



Piotr Wróblewski

ALGORYTMY

struktury danych
i techniki programowania
dla programistów Java

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek 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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/algoja>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5465-4

Copyright © Helion 2019

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	9
Rozdział 1. Zanim wystartujemy	17
Czym powinien się charakteryzować algorytm?	18
Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych	20
— 1804 —	20
— 1830 i później —	21
— 1890 —	21
— lata 30. XX w. —	21
— lata 40. XX w. —	22
— okres powojenny —	22
— 1969 —	23
— teraz —	23
Jak to się niedawno odbyło, czyli o tym, kto „wymyślił” metodologię programowania	24
Proces koncepcji programów	25
Poziomy abstrakcji opisu i wybór języka	26
Modelowanie działania algorytmów (maszyna Turinga)	28
Poprawność algorytmów	29
Zadania	31
Rozwiązania i wskazówki do zadań	31
Rozdział 2. Systemy obliczeniowe i podstawy kodowania	35
System dziesiętny i kilka definicji	36
System dwójkowy	36
Operacje arytmetyczne na liczbach dwójkowych	37
Operacje logiczne na liczbach dwójkowych	38
Kod BCD	40
System ósemkowy	41
System szesnastkowy	41
Kodowanie liczb ze znakiem	42
Kod znak-moduł (ZM)	42
Kod U2 (system uzupełnienia dwójkowego)	42
Zmienne w pamięci komputera	43
Kodowanie znaków	44
Kodowanie obrazów	46
Mapy bitowe na przykładzie formatu BMP	47

Rozdział 3. Rekurencja	51
Definicja rekurencji	51
Ilustracja pojęcia rekurencji	53
Jak wykonują się programy rekurencyjne?	55
Niebezpieczeństwa rekurencji	56
Ciąg Fibonacciego	56
Stack overflow!	58
Pułapek ciąg dalszy	61
Stąd do wieczności	61
Definicja poprawna, ale... ..	62
Typy programów rekurencyjnych	63
Myślenie rekurencyjne	65
Przykład 1. Spirala	66
Przykład 2. Kwadraty „parzyste”	67
Uwagi praktyczne na temat technik rekurencyjnych	69
Zadania	70
Rozwiązania i wskazówki do zadań	73
Rozdział 4. Analiza złożoności algorytmów	79
Definicje i przykłady	80
Jeszcze raz funkcja silnia	83
Zerowanie fragmentu tablicy	87
Wpadamy w pułapkę	89
Różne typy złożoności obliczeniowej	90
Nowe zadanie: uprościć obliczenia!	92
Analiza programów rekurencyjnych	92
Terminologia i definicje	92
Ilustracja metody na przykładzie	94
Rozkład logarytmiczny	95
Przeszukiwanie binarne... tym razem bez matematyki wyższej!	96
Zamiana dziedziny równania rekurencyjnego	97
Funkcja Ackermanna, czyli coś dla smakoszy	97
Złożoność obliczeniowa to nie religia!	99
Techniki optymalizacji programów	100
Zadania	101
Rozwiązania i wskazówki do zadań	102
Rozdział 5. Podstawy modelowania danych	103
Typy proste i złożone	104
Operatory i zmienne	105
Obiektowe typy proste, czyli klasy osłowne	107
Ciągi znaków i napisy	107
Tablice	110
Pojęcie referencji, czyli gdzie te wskaźniki z dawnych lat... ..	113
Programowanie obiektowe jako narzędzie modelowania danych i algorytmów	116
Terminologia	117
Modelowanie danych na przykładzie liczb zespolonych	119
Pola i metody statyczne klas	122
Dziedziczenie własności	123
Struktury rekurencyjne w Javie	126

Rozdział 6. Modelowanie abstrakcyjnych typów danych	129
Abstrakcyjne typy danych	130
Listy jednokierunkowe	131
Tablicowa implementacja list	154
Listy innych typów	160
Listy z iteratorem	164
Podsumowanie	169
Rozdział 7. Struktury danych o dostępie ograniczonym	171
Stos	171
Zasada działania stosu	171
Realizacja programowa stosu	173
Kolejki FIFO	176
Stery i kolejki priorytetowe	179
Zadania	186
Rozwiązania i wskazówki do zadań	186
Rozdział 8. Drzewa i zbiory	189
Drzewa i ich reprezentacje	189
Binarne drzewa poszukiwań (BST)	193
Drzewa binarne i wyrażenia arytmetyczne	199
Uniwersalna struktura słownikowa	205
Drzewa „egzotyczne”	210
Zbiory	211
Zadania	213
Rozwiązania zadań	214
Rozdział 9. java.util, czyli struktury danych dla leniuchów	215
Java i interfejsy	216
Klasa Arrays, operacje na tablicach	217
Klasa Vector, czyli tablice dynamiczne	218
Listy	221
Iteratory, czyli wygodne indeksowanie kolekcji	222
Stos	223
Sortowanie kolekcji	224
Klasa HashSet, czyli szybko do celu	225
Rozdział 10. Algorytmy przeszukiwania	227
Przeszukiwanie liniowe	227
Przeszukiwanie binarne	228
Transformacja kluczowa (hashing)	230
W poszukiwaniu funkcji H	232
Najbardziej znane funkcje H	233
Obsługa konfliktów dostępu	235
Powrót do źródeł	235
Jeszcze raz tablice!	236
Próbkowanie liniowe	237
Podwójne kluczowanie	238
Zastosowania transformacji kluczowej	240
Klasyczne funkcje C/C++ oraz Java	240
Funkcje hashujące a klasy Javy	241
Podsumowanie metod transformacji kluczowej	243

Rozdział 11. Algorytmy sortowania	245
Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	246
Sortowanie bąbelkowe, algorytm klasy $O(N^2)$	248
Sortowanie szybkie (Quicksort) — algorytm klasy $O(N \log N)$	250
Heapsort — sortowanie przez kopcowanie	253
Scalanie zbiorów posortowanych	256
Sortowanie przez scalanie, algorytm klasy $O(N \log N)$	257
Sortowanie zewnętrzne	258
Uwagi praktyczne	262
Rozdział 12. Derekursywacja i optymalizacja algorytmów	263
Jak pracuje kompilator?	264
Odrobina formalizmu nie zaszkodzi!	266
Kilka przykładów derekursywacji algorytmów	267
Derekursywacja z wykorzystaniem stosu	271
Eliminacja zmiennych lokalnych	271
Metoda funkcji przeciwnych	273
Klasyczne schematy derekursywacji	275
Schemat typu while	276
Schemat typu if-else	277
Schemat z podwójnym wywołaniem rekurencyjnym	279
Podsumowanie	281
Rozdział 13. Przeszukiwanie tekstów	283
Algorytm typu brute force	283
Nowe algorytmy poszukiwań	286
Algorytm KMP	286
Algorytm Boyera-Moore'a	290
Algorytm Rabina-Karpa	292
Rozdział 14. Zaawansowane techniki programowania	295
Programowanie typu „dziel i zwyciężaj”	296
Odszukiwanie minimum i maksimum w tablicy liczb	297
Mnożenie macierzy o rozmiarze $N \times N$	300
Mnożenie liczb całkowitych	302
Inne znane algorytmy „dziel i zwyciężaj”	303
Algorytmy „żarłocze”, czyli przekąsić coś nadszedł już czas... ..	303
Problem plecakowy, czyli niełatwe jest życie turysty piechura	304
Wydawanie reszty, czyli „A nie ma pan drobnych?” w praktyce	307
Programowanie dynamiczne	308
Ciąg Fibonacciego	309
Równania z wieloma zmiennymi	311
Najdłuższa wspólna podsekwencja	312
Najdłuższy wspólny podłańcuch	315
Heurystyczne techniki programowania	317
Uwagi bibliograficzne	319
Rozdział 15. Algorytmy grafowe	321
Definicje i pojęcia podstawowe	323
Etykiety i wartości	323
Cykle w grafach	325
Sposoby reprezentacji grafów	328
Reprezentacja tablicowa	328
Słowniki węzłów	330

Listy kontra zbiory	331
Podstawowe operacje na grafach	332
Suma grafów	332
Kompozycja grafów	332
Graf do potęgi	333
Algorytm Roya-Warshalla	334
Algorytm Floyda-Warshalla	339
Algorytm Dijkstry	342
Algorytm Bellmana-Forda	343
Drzewo rozpinające minimalne	344
Algorytm Kruskala	344
Algorytm Prima	345
Przeszukiwanie grafów	346
Strategia „w głąb” (przeszukiwanie zstępujące)	346
Strategia „wszerz”	348
Inne strategie przeszukiwania	350
Problem właściwego doboru	351
Podsumowanie	355
Zadania	355
Rozdział 16. Algorytmy numeryczne	357
Poszukiwanie miejsc zerowych funkcji	358
Iteracyjne obliczanie wartości funkcji	359
Interpolacja funkcji metodą Lagrange’a	360
Różniczkowanie funkcji	362
Całkowanie funkcji metodą Simpsona	363
Rozwiązywanie układów równań liniowych metodą Gaussa	364
Biblioteki naukowe dla Javy	367
Uwagi końcowe	367
Rozdział 17. Kodowanie i kompresja danych	369
Kodowanie danych i arytmetyka dużych liczb	371
Metody prymitywne	372
Kodowanie symetryczne	373
Kodowanie asymetryczne	374
Obliczenia na bardzo dużych liczbach całkowitych	376
Klasa BigInteger	380
Łamanie kodów	381
Jakość klucza szyfrującego	381
Metody łamania szyfrów	381
Techniki kompresji danych	382
Kompresja za pomocą modelowania matematycznego	383
Kompresja metodą RLE	384
Kompresja danych metodą Huffmana	385
Kodowanie LZW	390
Rozdział 18. Czy komputery mogą myśleć?	397
Przegląd obszarów zainteresowań sztucznej inteligencji (SI)	398
Systemy eksperckie	399
Sieci neuronowe	401
Reprezentacja problemów	402
Gry dwuosobowe i drzewa gier	405
Algorytm min-max	407

Rozdział 19. Zadania różne	415
Teksty zadań	415
Rozwiązania	417
Dodatek A. Java — szybki start	421
Instalacja środowiska Java	422
Środowiska IDE do Javy	423
Konfiguracja środowiska Java	425
Systemy pochodne UNIX (np. Linux)	425
System Windows	425
Kompilujemy program w Javie	426
Pakiety w Javie	427
Poznaj Javę w 5 minut!	430
Elementy języka Java na przykładach	431
Sterowanie przebiegiem programu	432
Konwersje typów i wprowadzanie danych	433
Operacje na plikach w Javie	435
Funkcje matematyczne w Javie	437
Literatura	439
Spis rysunków	441
Spis tabel	445
Skorowidz	447

Rozdział 13.

