



Piotr Wróblewski

Algorytmy, struktury danych i techniki programowania

Wydanie VI

Helion 

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/algor6>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5374-9

Copyright © Helion 2019

Printed in Poland.

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

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

Spis treści

Przedmowa	9
Uwagi do wydania VI	9
Co odróżnia tę książkę od innych podręczników?	10
Dlaczego C++?	11
Jak należy czytać tę książkę?	11
Co zostało opisane w tej książce?	12
Programy przykładowe	14
Konwencje typograficzne i oznaczenia	15
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. Rekurencja	33
Definicja rekurencji	33
Ilustracja pojęcia rekurencji	35
Jak wykonują się programy rekurencyjne?	36
Niebezpieczeństwa rekurencji	38
Ciąg Fibonacciego	38
Stack overflow!	40

Pułapek ciąg dalszy	42
Stąd do wieczności	43
Definicja poprawna, ale...	43
Typy programów rekurencyjnych	45
Myślenie rekurencyjne	46
Przykład 1. Spirala	47
Przykład 2. Kwadraty „parzyste”	48
Uwagi praktyczne na temat technik rekurencyjnych	50
Zadania	51
Rozwiązania i wskazówki do zadań	53
Rozdział 3. Systemy obliczeniowe i podstawy kodowania	59
System dziesiętny i kilka definicji	60
System dwójkowy	60
Operacje arytmetyczne na liczbach dwójkowych	61
Operacje logiczne na liczbach dwójkowych	62
Kod BCD	64
System ósemkowy	65
System szesnastkowy	65
Kodowanie liczb ze znakiem	65
Kod znak-moduł (ZM)	66
Kod U2 (system uzupełnienia dwójkowego)	66
Zmienne w pamięci komputera	67
Kodowanie znaków	68
Kodowanie obrazów	70
Mapy bitowe na przykładzie formatu BMP	71
Rozdział 4. Typy i struktury danych	75
Typy podstawowe i złożone	76
Tablice	77
Ciągi znaków i napisy w C++	78
Typy złożone	80
Struktury i wprowadzenie pojęcia referencji	80
Klasy i programowanie obiektowe	83
Abstrakcyjne struktury danych	83
Listy jednokierunkowe	85
Tablicowa implementacja list	106
Stos	111
Kolejki FIFO	116
Stery i kolejki priorytetowe	119
Drzewa i ich reprezentacje	125
Zbiory	138
STL, czyli struktury danych dla leniuchów	140
Klasyczne kontenery sekwencyjne	141
Adaptory (nakładki na inne kontenery)	147
Kontenery asocjacyjne	148
Algorytmy w STL	151
Dalsze materiały na temat STL	152
Zadania	152
Rozwiązania zadań	153

Rozdział 5. Analiza złożoności algorytmów	155
Definicje i przykłady	156
Jeszcze raz funkcja silnia	160
Zerowanie fragmentu tablicy	163
Wpadamy w pułapkę	165
Różne typy złożoności obliczeniowej	166
Nowe zadanie: uprościć obliczenia!	168
Analiza programów rekurencyjnych	169
Terminologia i definicje	169
Ilustracja metody na przykładzie	170
Rozkład logarytmiczny	171
Przeszukiwanie binarne... tym razem bez matematyki wyższej!	173
Zamiana dziedziny równania rekurencyjnego	174
Funkcja Ackermanna, czyli coś dla smakoszy	174
Złożoność obliczeniowa to nie religia!	176
Techniki optymalizacji programów	176
Zadania	177
Rozwiązania i wskazówki do zadań	178
Rozdział 6. Derekursywacja i optymalizacja algorytmów	181
Jak pracuje kompilator?	182
Odrobina formalizmu nie zaszkodzi!	184
Kilka przykładów derekursywacji algorytmów	185
Derekursywacja z wykorzystaniem stosu	188
Eliminacja zmiennych lokalnych	188
Metoda funkcji przeciwnych	190
Klasyczne schematy derekursywacji	192
Schemat typu while	193
Schemat typu if-else	194
Schemat z podwójnym wywołaniem rekurencyjnym	196
Podsumowanie	198
Rozdział 7. Algorytmy sortowania	199
Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	200
Sortowanie bąbelkowe, algorytm klasy $O(N^2)$	201
Sortowanie szybkie (Quicksort) — algorytm klasy $O(N \log N)$	203
Heapsort — sortowanie przez kopcowanie	206
Scalanie zbiorów posortowanych	209
Sortowanie przez scalanie, algorytm klasy $O(N \log N)$	209
Sortowanie zewnętrzne	211
Uwagi praktyczne	214
Rozdział 8. Algorytmy przeszukiwania	217
Przeszukiwanie liniowe	217
Przeszukiwanie binarne	218
Transformacja kluczowa (hashing)	220
W poszukiwaniu funkcji H	221
Najbardziej znane funkcje H	222
Obsługa konfliktów dostępu	224
Powrót do źródeł	225
Jeszcze raz tablice!	226
Próbkowanie liniowe	226

