]) / 2.0`.

trójkąt Pascala					*rozkład dwumianowy*				
1	1				1				
1	1				1/2	1/2			
1	2	1			1/4	1/2	1/4		
1	3	3	1		1/8	3/8	3/8	1/8	
1	4	6	4	1	1/16	1/4	3/8	1/4	1/16

Kup ksi k Pole ksi k



PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

JAVA — NAJLEPSZE NARZĘDZIE DLA INŻYNIERÓW, INFORMATYKÓW I NAUKOWCÓW!

Programowania uczą się już najmłodsi w szkole podstawowej. Umiejętność kodowania będzie wkrótce jednym z wyznaczników dobrego wykształcenia. Rozwiązywanie złożonych problemów za pomocą specjalnie napisanego kodu jest dogodnym sposobem pracy nie tylko inżyniera czy informatyka, ale również biologa, fizyka, a nawet socjologa. Oznacza to, że znajomość języka programowania i choćby podstawowych zagadnień algorytmiki jest cennym uzupełnieniem warsztatu każdego, kto zajmuje się nauką lub techniką. Jeśli więc w programie Twoich studiów zabrakło kursu programowania, warto uzupełnić tę lukę!

Trzymasz w rękach znakomity, interdyscyplinarny podręcznik programowania, w którym skupiono się na zastosowaniu kodu Javy do badań z wielu ciekawych dziedzin. Przedstawiono tu zagadnienia podstawowe (zmienne, typy danych, przepływ sterowania, operacje wejścia-wyjścia) oraz bardziej zaawansowane (funkcje, zagadnienia programowania obiektowego, własne typy danych). W książce znalazło się również solidne wprowadzenie do algorytmów i struktur danych z uwzględnieniem charakterystyki wydajności implementacji. Kluczowym elementem wyróżniającym tę publikację spośród innych jest jednak zastosowanie prezentowanych koncepcji do rozwiązania konkretnych problemów nauki i inżynierii.

Robert Sedgewick
jest profesorem informatyki na Uniwersytecie Princeton, jednym z założycieli Wydziału Informatyki tej uczelni. Pracował również w Xerox PARC, Institute for Defense Analyses, INRIA oraz Adobe Systems. Jest autorem wielu książek, naukowo zajmuje się kombinatoryką analityczną, projektowaniem oraz analizą algorytmów i struktur danych.

Kevin Wayne
wykłada na Wydziale Informatyki Uniwersytetu Princeton. Otrzymał tytuł ACM Distinguished Educator. Obronił doktorat w dziedzinie badań operacyjnych i informatyki przemysłowej.

NAJWAŻNIEJSZE ZAGADNIENIA UJĘTE W KSIĄŻCE:

- Podstawowe zasady budowania programów w Javie
- Instrukcje warunkowe, pętle, tablice
- Sterowanie grafiką i dźwiękiem z poziomu Javy
- Funkcje, biblioteki, rekurencja
- Projektowanie API, hermetyzacja, dziedziczenie
- Studia przypadków i przykłady zastosowań w nauce i technice

ISBN 978-83-283-3793-0

Addison-Wesley

ALWAYS LEARNING

PEARSON