Przeszukiwanie tekstów

Przeszukiwanie tekstów traktuje się jako odrębną dziedzinę z uwagi na szeroką gamę zastosowań praktycznych. Tekst jest tutaj definiowany jako ciąg znaków w sensie informatycznym (nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną”) i może być ciągiem bitów, który oczywiście można interpretować za pomocą umownych kodów (np. ASCII, Unicode) jako jednostki leksykalne mające określone znaczenie dla czytelnika. Wszystko jest zresztą kwestią umowy, w szczególności ciąg bitów może reprezentować... pamięć ekranu (patrz rozdział 2.).

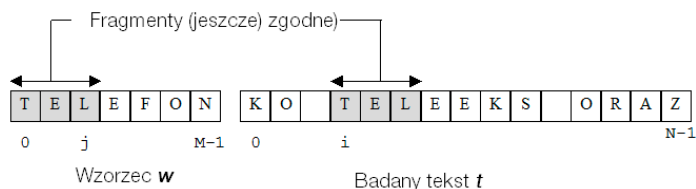
Okazuje się wszelako, że przyjęcie konwencji dotyczących interpretowania informacji ułatwia wiele operacji na niej. Dlatego też pozostanmy przy ogólnikowym stwierdzeniu „tekst”, wiedząc, że za tym określeniem może się kryć sporo znaczeń.

Algorytm typu brute force

Określenie *brute force* w przypadku algorytmów zazwyczaj tłumaczymy: „na siłę” lub „siłowy”, a jeden z moich znajomych pomysłowo przełożył całość jako „metodę mastodonta”, co doskonale odzwierciedla jej nieco „bezmyślny” charakter.

Zadaniem, które będziemy usiłowali wspólnie rozwiązać, jest poszukiwanie wzorca w o długości M znaków w tekście t o długości N (ang. *pattern matching*). Z łatwością możemy zaproponować dość oczywisty algorytm rozwiązujący to zadanie, a bazować będziemy na pomysłach symbolicznie przedstawionych na rysunku 13.1.

Rysunek 13.1.
*Algorytm typu
brute force
przeszukiwania
tekstu*



Zarezerwujmy indeksy j i i do poruszania się odpowiednio we wzorcu i w tekście podczas operacji porównywania znak po znaku zgodności wzorca z tekstem. Załóżmy, że w trakcie poszukiwań obszary objęte szarym kolorem na rysunku okazały się zgodne. Po stwierdzeniu tego faktu przesuwamy się zarówno we wzorcu, jak i w tekście o jedną pozycję do przodu ($i++$; $j++$).

Cóż jednak powinno się stać z indeksami *i* oraz *j* podczas stwierdzenia niezgodności znaków? W takiej sytuacji całe poszukiwanie kończy się porażką, co zmusza do anulowania „szarej strefy” zgodności. Czynimy to poprzez cofnięcie się w tekście o to, co było zgodne, czyli o *j*-1 znaków, wyzerowując przy okazji *j*. Omówię jeszcze moment stwierdzenia całkowitej zgodności wzorca z tekstem. Kiedy to nastąpi? Otóż nietrudno zauważyć, że podczas stwierdzenia zgodności ostatniego znaku *j* powinno się zrównać z *M*. Możemy wówczas łatwo odtworzyć pozycję, od której wzorec startuje w badanym tekście: będzie to oczywiście *i*-*M*.

Tłumacząc powyższe sytuacje na instrukcje Javy, możemy łatwo dojść do następującej procedury:



SzukajTxt.java

Listing

```
class SzukajTxt{
public static int szukaj(char t[], char w[]) { //szukaj wzorca 'w' w tablicy znaków 't'
    int i=0,j=0;
    int M=w.length, N=t.length;
    while( (j<M) && (i<N) ){
        if(t[i]!=w[j]){ // *
            i-=j-1;
            j=-1;
        }
        i++; // **
        j++;
    }
    if(j==M)
        return i-M;
    else
        return -1; // nie znaleziono wzorca
    }
public static void main(String[] args) {
    char t[] = {'a', 'b', 'r', 'a', 'k', 'a', 'd', 'a', 'b', 'r', 'a'}; // jawna tablica
                                                                    // znaków
    String tS = new String(t); // tablica znaków skonwertowana na String
    char w1[] = {'r', 'a', 'k', 'i'}; // wzorec 1
    char w2[] = {'r', 'a', 'k'};
    String w1S = new String(w1); // wzorec 1 jako String
    String w2S = new String(w2);
    System.out.printf("Szukam [%s] w [%s], wynik (pozycja): %d\n", w1S, tS, szukaj
        ↵(t, w1) );
    System.out.printf("Szukam [%s] w [%s], wynik (pozycja): %d\n", w2S, tS, szukaj
        ↵(t, w2) );
    }
}
```

Wynik uruchomienia programu jest następujący:



```
C:\> Szukam [raki] w [abrakadabra], wynik (pozycja): -1
      Szukam [rak] w [abrakadabra], wynik (pozycja): 2
```

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorec, lub -1, gdy poszukiwany tekst nie został odnaleziony — to znana już doskonale konwencja. Program można też nieznacznie zmodyfikować, aby wyszukiwał wszystkie wystąpienia wzorca w tekście — wystarczy w takim przypadku wypisać odnaleziony indeks *i* i kontynuować pętlę aż do osiągnięcia końca tekstu.



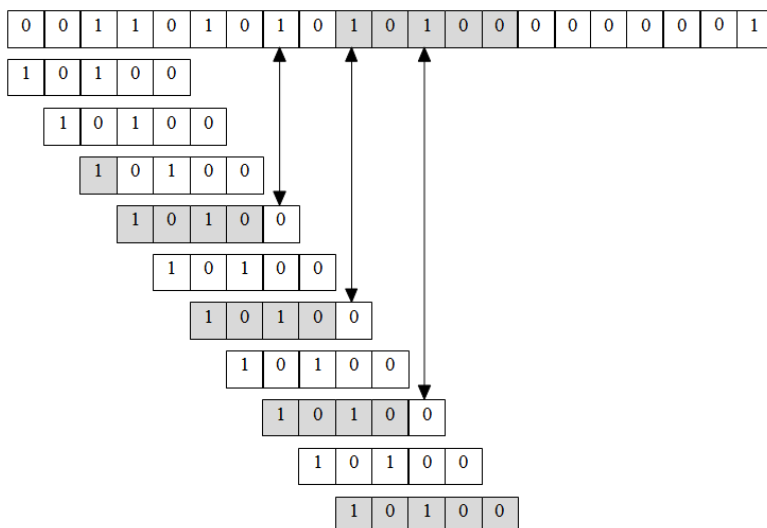
W Javie możesz ewentualnie użyć obiektów klasy `String` zamiast tablic znaków, ale ponieważ w tej klasie znajdują się już gotowe metody pozwalające wyszukiwać wzorce, realizacja algorytmu wyszukiwania byłaby nadmiarowa — w ostatniej linijce można by użyć instrukcji `tS.indexOf(w2S)` zamiast `szukaj(t, w2)`! Niemniej jednak w kolejnych przykładach będę już pomijał tablice znakowe i dla ułatwienia będę stosował obiekty klasy `String` także w parametrach procedur, gdyż znacznie łatwiej można nimi operować w kodzie. W celu dostępu do pojedynczego znaku i napisu `t` będziemy używali instrukcji `t.charAt(i)` zamiast `t[i]`.

Przypatrzmy się teraz dokładniej przykładowi poszukiwania wzorca 10100 w pewnym tekście binarnym (rysunek 13.2).

Rysunek jest nieco uproszczony: w istocie poziome przesuwanie się wzorca oznacza instrukcje zaznaczone na listingu *SzukajTxt.java* jako `(*)`, natomiast cała szara strefa o długości `k` oznacza `k`-krotne wykonanie `(**)`.

Na podstawie zobrazowanego przykładu możemy podjąć próbę wymyślenia takiego najgorszego tekstu i wzorca, dla których proces poszukiwania będzie trwał możliwie najdłużej. Chodzi oczywiście zarówno o tekst, jak i wzorec złożone z samych zer i zakończone jedynką (umawiamy się, że zera i jedynki symbolizują tu dwa różne znaki).

Rysunek 13.2.
„Fałszywe starty”
podczas
poszukiwania



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

Zaprezentowany w tym paragrafie algorytm wykorzystuje komputer jako bezmyślne, ale sprawne liczydło. Jego złożoność obliczeniowa eliminuje go w praktyce z przeszukiwania tekstów binarnych, w których może wystąpić wiele niekorzystnych konfiguracji danych. Jediną zaletą algorytmu jest jego prostota, co i tak nie czyni go wystarczająco atrakcyjnym, by dać się zamęczyć jego powolnym działaniem.

¹ Zwykle M będzie znacznie mniejsze niż N .