Podwójne kluczowanie	228
Zastosowania transformacji kluczowej	229
Podsumowanie metod transformacji kluczowej	230
Rozdział 9. Przeszukiwanie tekstów	233
Algorytm typu brute force	233
Nowe algorytmy poszukiwań	235
Algorytm KMP	236
Algorytm Boyera-Moore'a	240
Algorytm Rabina-Karpa	242
Rozdział 10. Zaawansowane techniki programowania	245
Programowanie typu „dziel i zwyciężaj”	246
Odszukiwanie minimum i maksimum w tablicy liczb	247
Mnożenie macierzy o rozmiarze $N \times N$	249
Mnożenie liczb całkowitych	252
Inne znane algorytmy „dziel i zwyciężaj”	253
Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas...	253
Problem plecakowy, czyli niełatwe jest życie turysty piechura	254
Wydawanie reszty, czyli „A nie ma pan drobnych?” w praktyce	257
Programowanie dynamiczne	258
Ciąg Fibonacciego	259
Równania z wieloma zmiennymi	260
Najdłuższa wspólna podsekwencja	261
Inne techniki programowania	264
Uwagi bibliograficzne	266
Rozdział 11. Elementy algorytmiki grafów	269
Definicje i pojęcia podstawowe	270
Etykiety i wartości	271
Cykle w grafach	273
Sposoby reprezentacji grafów	276
Reprezentacja tablicowa	276
Słowniki węzłów	278
Listy kontra zbiory	279
Podstawowe operacje na grafach	279
Suma grafów	279
Kompozycja grafów	280
Graf do potęgi	280
Algorytm Roya-Warshalla	281
Algorytm Floyda-Warshalla	284
Algorytm Dijkstry	287
Algorytm Bellmana-Forda	289
Drzewo rozpinające minimalne	289
Algorytm Kruskala	290
Algorytm Prima	291
Przeszukiwanie grafów	291
Strategia „w głąb” (przeszukiwanie zstępujące)	292
Strategia „wszerz”	294
Inne strategie przeszukiwania	295
Problem właściwego doboru	296
Podsumowanie	300
Zadania	300