Nowe algorytmy poszukiwań

Algorytm, o którym będzie mowa w tym rozdziale, posiada ciekawą historię, którą w formie anegdoty warto przytoczyć. Otóż w roku 1970 Stephen Arthur Cook udowodnił teoretyczny rezultat dotyczący pewnej abstrakcyjnej maszyny. Wynikało z niego, że istniał algorytm poszukiwania wzorca w tekście, który działał w czasie proporcjonalnym do $M+N$ w najgorszym przypadku. Rezultat pracy Cooka wcale nie był przewidziany do praktycznych celów, niemniej Donald Knuth i Vaughan Ronald Pratt otrzymali na jego podstawie algorytm, który można już było zaimplementować w komputerze — ukazując przy okazji, że pomiędzy praktycznymi realizacjami a rozważaniami teoretycznymi nie istnieje wcale aż tak ogromna przepaść, jak by się mogło wydawać. W tym samym czasie James Morris odkrył dokładnie ten sam algorytm jako rozwiązanie problemu, który napotkał podczas praktycznej implementacji edytora tekstu. Algorytm KMP — bo tak będziemy go dalej zwali — jest jednym z przykładów dość częstych w nauce odkryć równoległych: z jakichś niewiadomych powodów nagle kilku pracujących osobno ludzi dochodzi do tego samego dobrego rezultatu. Prawda, że jest w tym coś niesamowitego i aż się prosi o jakieś metafizyczne hipotezy?

Knuth, Morris i Pratt opublikowali swój algorytm dopiero w 1976 r. Tymczasem pojawił się kolejny „cudowny” algorytm, tym razem autorstwa Roberta Boyera i J Strothera Moore’a, który okazał się w pewnych zastosowaniach znacznie szybszy od algorytmu KMP. Został on także równolegle wynaleziony (odkryty?) przez Billa Gospera. Oba te algorytmy są jednak dość trudne do zrozumienia bez pogłębionej analizy, co utrudniło ich rozpropagowanie.

W roku 1980 Richard Karp i Michael Rabin doszli do wniosku, że przeszukiwanie tekstów nie jest aż tak dalekie od standardowych metod przeszukiwania, i wynaleźli algorytm, który — działając ciągle w czasie proporcjonalnym do $M+N$ — jest ideowo zbliżony do poznanego już algorytmu typu *brute force*. Na dodatek jest to algorytm, który można względnie łatwo uogólnić na przypadek poszukiwania w tablicach dwuwymiarowych, co sprawia, że jest potencjalnie użyteczny w obróbce obrazów.

W następnych trzech punktach szczegółowo omówię algorytmy wspomniane w tym historycznym przeglądzie.

Algorytm KMP

Wadą algorytmu *brute force* jest jego uwrażliwienie na konfigurację danych: fałszywe restarty są tu bardzo kosztowne; w analizie tekstu cofamy się o całą długość wzorca, zapominając po drodze o wszystkim, co przetestowaliśmy do tej pory. Narzuca się tu niejako chęć skorzystania z informacji, które już w pewien sposób posiadamy — przecież w następnym etapie będą wykonywane częściowo te same porównania co poprzednio!

W pewnych szczególnych przypadkach przy znajomości struktury analizowanego tekstu możliwe jest ulepszenie algorytmu. Jeśli przykładowo wiemy na pewno, że w poszukiwanym wzorcu pierwszy znak w ogóle się nie pojawia², to w razie restartu nie musimy cofać wskaźnika i o $j-1$ pozycji, jak to było poprzednio (patrz listing *SzukajTxt.java*). W tym przypadku możemy po prostu inkrementować i , wiedząc, że ewentualne powtórzenie poszukiwań na pewno nic by nie dało. Owszem, można się łatwo zgodzić z twierdzeniem, że tak wyspecjalizowane teksty zdarzają się relatywnie rzadko, jednak powyższy przykład ukazuje, iż ewentualne manipulacje algorytmami poszukiwań są ciągle możliwe — wystarczy się tylko rozejrzeć. Idea algorytmu KMP polega na wstępnym zbadaniu wzorca w celu obliczenia liczby pozycji, o które należy cofnąć wskaźnik i w przypadku stwierdzenia niezgodności

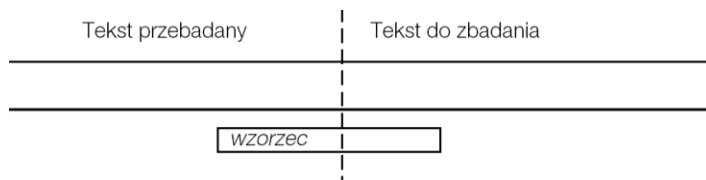
² Przykład: „ABBBBBBB” — znak „A” wystąpił tylko jeden raz.

badanego tekstu ze wzorcem. Oczywiście można również rozumować w kategoriach przesuwania wzorca do przodu — rezultat będzie ten sam. To właśnie tę drugą konwencję będziemy stosować dalej. Wiemy już, że powinniśmy przesuwać się po badanym tekście nieco inteligentniej niż w poprzednim algorytmie. W przypadku zauważenia niezgodności na pewnej pozycji j wzorca³ należy zmodyfikować ten indeks, wykorzystując informację zawartą w już zbadanej „szarej strefie” zgodności.

Brzmi to wszystko zapewne niesłychanie tajemniczo, pora więc jak najszybciej wyjaśnić tę sprawę, aby uniknąć możliwych nieporozumień. Popatrzymy w tym celu na rysunek 13.3.

Rysunek 13.3.

Wyszukiwanie optymalnego przesunięcia w algorytmie KMP



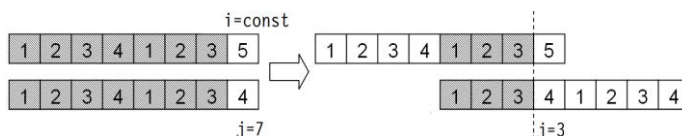
Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorec bardzo wolno w prawo, patrząc jednocześnie na już zbadany tekst — tak aby obserwować ewentualne pokrycie się tej części wzorca, która znajduje się po lewej stronie przerywanej kreski, z tekstem, który jest umieszczony powyżej wzorca. W pewnym momencie może się okazać, że następuje pokrycie obu tych części. Zatrzymujemy wówczas przesuwanie i kontynuujemy testowanie (znak po znaku) zgodności obu części znajdujących się za pionową kreską.

Od czego zależy ewentualne pokrycie się oglądanych fragmentów tekstu i wzorca? Otóż dość paradoksalnie badany tekst nie ma tu nic do powiedzenia — jeśli można to tak określić. Informacja o tym, jaki był, jest ukryta w stwierdzeniu „ $j-1$ znaków było zgodnych” — w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie wzorec, odkryć poszukiwane optymalne przesunięcie. Na tym właśnie spostrzeżeniu opiera się idea algorytmu KMP. Okazuje się, że badając samą strukturę wzorca, można obliczyć, jak powinniśmy zmodyfikować indeks j w razie stwierdzenia niezgodności tekstu ze wzorcem na j -tej pozycji.

Zanim zagłębimy się w wyjaśnienia na temat obliczania tych przesunięć, popatrzymy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 13.4 możemy dostrzec, że na siódmej pozycji wzorca⁴ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

Rysunek 13.4.

Przesuwanie się wzorca w algorytmie KMP (1)



Jeśli zostawimy indeks i w spokoju, to modyfikując wyłącznie j , możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? Przesuwając go wolno w prawo (rysunek 13.4), doprowadzamy w pewnym momencie do nałożenia się ciągów 123 przed kreską — cała strefa niezgodności została wyprowadzona na prawo i ewentualne dalsze testowanie może być kontynuowane!

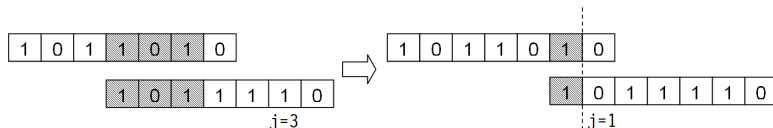
Analogiczny przykład znajduje się na rysunku 13.5.

³ Lub i w przypadku badanego tekstu.

⁴ Licząc indeksy tablicy tradycyjnie od zera.

Rysunek 13.5.

Przesuwanie
się wzorca
w algorytmie
KMP (2)



Tym razem niezgodność wystąpiła na pozycji $j=3$. Wykonując podobnie jak poprzednio przesuwanie wzorca w prawo, zauważamy, że jedyne możliwe nałożenie się znaków wystąpi po przesunięciu o dwie pozycje w prawo — czyli dla $j=1$. Dodatkowo okazuje się, że znaki za kreską też się pokryły, ale o tym algorytm „dowie się” dopiero podczas kolejnego testu zgodności na pozycji i .

Dla algorytmu KMP konieczne okazuje się wprowadzenie tablicy przesunień `int shift[M]`, gdzie M jest długością wzorca. Sposób jej zastosowania będzie następujący: jeśli na pozycji j wystąpiła niezgodność znaków, kolejną wartością j będzie `shift[j]`. Nie wnikając chwilowo w sposób inicjowania tej tablicy (odmiennej oczywiście dla każdego wzorca), możemy natychmiast podać algorytm KMP, który w konstrukcji jest niemal dokładną kopią algorytmu typu *brute force*:



Listing

KMP.java (fragment)

```
class KMP{
    public static int kmp(String w, String t, int shift[]) {
        int i,j;
        int N = t.length();
        int M = shift.length;
        for(i=0,j=0; (i<N) && (j<M); i++,j++)
            while( (j>=0) && (t.charAt(i)!=w.charAt(j)) )
                j=shift[j];
        if (j==M)
            return i-M;
        else
            return -1;
    }

    public static void main(String[] args) {
        String t = new String("abcd1010efg");
        String w1 = new String("1010");
        String w2 = new String("1011");
        int shift1[] = new int[ w1.length() ];
        init_shifts(w1, shift1);
        System.out.printf("Szukam [%s] w [%s], wynik (pozycja): %d\n", w1, t, kmp
            ↪(w1,t,shift1) );
        int shift2[] = new int[ w2.length() ];
        init_shifts(w2, shift2);
        System.out.printf("Szukam [%s] w [%s], wynik (pozycja): %d\n", w2, t, kmp
            ↪(w2,t,shift2) );
    }
}
```

Wynik uruchomienia programu jest następujący:



```
C:\> Szukam [1010] w [abcd1010efg], wynik (pozycja): 4
      Szukam [1011] w [abcd1010efg], wynik (pozycja): -1
```

Szczególnym przypadkiem jest wystąpienie niezgodności na pozycji zerowej: z założenia niemożliwe jest tu przesuwanie wzorca w celu uzyskania nałożenia się znaków. Z tego

powodu chcemy, aby indeks j pozostał niezmienny, przy jednoczesnej progresji indeksu i . Jest to możliwe do uzyskania, jeśli umówimy się, że $\text{shift}[0]$ zostanie zainicjowany wartością -1 . Wówczas podczas kolejnej iteracji pętli `for` nastąpi inkrementacja i oraz j , co wyzeruje j .

Pozostaje do omówienia sposób konstrukcji tablicy $\text{shift}[M]$. Jej obliczenie powinno nastąpić przed wywołaniem funkcji `kmp`, co sugeruje, że w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona niemal identyczna z `kmp`, z tą tylko różnicą, że algorytm sprawdza zgodność wzorca... z nim samym!

```
public static void init_shifts(String w, int shift[]) {
    int i,j;
    int M = shift.length;
    shift[0]=-1;
    for(i=0,j=-1; i < M-1; i++, j++, shift[i]=j )
        while( (j>=0)&&( w.charAt(i)!=w.charAt(j) ) )
            j=shift[j];
}
```

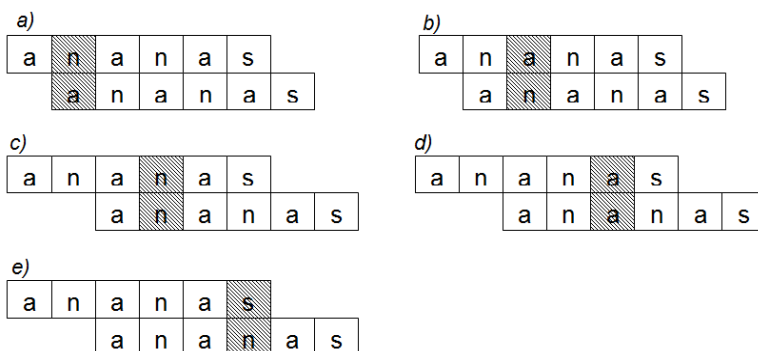
Sens tego algorytmu jest następujący: tuż po inkrementacji i oraz j wiemy, że pierwsze j znaków wzorca jest zgodne ze znakami na pozycjach $p[i-j-1] \dots p[i-1]$ (ostatnie j pozycji w pierwszych i znakach wzorca). Ponieważ jest to największe j spełniające powyższy warunek, zatem aby nie ominąć *potencjalnego* miejsca wykrycia wzorca w tekście, należy ustawić $\text{shift}[i]$ na j .

Popatrzmy, jaki będzie efekt zadziałania funkcji `init_shifts` na słowie *ananas* (rysunek 13.6). Zacięniowane litery oznaczają miejsca, w których wystąpiła niezgodność wzorca z tekstem. W każdym przypadku graficznie przedstawiono efekt przesunięcia wzorca — widać wyrażenie, które strefy pokrywają się przed strefą zacięniowaną (porównaj z rysunkiem 13.5).

Przypomnijmy jeszcze, że tablica `shift` zawiera nową wartość dla indeksu j , który przemieszcza się po wzorcu.

Rysunek 13.6.

Optymalne przesunięcia wzorca „ananas” w algorytmie KMP



Algorytm KMP działa w czasie proporcjonalnym do $M+N$ w najgorszym przypadku. Największy zauważalny zysk związany z jego użyciem dotyczy przypadku tekstów o wysokim stopniu samopowtarzalności — dość rzadko występujących w praktyce. Dla typowych tekstów zysk związany z wyborem metody KMP będzie zatem słabo zauważalny.

Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik i w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez

Jedną z możliwych wersji algorytmu przedstawiam poniżej (nazwy parametrów i kluczowych zmiennych są identyczne jak w KMP, co powinno ułatwić analizę kodu). Zaczniemy od tablicy inicjalizacji przesunięć:



Listing

BM.java (fragment)

```
class BM{
    static int MAX = 256; // liczba znaków alfabetu (pomijamy polskie znaki)

    static void init_shifts(String w, int shift[]) {
        int i;
        for (i = 0; i < MAX; i++) // inicjalizacja
            shift[i] = -1;
        // zapiszmy ostatnie wystąpienie każdego możliwego znaku wzorca w tablicy o rozmiarze równym
        // rozmiarowi alfabetu. Przy braku znaku w tablicy umożliwi nam to przesunięcie o całą długość
        // wzorca!
        for (i = 0; i < w.length(); i++)
            shift[ (int) w.charAt(i) ] = i;
    }
}
```



Uwaga

Uwaga: aby nie komplikować kodu, pominąłem polskie znaki z powodu ich kodowania na dwóch bajtach (Unicode) — w przypadku wystąpienia polskiego znaku w napisie konwersja `char` na `int` nie uda się i program zgłosi wyjątek podczas pobierania znaku.

Teraz przejdziemy do prezentacji listingu algorytmu z przykładem wywołania:

```
static void bm(String t, String w) {
    int N = t.length();
    int M = w.length();

    int shift[] = new int[MAX];
    init_shifts(w, shift); // inicjalizacja tablicy przesunięć dla wzorca 'w'

    int przes = 0; // aktualne przesunięcie wzorca 'w' względem tekstu
    while(przes <= (N - M)) { // sytuacja, gdy wzorec będzie znajdował się na samym końcu
        int j = M-1;
        // przesuwanie indeksu 'j' (pozycja dopasowania), gdy porównywane znaki są równe:
        while(j >= 0 && w.charAt(j) == t.charAt(przes+j))
            j--;
        // przesuwanie wzorca tak, aby kolejny znak zrównał się z ostatnim swoim wystąpieniem we wzorcu:
        if (j < 0) {
            System.out.println("Znalazłem dopasowanie na pozycji: " + przes);
            przes += (przes+M < N)? M-shift[t.charAt(przes+M)] : 1;
        } else // przesuwanie do ostatniej niezgodności z uwzględnieniem możliwego końca tekstu
            przes += max(1, j - shift[t.charAt(przes+j)]);
    }
}

public static void main(String []args) {
    String t = "Z pamiętnika młodej lekarki podającej w tym momencie pacjentowi nowe
↳ lekarstwo";
    String w = "lek";
    System.out.println(t);
    bm(t, w); // program znajdzie dopasowanie na pozycjach: 20 i 69
}
```

Algorytm Boyera-Moore'a, podobnie jak KMP, jest klasy $M+N$, jednak jest o tyle od niego lepszy, że w przypadku krótkich wzorców i długiego alfabetu kończy się po ok. M/N porównaniach. W celu obliczenia optymalnych przesunięć⁶ autorzy algorytmu proponują połączenie powyższego algorytmu z zaproponowanym przez Knutha, Morrisa i Pratta. Celowość tego zabiegu wydaje się jednak wątpliwa, gdyż optymalizując sam algorytm, można bardzo łatwo sprawić, że proces prekompilacji wzorca stanie się zbyt czasochłonny.

Algorytm Rabina-Karpa

Ostatni algorytm do przeszukiwania tekstów, który przeanalizuję, wymaga znajomości rozdziału 10. i terminologii transformacji kluczowej, która została w nim przedstawiona.

Algorytm Rabina-Karpa polega na dość przewrotnej idei:

- ◆ Wzorec w (do odszukania) jest *kluczem* o długości M znaków, charakteryzującym się pewną wartością wybranej przez nas funkcji H . Możemy zatem obliczyć jednokrotnie $H_w = H(w)$ i korzystać z tego wyliczenia w sposób ciągły.
- ◆ Tekst wejściowy t (do przeszukania) może być odczytywany w taki sposób, aby na bieżąco znać M ostatnich znaków⁷. Z tych M znaków wyliczamy na bieżąco $H_t = H(t)$.

Gdy założymy jednoznaczność wybranej funkcji H , sprawdzenie zgodności wzorca z aktualnie badanym fragmentem tekstu sprowadza się do odpowiedzi na pytanie: czy H_w jest równe H_t ? Jeśli jesteś spostrzegawczy, masz prawo pokreślić w tym miejscu z powątpiewaniem głową i stwierdzić, że przecież to nie ma prawa działać szybko! Istotnie pomysł wyliczenia dodatkowo funkcji H dla każdego słowa wejściowego o długości M wydaje się tak samo kosztowny — jeśli nie bardziej — jak zwykle sprawdzanie tekstu znak po znaku (np. stosując algorytm siłowy typu *brute force*). Tym bardziej, że do tej pory nie powiedziałem ani słowa na temat funkcji H ! W rozdziale 10. mogliśmy się przekonać, że jej wybór wcale nie jest taki oczywisty.

Omawiany algorytm jednak istnieje i na dodatek działa szybko! Aby zatem to wszystko, co poprzednio zostało napisane, logicznie się łączyło, potrzebny będzie jakiś trik. Sztuka polega na właściwym wyborze funkcji H . Robin i Karp wybrali taką funkcję, która dzięki swym szczególnym właściwościom umożliwia dynamiczne wykorzystywanie wyników obliczeń wykonanych krok wcześniej, co znacząco może uprościć obliczenia wykonywane w kroku bieżącym.