Rozdział 12. Algorytmy numeryczne	301
Poszukiwanie miejsc zerowych funkcji	301
Iteracyjne obliczanie wartości funkcji	303
Interpolacja funkcji metodą Lagrange'a	304
Różniczkowanie funkcji	305
Całkowanie funkcji metodą Simpsona	307
Rozwiązywanie układów równań liniowych metodą Gaussa	308
Biblioteka GSL (GNU Scientific Library)	311
Uwagi końcowe	311
Rozdział 13. Czy komputery mogą myśleć?	313
Przegląd obszarów zainteresowań sztucznej inteligencji (SI)	314
Systemy eksperckie	315
Sieci neuronowe	317
Reprezentacja problemów	318
Gry dwuosobowe i drzewa gier	320
Algorytm min-max	321
Rozdział 14. Kodowanie i kompresja danych	327
Kodowanie danych i arytmetyka dużych liczb	329
Metody prymitywne	329
Kodowanie symetryczne	331
Kodowanie asymetryczne	332
Łamanie kodów	338
Jakość klucza szyfrującego	338
Metody łamania szyfrów	339
Techniki kompresji danych	340
Kompresja za pomocą modelowania matematycznego	341
Kompresja metodą RLE	342
Kompresja danych metodą Huffmana	343
Kodowanie LZW	348
Rozdział 15. Zadania różne	355
Teksty zadań	355
Rozwiązania	357
Dodatek A Poznaj C++ w pięć minut!	361
Elementy języka C++ na przykładach	361
Pierwszy program	361
Dyrektywa #include	362
Kod warunkowy w C++	362
Operacje arytmetyczne i zmienne	363
Operacje logiczne	363
Wskaźniki i zmienne dynamiczne	364
Referencje	365
Typy proste i typy złożone	365
Podprogramy	367
Procedury	367
Funkcje	367
Instrukcja wyboru (switch)	368
Iteracje	369
Struktury rekurencyjne	369
Parametry programu main()	370

Operacje na plikach w C++	370
Programowanie obiektowe w C++	371
Terminologia	372
Obiekty na przykładzie	373
Składowe statyczne klas	376
Metody stałe klas	376
Dziedziczenie własności	376
Dodatek B	
Kompilowanie programów przykładowych	381
Zawartość archiwum ZIP na FTP-ie	381
Darmowe kompilatory C++	382
GCC (GNU Compiler Collection)	382
Microsoft Visual Studio Community	384
macOS	386
Dev-C++ (Orwell)	386
Kompilacja i uruchamianie programów w C++	387
GCC	387
Microsoft Visual Studio	388
Dev-C++	395
Cygwin	395
Literatura	397
Spis ilustracji	399
Spis tabel	404
Skorowidz	406

Rozdział 9.

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) jako jednostki leksykalne mające określone znaczenie dla czytelnika. Wszystko jest zresztą kwestią umowy, w szczególności ciąg bitów może reprezentować... pamięć ekranu.

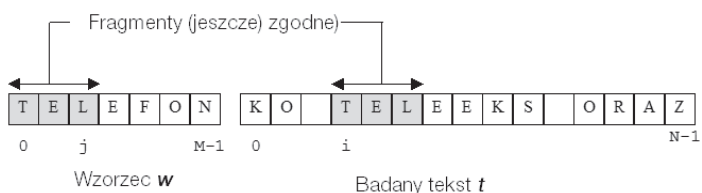
Okazuje się wszelako, że przyjęcie konwencji dotyczących interpretowania informacji ułatwia wiele operacji na niej. Dlatego też pozostajmy przy ogólnikowym stwierdzeniu „tekst”, wiedząc, że za tym określeniem może się kryć sporo znaczeń.

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 „bezmądry” 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 9.1.

Rysunek 9.1.
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 wzorzec startuje w badanym tekście: będzie to oczywiście *i*-*M*.

Tłumacząc powyższe sytuacje na C++, możemy łatwo dojść do następującej procedury¹:



szukaj-txt.cpp

```
#include <iostream>
#include <string.h> //z uwagi na strlen()
using namespace std;

int szukaj(char w[], char t[]){
    int i=0,j=0, M=strlen(w), N=strlen(t);
    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
}
```

Funkcję `szukaj` wywołujemy, wkładając do jej listy argumentów zmienne lub bezpośrednie wartości, np. `int wynik=szukaj(w, t)` lub `wynik=szukaj("wzorzec", t)`.

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorzec, lub -1, gdy poszukiwany tekst nie został odnaleziony — to znana już doskonale konwencja.