Założmy, że ciąg M znaków będziemy interpretować jako pewną liczbę całkowitą. Przyjmując za b jako podstawę systemu liczbę wszystkich możliwych znaków, otrzymamy:

$$x = t[i]b^{M-1} + t[i+1]b^{M-2} + \dots + t[i+M-1].$$

Przesuńmy się teraz w tekście o jedną pozycję do przodu i zobaczmy, jak zmieni się wartość x :

$$x' = t[i+1]b^{M-1} + t[i+2]b^{M-2} + \dots + t[i+M].$$

Jeśli dobrze przyjrzyś się x i x' , zobaczysz, że wartość x' jest w dużej części zbudowana z elementów tworzących x pomnożonych przez b z uwagi na przesunięcie. Nietrudno wówczas wywnioskować, że:

$$x' = (x - t[i]b^{M-1}) + t[i+M].$$

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

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

Jako funkcji H użyjemy klasycznej realizacji $H(x) = x \% p$, gdzie p jest dużą liczbą pierwszą. Założymy, że dla danej pozycji i wartość $H(x)$ jest znana. Po przesunięciu się w tekście o jedną pozycję w prawo pojawia się konieczność wyliczenia wartości funkcji $H(x')$ dla tego „nowego” słowa. Czy faktycznie trzeba powtarzać całe wyliczenie? Być może istnieje pewne ułatwienie bazujące na zależności, jaka istnieje pomiędzy x i x' ?

Z pomocą przychodzi tu własność funkcji modulo użytej w wyrażeniu arytmetycznym. Można oczywiście obliczyć modulo z wyniku końcowego, lecz to bywa czasami niewygodne, np. z uwagi na wielkość liczby, z którą mamy do czynienia, a poza tym gdzie tu byłby zysk szybkości?! Jednak identyczny wynik otrzymuje się, aplikując funkcję modulo po każdej operacji cząstkowej i przenosząc otrzymaną wartość do następnego wyrażenia cząstkowego! Dla przykładu weźmy obliczenie:

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

Wynik ten jest oczywiście prawdziwy, co można łatwo sprawdzić na kalkulatorze. Identyczny rezultat da jednak następująca sekwencja obliczeń:

$$(5 \cdot 100) \% 7 = 3 \rightarrow (3 + 6 \cdot 100) \% 7 = 0 \rightarrow (0 + 8) \% 7 = 1,$$

co jest też łatwe do weryfikacji.

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



Listing

RK.java

```
class RK{
    static int MAX = 256; // liczba znaków alfabetu
    static int p = 33554393; // duża liczba pierwsza

    static void rk(String w, String t) {
        int M = w.length();
        int N = t.length();
        int i, j;
        int Hw = 0; // funkcja H dla wzorca
        int Ht = 0; // funkcja H dla tekstu
        int bM_1 = 1; // wyliczymy wartość pow(MAX, M-1)%p
        for (i = 0; i < M-1; i++)
            bM_1 = (MAX*bM_1) % p;

        for (i = 0; i < M; i++) {
            Hw = (Hw*MAX + w.charAt(i)) % p; // inicjacja funkcji H dla wzorca
            Ht = (Ht*MAX + t.charAt(i)) % p; // inicjacja funkcji H dla tekstu
        }

        for (i = 0; i <= N - M; i++) { // właściwa pętla przeszukiwania
            if (Hw == Ht) { // tutaj już musimy sprawdzić znak po znaku i potwierdzić, czy wyszukanie
                // się powiodło
                for (j = 0; j < M; j++) { // (*)
                    if (t.charAt(i+j) != w.charAt(j))
                        break;
                }
                if (j == M)
                    System.out.println("Znalazłem wzorzec na pozycji: " + i);
            }
            if (i < N-M) { // kontynuujemy pod warunkiem, że pozostały fragment tekstu jest dłuższy
                // niż wzorzec
                Ht = (MAX*(Ht - t.charAt(i)*bM_1) + t.charAt(i+M)) % p;
            }
        }
    }
}
```

```

        if (Ht < 0) //korekta znaku (**)
            Ht = (Ht + p);
    }
}

public static void main(String[] args) {
    String t = "Z pamiętnika młodej lekarki podającej w tym momencie pacjentowi nowe
    ↳ lekarstwo";
    String w = "lek";
    System.out.println(t);
    System.out.println("Szukam: " + w);
    rk(w, t);
}
}

```

Na pierwszym etapie następuje wyliczenie początkowych wartości H_t i H_w . Nie dotyczy to jednak aktualnie badanego fragmentu tekstu — tutaj wartość H_t ulega zmianie podczas każdej inkrementacji zmiennej i . Do obliczenia $H(x')$ możemy wykorzystać omówioną wcześniej własność funkcji modulo. Dodatkowego wyjaśnienia wymaga być może linia listingu oznaczona (**), pozwalająca uniknąć przypadkowego wskoczenia w liczby ujemne. Gdyby istotnie tak się stało, przeniesiona do następnego wyrażenia arytmetycznego wartość modulo byłaby nieprawidłowa i sfalszowałaby końcowy wynik!

Kolejne uwagi dotyczą parametrów p i MAX . Zaleca się, aby p było dużą liczbą pierwszą⁸, jednakże nie można tu przesadzać z uwagi na możliwe przekroczenie zakresu pojemności użytych zmiennych. W przypadku wyboru dużego p zmniejszamy prawdopodobieństwo wystąpienia kolizji spowodowanej niejednoznacznością funkcji H . Możliwość ta, mimo że mało prawdopodobna, ciągle istnieje i dlatego po upewnieniu się, że $H_w == H_t$ sprawdzamy już klasycznie (znak po znaku), czy wyszukanie się powiodło — linia oznaczona (**).

Co zaś się tyczy wyboru podstawy systemu (oznaczonej w programie jako MAX), warto wybrać liczbę nawet nieco za dużą, zawsze jednak będącą potęgą liczby 2. Możliwe jest wówczas zaimplementowanie operacji mnożenia przez MAX jako przesunięcia bitowego — wykonywanego przez komputer znacznie szybciej niż zwykle mnożenie. Przykładowo dla $M = 64$ możemy zapisać mnożenie $MAX * p$ jako $p << 6$.

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

Wynik uruchomienia programu jest następujący:

C:_

```

Z pamiętnika młodej lekarki podającej w tym momencie pacjentowi nowe lekarstwo
Szukam: lek
Znalazłem wzorzec na pozycji: 20
Znalazłem wzorzec na pozycji: 69

```



Uwaga

Podobnie jak poprzednio, aby nie komplikować kodu, w napisach i wzorcach wyszukiwania pominąłem polskie znaki z powodu ich kodowania na dwóch bajtach (Unicode). Pełna obsługa polskich znaków wymagałaby napisania dodatkowej funkcji pośredniczącej, tłumaczącej kod polskiego znaku diakrytycznego na liczbę (indeks) w ramach przyjętego alfabetu mającego określoną długość i obejmującego wybrany zakres dozwolonych elementów (np. znaki $a \dots z$, $A \dots Z$, liczby i polskie znaki narodowe).

⁸ W naszym przypadku jest to liczba 33 554 393.

Skorowidz

A

- abakus, 20
- Adleman Leonard, 375
- Aiken Howard, 22
- alfabet
 - Braille'a, 383
 - Morse'a, 382
- algebra Boole'a, 36, 37
- algorytm, 17, 18
 - 3DES, 374, 380
 - A*, 406
 - Bellmana-Forda, 343
 - Boyera-Moore'a, 290, 291, 292
 - optymalizacja, 292
 - tablica przesunięć, 291
 - brute force, 283, 286
 - cechy, 18
 - cięcie α - β , 406, 408, 414
 - czas wykonania, 80, 82, 84, 87, 89, 99, 183, 308
 - DES, 374
 - deterministyczny, 19
 - Dijkstry, 342, 343, 344
 - efektywność, 19
 - Euklidesa, 26
 - Fleury'ego, 327, 328
 - Floyda, *Patrz*: algorytm Floyda-Warshalla
 - Floyda-Warshalla, 339, 340, 341
 - generowanie automatyczne, 30
 - genetyczny, 319, 326
 - Heapsort, *Patrz*: sortowanie przez kopcowanie
 - heurystyczny, 317
 - Huffmana, 385, 387, 388
 - implementacja, 389, 390
 - iteracyjny, 51
 - klasa, 85, 86, 96, *Patrz też*: algorytm
 - złożoność teoretyczna
 - KMP, 286, 289, 291, 292
 - optymalizacja, 286, 288, 289
 - tablica przesunięć, 288, 289
 - kompresji, 383
 - Kruskala, 344
 - kryteria wyboru, 79
 - LZW, 391, 392, 393
 - mini-max, 406, 407, 414
 - niedeterministyczny, 19
 - niestabilny, 69
 - numeryczny, 357, 358, 359, 361, 362
 - całkowanie, 363
 - opis słowny, 26
 - parametr, 56, 63
 - poprawność, 19, 29, 30, 63, 79
 - dowód, 30
 - poziom abstrakcji, 26
 - prefiksowy, 385
 - prezentacja, 26
 - Prima, 344, 345
 - prostota, 79, 99
 - przeszukiwania, 225,
 - Patrz też*: przeszukiwanie binarnego, 228, 229
 - tablicy, 86, 227
 - Quicksort, *Patrz*: sortowanie szybkie
 - Rabina-Karpa, 292
 - klucz, 292
 - rekurencyjny, 51, 52, 55, 63, 69, 92,
 - Patrz też*: rekurencja
 - kwadrat parzysty, 67
 - nieskończona liczba wywołań, 61
 - poziom, 56, 59, 65
 - spirala, 66
 - zajętość pamięci, 59, 65, 69
 - zakończenie, 52, 54, 61, 69
 - Roya-Warshalla, 334, 335, 336
 - RSA, 375, 380

algorytm

- skończony, 19
 - sortowania, 245, *Patrz też:* dane sortowanie, sortowanie
 - kryteria wyboru, 262
 - SSS*, 406
 - Strassena, 301
 - strategia przeszukiwania, 406
 - warunek
 - końcowy, 30
 - wejściowy, 19, 30
 - zajętość pamięci, 80
 - złożoność
 - czasowa, 80
 - obliczeniowa, 79, 80, 90, 91, 95, 96, 99, 299
 - pamięciowa, 80
 - praktyczna, 84, 87, 299
 - teoretyczna, 85
 - żarłoczny, 303, 305, 307
- algorytmika, 20, 129
- analiza bezpieczeństwa sieci, 322
- AND, 38
- architektura warstwowa, 23
- arytmetyka dużych liczb, 376, 380
- assembler, 26
- atak siłowy, 381
- automat skończony, 325

B

- Babbage Charles, 21, 22
- bajt, 37
- BCD, *Patrz:* kod BCD
- B-drzewo, 210
- Bellman Richard Ernest, 308, 343
- BFS, *Patrz:* przeszukiwanie wszerz, przeszukiwanie grafu wszerz
- biblioteka, 428
- CryptPak406.zip, 382
 - java.io, 428
 - java.lang, 428, 437
 - java.util, 215, 230, 428
 - tworzenie, 428
- błąd
- Segmentation fault, 61
 - Stack overflow, 61
- BM.java, 291
- Boyer Robert, 286
- breadth-first search, *Patrz:* przeszukiwanie wszerz, przeszukiwanie grafu wszerz
- break, 433
- BST, 193, 194
- korzeń, 194
 - rekurencja, 195, 196

węzeł

- usuwanie, 197
 - wstawianie, 193, 195
- BULL Gamma3, 22
- bytecode, 105

C

- całkowanie funkcji metodą Simpsona, 363
- Carnot Lazare, 20
- Case Shaun, 385
- ciąg
- Fibonnaciego, 56, 57, 101, 309
 - znaków, 107
 - długość, 110
 - najdłuższa wspólna podsekwencja, 312, *Patrz też:* funkcja LCS reprezentacja, 43
 - najdłuższy wspólny podłańcuch, 315
- compareTo, 151
- continuous integration, *Patrz:* system ciągłej integracji
- Cook Stephen Arthur, 286
- COPACOBANA, 374
- CPU, *Patrz:* procesor
- cykl Eulera, 327
- czas jednostkowy wykonania instrukcji, 89

D

- DAG, *Patrz:* graf skierowany acykliczny
- dane
- bezpieczeństwo, 369
 - kodowanie, *Patrz:* kodowanie, kompresja
 - kompresja, *Patrz:* kompresja, kodowanie
 - rozmiar, 81
 - sortowanie, 185, 245, *Patrz też:* algorytm sortowania, sortowanie
 - struktura, 103, 130
 - typ, *Patrz:* typ
- DARPA, 398
- debugger, 30
- depth-first search, *Patrz:* przeszukiwanie w głąb, przeszukiwanie grafu w głąb
- derekursywacja, 263, 266, 273
- z wykorzystaniem stosu, 271, 272
- DFS, *Patrz:* przeszukiwanie w głąb, przeszukiwanie grafu w głąb
- Diffie Whitfield, 375
- digraf, 324
- Dijkstra Edseger, 24, 30, 342
- directed acyclic graph, *Patrz:* graf skierowany acykliczny
- directed graph, *Patrz:* graf skierowany
- double, 358
- DrawLine, 67

drzewo, 189
 B-drzewo, 210
 binarne, 130, 180, 190, 191, 192, 199
 kompletne, 180
 postać wrostkowa, 203
 poszukiwań, *Patrz:* BST
 ścieżka, 196
 wysokość, 191
 czerwono-czarne, 210
 decyzyjne, 210
 gry, 405, 406
 implementacja, 192
 klucz, 190, 193
 kodowe, 386, 387
 korzeń, 189
 krawędź, 189, 190
 liść, 190
 m-drzewo, 190
 nieregularne, 190
 ojciec, 191
 potomek, 191
 realizacja tablicowa, 192, 193
 rozpinające minimalne, 344
 syn, 191
 ścieżka, 190, 191
 uporządkowane, 190
 USS, *Patrz:* USS
 węzeł, 189, 190
 końcowy, 190
 poziom, 191
 stopień, 190
 usuwanie, 197
 wewnętrzny, 190, 191
 wstawianie, 193, 195
 zewewnętrzny, 190, 191
 wysokość, 191, 195
 zbiór, *Patrz:* las
 zupełne, 190
 Dziedzic.java, 125
 dziedziczenie, 123, 149
 private, 123, 124
 protected, 123, 124
 public, 123
 dziel i zwyciężaj, 257, 258, 296, 297, 298, 309
 mnożenie
 liczb całkowitych, 302
 macierzy, 300
 Quicksort, 303
 dzielnik największy wspólny, *Patrz:* NWD

E

Eckert John Presper, 22
 EDVAC, 22
 eliminacja Gaussa, *Patrz:* metoda Gaussa
 end-recursion, *Patrz:* rekurencja terminalna

ENIAC, 22
 Euklides, 17, 72
 Euler Leonhard, 321
 Eulera
 cykl, *Patrz:* graf cykl Eulera
 twierdzenie, *Patrz:* twierdzenie Eulera

F

Fibonacci, 57
 ciąg, *Patrz:* ciąg Fibonnaciego
 FIFO, *Patrz:* kolejka FIFO
 First In First Out, *Patrz:* kolejka FIFO
 Flawiusz Józef, 164
 float, 358
 Floyd Robert, 24
 Ford Lester Randolph, 343
 format
 BMP, 47, 48
 GIF, 47, 371, 391, 393, 394, 396
 GIF87a, 393
 GIF89a, 393
 ICO, 47
 JPEG, 47
 LZW, 371, 392, 395
 PNG, 47
 SVG, 47
 Fortran, 367
 fraktal, 51
 funkcja
 Ackermanna, 97, 98
 całkowanie, 363
 czas wykonania, 84
 H, *Patrz:* funkcja mieszająca
 hashująca, *Patrz:* funkcja mieszająca
 heurystyczna, 318
 interpolacja, 360
 LCS, 313, 314
 matematyczna, 437
 McCarthy'ego, 59, 69
 miejsce zerowe, 358
 mieszająca, 231, 232, 233, 243
 algorytm Rabina-Karpa, 292, 294
 Java, 241
 kodowanie liter, 233
 konflikt dostępu, 235, 236, 237
 mnożenie, 235
 podwójne kluczowanie, 239
 próbkowanie liniowe, 237, 238, 243
 przykłady, 240
 suma modulo, 293
 suma modulo 2, 233
 suma modulo $R_{\max}$, 234
 tablica, 236
 zastosowania, 240
 modulo, 293, 376, 379

funkcja

O, *Patrz*: notacja O
 obliczanie wartości, 359
 odwrotna, 274, 280
 pierwiastek, *Patrz*: funkcja miejsce zerowe
 pop, 172, 173
 postać uwikłana, 359
 przeciwna, *Patrz*: funkcja odwrotna
 push, 172, 173
 różniczkowanie, 362, 363
 silnia, 55, 56, 64, 81, 83, 94, 269
 wywołanie przez wartość, 63
 XOR, 372

G

generator liczb losowych, 19

GNU Scientific Library, 367

Gödel Kurt, 21

Goldman Alan, 327

Gosper Ralph William, 286

gra

dwuosobowa, 405
 Go, 406
 kółko i krzyżyk, 406, 407, 408, 412
 Reversi, 406
 szachy, 406

graf, 130, 189, 321

0-regularny, 324

automatów skończonych, 325

cykl, 325

Eulera, 326, 327

Hamiltona, 325, 326, 403

skierowany, 325

DAG, *Patrz*: graf skierowany acykliczny

diagonalny, 333

digraf, 324

domknięcie przechodnie, 334

eulerowski, 326

implementacja, 329

zbiór, 331

kompozycja, 332

krawędź, 323, 325

liczba chromatyczna, 324

minimalizowanie konfliktów, 351

nieskierowany, 324, 344

planarny, 323

potęgowanie, 333

problem doboru, 351, 352

przechodni, 334

przeszukiwanie, *Patrz*: przeszukiwanie grafu

regularny, 324

relacja binarna, 333

reprezentacja, 328

słownik węzłów, 346

tablica dwuwymiarowa, 328

tablicowa, 346

skierowany, 324, 325

spójny, 324

stanu, 405

suma, 332

symetryczny, 333

ścieżka, 324

węzeł, 323, 331

wierzchołek, 323, 324

zastosowania, 322

zredukowany, 324

grafika

rastrowa, 46, 47, 48

wektorowa, 47

GrafTabl.java, 330

greedy, *Patrz*: algorytm żarłocznyGSL, *Patrz*: GNU Scientific Library**H**

Hahn Markus, 382

Hamiltona cykl, *Patrz*: graf cykl Hamiltona

Hart Peter, 350

hashcod, 115

hashing, 230, 231

heap, *Patrz*: sterta

Hellman Martin, 375

hermetyzacja, 113, 118

heurystyka, 317, 346

Warnsdorfa, 404

Hilbert Dawid, 21

hill-climbing, *Patrz*: przeszukiwanie grafu metoda
podjeżdżania

Hoare Tony, 24

Hollerith Herman, 21

Huffman, 390

I

IBM, 21, 22

IFIP, 24

iloczyn logiczny, 38

indukcja matematyczna, 30

instrukcja

czas jednostkowy wykonania, *Patrz*: czas
 jednostkowy wykonania instrukcji

if... else, 432

switch, 432

int, 104

interfejs, 23, 216, 217

Deque, 221

java.util.List, 224

List, 221

użytkownika, 25, 124

inżynieria odwrotna, 374

iteracja, 51

iterator, 164, 165, 166, 167, 168, 216, 221, 222

deklarowanie, 222

J

Jacquard Joseph Marie, 20
Java, 25, 26, 421
 środowisko
 DKE, 423
 Eclipse, 424
 IDE, 423
 JRE, 422
 kompilowanie programów, 426
 konfiguracja, 425
 uruchomieniowe, 422
 tekst, 431
Java Garbage Collector, 118, 140, 144
język
 assembler, *Patrz:* assembler
 C, 23
 COBOL, 258, 263
 Fortran, 263
 Java, *Patrz:* Java
 Lisp, 27, 144, 400
 Prolog, 27, 144, 400
 strukturalny, 26
JVM, 105, 118, 140, 144, 422

K

Karp Richard, 286
klasa, 117, 118
 ArrayList, 160
 BigInteger, 380
 Boolean, 107
 Character, 107
 definicja, 118
 generyczna, 173
 instancja, *Patrz:* obiekt
 Integer, 107
 java.util.Arrays, 217
 java.util.HashSet, 225, 241
 java.util.LinkedList, 221, 222
 java.util.Stack, 223
 java.util.Vector, 218, 222, 224
 metoda statyczna, *Patrz:* metoda statyczna
 O, 86
 osłonowa, 107
 pochodna, 123
 pole statyczne, *Patrz:* pole statyczne
 Scanner, 435
 sekcja prywatna, 118
 String, 105, 107, 109, 115, 285
 dokumentacja, 110
klucz, 231, 232, 331, 373
 problem transmisji, 374
 prywatny, 375
 publiczny, 371, 375, 376

Knuth Donald, 286
kod
 ASCII, 44, 46, 283, 370, 371, 372, 385
 BCD, 40
 Huffmana, 371, 385, 387, 388, 389
 ISO 8859-2, 45
 ISO Latin-2, 45
 jednoznaczny, 386
 nadmiarowy, 371
 nierównomierny, 371
 optymalizacja, 100, 263
 równomierny, 371
 U2, 42
 Unicode, 46, 104, 283, 295
 UTF-16, 46
 wykonywalny, 25
 znak-moduł, 42
 źródłowy, 25
kodowanie, 369, 371, *Patrz też:* szyfrowanie,
 kompresja
 asymetryczne, 374
 podpis cyfrowy, 375
 przedrostek, 386
 słownik, 390, 391
 symetryczne, 373, 380
kolejka
 FIFO, 176, 177, 224, 348, 349
 implementacja, 177, 178
 LIFO, 171
 priorytetowa, 179, 182
kompilator, 25, 43, 264, 322, 426
komponent, 23
kompresja, 205, 369, 370, 382, 387,
 Patrz też: kodowanie, szyfrowanie
 bezzatna, 383
 danych, 206
 GIF, 371, 391, 393
 LZW, 371, 390, 391, 395
 redundancja, 383
 RLE, 384
 stopień, 383
 stratna, 383
 szybkość działania, 383
komputer, 22
 kwantowy, 19, 381, 382
konstruktor, 118
kopiec, *Patrz:* sarta
kółko i krzyżyk, 406, 407, 408
 generowanie listy możliwych ruchów, 412
 kodowanie listy węzłów potomnych, 412
 linie otwarte, 408
kwadrat parzysty, 67
Kwadraty.java, 68

L

las, 344
 Last In First Out, *Patrz:* kolejka LIFO
 LCS, 313
 Leibniz Gottfried Wilhelm, 20
 Lempel Abraham, 391
 Leonardo z Pizy, *Patrz:* Fibonacci
 liczba
 całkowita, 104
 bardzo duża, 376, 380
 mnożenie, 302
 dwójkowa
 arytmetyka, 37
 operacje logiczne, 38
 ze znakiem, 41
 losowa, 19
 ósemkowa, 41
 reprezentacja, 43
 szesnastkowa, 35, 41
 zespolona, 117, 119, 120
 zmiennoprzecinkowa, 104
 LIFO, *Patrz:* kolejka LIFO
 Lisp, 27, 144, 400
 lista, 130
 cykliczna, 163
 długość, 81
 dodawanie, 143
 dwukierunkowa, 160, 163
 element
 referencja, 147, 157
 usuwanie, 139, 155
 wstawianie, 135, 136, 147, 156, 160, 167
 wyszukiwanie, 138
 fuzja, 144
 implementacja, 154, 157
 jednokierunkowa, 131, 139, 192
 głowa, 132, 133, 139
 implementacja, 133
 ogon, 132, 133, 139
 reprezentacja tablicowa, 154
 wady i zalety, 146
 zapętłona, 164
 nieposortowana, 138
 posortowana, 136, 138, 147
 rankingowa, 351
 sąsiedztwa, 328, 330
 sortowanie, 146, 147
 tworzenie, 131
 z iteratorem, 164, 166, 167, 168
 literał, 104
 Lovelace Ada, 21
 Lucas Édouard, 267, 268
 LZW, *Patrz:* format LZW

Ł

łańcuch znakowy, *Patrz:* napis

M

McCarthy 91, *Patrz:* funkcja McCarthy'ego
 macierz, *Patrz też:* tablica
 kierowania ruchem, 336
 redukcja wsteczna, 365
 trójkątna, 365
 main, 431
 MARK 1, 22
 Markow Andriej, 21
 maszyna
 algorytmiczna, 20
 analityczna, 21
 Moore'a, 325
 różnicowa, 21
 tkacka Jacquarda, 20
 Turinga, 22, 28, 29, 397
 Mathcad, 357
 Mathematica, 357
 Matlab, 357
 Mauchly John William, 22
 McCarthy John, 24
 m-drzewo, *Patrz:* drzewo m-drzewo
 metadane, 47
 metoda, 117
 addAll, 220
 binarySearch, 218
 boolean, 435
 booleanhasNextInt, 435
 capacity, 221
 clear, 220
 clone, 220
 compare, 218
 contains, 220
 copyOf, 218
 empty, 223
 equals, 218, 220
 finalize, 118
 Floyda, *Patrz:* metoda niezmienników
 funkcji przeciwnych, 274, 280
 Gausa, 364, 365, 366
 indexOf, 222
 isEmpty, 220
 Lagrange'a, 360, 361
 Newtona, 358
 nextByte, 435
 nextDouble, 435
 nextInt, 435
 niezmienników, 30
 peek, 223
 pop, 223
 push, 223

- remove, 220
- równań charakterystycznych, 93, 95
- search, 223
- Simpsona, 363
- size, 221
- sort, 218
- statyczna, 122
- Stirlinga, 362, 363
- System.arraycopy, 218
- toString, 218, 221
- transformacji kluczowej, 85
- miara złożoności obliczeniowej, 80
- Mitnick Kevin, 381
- modelowanie matematyczne, 383
- Moore J Strother, 286
- Morris James, 286
- Morse Samuel, 382
- Muhammad ibn Musa al-Chuwarizmi, 17
- MYCIN, 400

N

- nadklasa, 123
- największy wspólny dzielnik, *Patrz:* NWD
- napis, 107, 108
 - jako tablica, 115
- negacja, 38
- niezmiennik, 30
- Nilsson Nils, 350
- notacja
 - dużego O, 86, 99
 - Landaua, 86
 - O, 85, 86, 87
 - $O(1)$, 85, 230
 - $O(2^n)$, 86
 - $O(\log n)$, 86
 - $O(\log N)$, 229
 - $O(n)$, 85
 - $O(n^2)$, 86
 - ONP, 201, 203
 - z kropką, 123, 124
- null, 114
- NWD, 17, 26, 72

O

- obiekt, 113, 117
 - adres, 115
 - lokalny, 140
 - tworzenie, 120
- obraz, 46
- obwód zamknięty, 335
- odpluskwanie, 24
- odwrotna notacja polska, 201, 203
- O-notacja, *Patrz:* notacja O

- ONP, *Patrz:* odwrotna notacja polska
- operacje logiczne, 106
- operand, 200
- operator, 200
 - %, 105
 - /, 105
 - bitowy, 107
 - dekrementacji, 105
 - inkrementacji, 105
 - logiczny, 106
- oprogramowanie wydajność, 81
- OR, 38

P

- PageRank, 322
- pakiet, *Patrz:* biblioteka
- pamięć, 245
 - cache, 100
- Pascal Blaise, 20
- pętla, 51
 - for, 51, 432
 - while, 51, 138, 433
- plik wykonywalny, 25
- podklasa, 123
- pole statyczne, 122
- Pratt Vaughan Ronald, 286
- prawdopodobieństwo występowania liter w języku polskim, 388
- problem
 - 8 hetmanów, 318
 - algorytmiczny, 29
 - chińskiego listonosza, 326
 - doboru, 351, 352, 353
 - Flawiusza, 164
 - komiwojażera, 326
 - konika szachowego, 403
 - NP-zupełny, 326
 - optymalnego doboru, 322
 - plecakowy, 304, 305
 - transmisji klucza, 374
 - wież Hanoi, 303
 - wież Hanoi, 267, 268, 273, 280
 - wydawania reszty, 307
 - wyrażania preferencji, 351
- procesor, 28
- program, *Patrz też:* algorytm
 - kontekst, 65
 - optymalizacja, 100
 - tworzenie, 25
- programowanie
 - dynamiczne, 58, 308, 309, 311, 312
 - etapy, 309
 - metodologia, 24

programowanie
 metody, 295
 dziel i zwyciężaj, *Patrz:* dziel i zwyciężaj
 heurystyczne, 317, *Patrz też:* algorytm
 heurystyczny, heurystyka
 obiektowe, 107, 116, 117, 130
 Prolog, 27, 144, 400
 przebieg, *Patrz:* iteracja
 przeciążanie, 123
 przesunięcie bitowe, 38
 przeszukiwanie, 227, *Patrz też:* algorytm
 przeszukiwania
 binarne, 70, 95, 228, 229, 303
 klasa algorytmu, 96
 grafu, 350
 ekspansja węzłów, 346
 metoda podejżdżania, 350
 strategia A*, 350
 w głąb, 346, 348
 wszecz, 346, 348
 z powracaniem, 350
 liniowe, 227
 rekurencyjne, 227
 tekstu, 283, 284, 286, 290
 przetwarzanie równoległe, 19
 push, 175

R

Rabin Michael, 286
 Raphael Bertram, 350
 RC, *Patrz:* równanie charakterystyczne
 redundancja, 383
 reguła Warnsdorfa, 404
 rekurencja, 51, 56, 63, 65, 69, 92, 126, 192, 195,
 196, 266, 295, 308, *Patrz też:* algorytm
 rekurencyjny
 definicja, 51
 derekursywacja, *Patrz:* derekursywacja
 naturalna, 64
 niebezpieczeństwa, 56, 59, 61, 69
 schematy, 276
 if-else, 277
 podwójne wywołanie, 279
 while, 276
 skrośna, 70
 terminalna, 266, 270
 z parametrem dodatkowym, 64, 65, 69, 270
 zasada działania, 52
 rekursja, *Patrz:* rekurencja
 relaksacja, 342
 reverse engineering, *Patrz:* inżynieria odwrotna
 Ritchie Dennis, 23
 Rivest Ron, 375
 RLE, 384
 routing matrix, 336

Roy Bernard, 339
 rozkład logarytmiczny, 95
 rozmiar danych, 81
 równanie
 charakterystyczne, 93, 94
 liniowe, 364, 366
 rekurencyjne, 89, 96
 dziedzina, 97
 rozwiązanie ogólne, 93, 94
 rozwiązanie szczególne, 93, 94
 z wieloma zmiennymi, 311, 312
 różnica symetryczna, 38
 RS, *Patrz:* równanie rekurencyjne rozwiązanie
 szczególne

S

schemat Hornera, 359, 377
 Shamir Adi, 375
 SI, *Patrz:* Sztuczna Inteligencja
 sieć neuronowa, 401, 402
 sito Erastotenesa, 415, 417
 słownik, 205, 390, 391
 węzłów, 328, 330, 331, 346
 sort, 224
 sortowanie, 245, *Patrz też:* algorytm sortowania,
 dane sortowanie
 bąbelkowe, 248, 249
 przez kopcowanie, 253, 254
 przez scalanie, 256, 257, 259, 262
 wydajność, 258, 260
 przez wstawianie, 246
 przez wytrząsanie, 250
 szybkie, 250, 251, 258, 259
 implementacja, 252
 wydajność, 253
 wewnętrzne, 245
 zewnętrzne, 245, 258, 259, 262
 dzielenie plików, 259
 wady, 260
 sortowanie szybkie, 303
 spirala, 66
 SRL, *Patrz:* szereg rekurencyjny liniowy
 Stack overflow, 58
 stała, 105
 null, *Patrz:* null
 znakowa, 104
 Static.java, 122
 sterta, 179, 180, 183, 210, 253
 element
 największy, 182
 wstawianie, 181, 182, 184
 implementacja, 183
 niezmiennik, 181
 sortowanie, 186
 tworzenie, 181

Serta.java, 183
stos, 56, 171, 203, 223, 263, 271, 272
 implementacja, 173
 przepełnienie, 58, 98, 173
 push, 175
 wywołań rekurencyjnych, 265
 zasada działania, 171, 172
Strassen Volker, 300
struktura danych, *Patrz:* dane struktura
suma logiczna, 38
suma
 modulo, 293
 modulo 2, 38, 233
 modulo $R_{\max}$, 234
superklasa, 123
system
 ciągłej integracji, 23
 ekspercki, 399, 400
 kodowania, 36
 dwójkowy, 35, 36
 dziesiętny, 36
 ósemkowy, 41
 szesnastkowy, 35, 41
 kontroli wersji, 23
 pozycyjny, *Patrz:* system kodowania
 UNIX, 23
 uzupełnienia dwójkowego, *Patrz:* kod U2
szablon, *Patrz:* wzorec projektowy
szereg rekurencyjny liniowy, 92
Sztuczna Inteligencja, 397, 398
 cele, 398
 graf stanu, 405
 modelowanie zadania, 402
 sztuczny lekarz, 400
szyfr
 blokowy, *Patrz:* algorytm DES
 łamanie, 381, 382
szyfrowanie, 369, 372, *Patrz też:* kodowanie
 klucz, *Patrz:* klucz

T

tablica, 110, 131, 154, 218, *Patrz też:* macierz
 dwuwymiarowa, 328
 element, 297, 298, 299
 mnożenie, 300, 301
 równoległa, 192
 różnic centralnych, 362
 scalanie, 256
 sumowanie, 218
 wielowymiarowa, 112
 zamiana na listę, 154
 zerowanie fragmentu, 87
techniki programowania, *Patrz:* programowanie
 metody
tekst źródłowy, 25

teoria
 automatów, 325
 gier, 346, 397
 gra dwuosobowa, 405
 grafów, *Patrz:* graf
test Turinga, 29, 398
Thompson Ken, 23
tictac.cpp, 413
token, 435
transformacja kluczowa, *Patrz:* hashing
traveling salesman proble, *Patrz:* problem
 komiwojażera
Turing Alan, 21, 28, 29, 398
Turinga maszyna, *Patrz:* maszyna Turinga
twierdzenie Eulera, 327
typ, 103
 abstrakcyjny, 129, 130
 BigDecimal, 380
 BigInteger, 380
 boolean, 107, *Patrz:* typ logiczny
 byte, 104
 char, 104, 107
 deklarowanie, 105
 double, 104
 float, 104
 int, 104, 107
 logiczny, 104
 long, 104
 prosty, 104
 short, 104
 struct, 112
 wskaźnikowy, 113
 złożony, 113

U

uczenie maszynowe, 402
układ równań liniowych, 364, 366
UNIVAC 1, 22
uniwersalna struktura słownikowa, *Patrz:* USS
USS, 205, 207

V

von Neumann Johannes, 22

W

Warshall Stephen, 339
warstwa, 23
Welch Terry, 391
wielomian, 376, 377
 interpolacyjny, 361
wielowątkowość, 100
Wirth Niklaus, 24
wydajność oprogramowania, 81

wyjątek, 136
wyrażenie arytmetyczne, 199, 200, 201, 203
wywołanie terminalne, 266
wyznacznik Vandermonde’a, 361
wzorzec projektowy, 23
wzór
 Simpsona, 364
 Stirlinga, 362, 363

X

XOR, 38

Z

zasada
 optymalności, 308
 Pareta, 100
zbiór, 211, 328
Ziv Jacob, 391
zmienna, 103
 globalna, 271
 inicjacja, 30
 logiczna, 104
 lokalna, 271
 eliminacja, 272, 274, 275
 referencyjna, 112, 113, 120
 wskaźnikowa, 105

znak, 104
 %, 105
 ++, 105
 apostrofu, 104
 ciąg, *Patrz:* ciąg znaków
 cudzysłowu, 104
 LF, 371
 łańcuch, *Patrz:* napis
 równości, 105
 ukośnik, 105

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>

GRUPA
Helion

Opanuj Javę jak prawdziwy profesjonalista!

- Podstawy algorytmiki dla praktyków
- Modelowanie struktur danych w Javie
- Skuteczne techniki programowania na przykładach

Java jest obecnie jednym z najpopularniejszych języków programowania, co zawdzięcza przede wszystkim swojej prostocie, nowoczesności, dużym możliwościom oraz niezależności od architektury platform sprzętowych i systemowych, na których mają pracować napisane w tym języku programy. Java znalazła zastosowanie w wielu różnych branżach — zdecydowanie dominuje w rozwiązaniach działających w sieci, stanowiących obecnie dużą część oprogramowania tworzonego komercyjnie. Mimo to dotychczas trudno było znaleźć rzetelne źródło wiedzy o algorytmach, które byłoby przeznaczone dla użytkowników Javy, wyjaśniało zasady modelowania danych w tym języku i pozwalało szybko testować gotowe programy.

Na szczęście to już przeszłość! Książka *Algorytmy, struktury danych i techniki programowania dla programistów Java* jest pierwszą poważną pozycją przybliżającą tematykę algorytmów osobom posługującym się tym językiem. W prosty i praktyczny sposób przedstawia najważniejsze zagadnienia algorytmiki, pozwala poznać struktury danych i ich zastosowania, prezentuje popularne algorytmy oraz problemy, które można za ich pomocą rozwiązać, omawia także techniki programowania wykorzystywane przez miliony specjalistów w ich codziennej pracy. Jeśli chcesz być profesjonalnym programistą Javy, nie mogłeś trafić lepiej!

- Podstawy algorytmiki i kodowania liczb
- Algorytmy rekurencyjne i iteracyjne
- Analiza złożoności i optymalizacja algorytmów
- Modelowanie i zastosowanie struktur danych
- Wykorzystanie biblioteki java.util
- Przeszukiwanie i sortowanie danych
- Przegląd technik programowania
- Algorytmy grafowe i numeryczne
- Kodowanie i kompresja danych
- Wprowadzenie do języka Java i narzędzi JDK

Rozwiązuje problemy programistyczne w Javie!

Helion 



helion.pl



HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!



AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-5465-4



9 788328 354654

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 67,00 zł