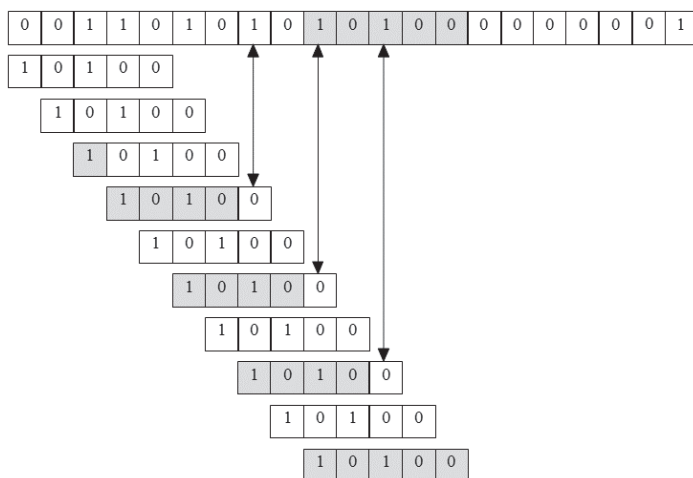
Przypatrzmy się teraz dokładniej przykładowi poszukiwania wzorca 10100 w pewnym tekście binarnym (rysunek 9.2).

Rysunek jest nieco uproszczony: w istocie poziome przesuwanie się wzorca oznacza instrukcje zaznaczone na listingu `szukaj-txt.cpp` 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 wzorzec złożone z samych zer i zakończone jedyneką (umawiamy się, że zera i jedyńki symbolizują tu dwa różne znaki).

¹ W algorytmie możesz ewentualnie użyć obiektów klasy `string` zamiast tablic znakowych, ale ponieważ w tej klasie znajdują się już gotowe metody pozwalające wyszukiwać wzorce (`find`, `rfind`), realizacja algorytmu wyszukiwania byłaby nadmiarowa.

Rysunek 9.2.
*„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.² $M \cdot N$.

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.

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

² Zwykle M będzie znacznie mniejsze niż N .

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 następujących trzech punktach szczegółowo omówię algorytmy wspomniane w tym historycznym przeglądzie.

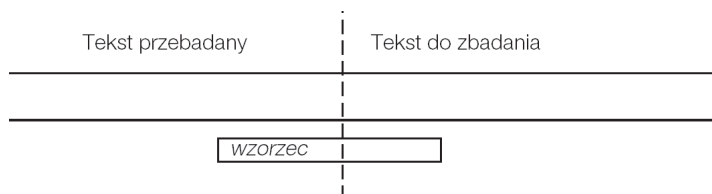
Algorytm KMP

Wadą algorytmu *brute force* jest jego czułość 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 pojawia³, to w razie restartu nie musimy cofać wskaźnika i o $j-1$ pozycji, jak to było poprzednio (patrz listing *szukaj-txt.cpp*). 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 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 wzorca⁴ 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. Popatrzmy w tym celu na rysunek 9.3.

Rysunek 9.3.
Wyszukiwanie optymalnego przesunięcia w algorytmie KMP



³ Przykład: „ABBBBBBB” — znak „A” wystąpił tylko jeden raz.

⁴ Lub i w przypadku badanego tekstu.

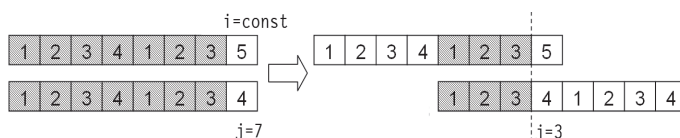
Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz wzorzec bardzo wolno w prawo, patrząc jednocześnie na już zbadany tekst — tak aby obserwować ewentualne pokrycie się tej części wzorca, która znajduje się po lewej stronie 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 wzorzec, 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, popatrzmy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 9.4 możemy dostrzec, że na siódmej pozycji wzorca⁵ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

Rysunek 9.4.

Przesuwanie się wzorca w algorytmie KMP (1)

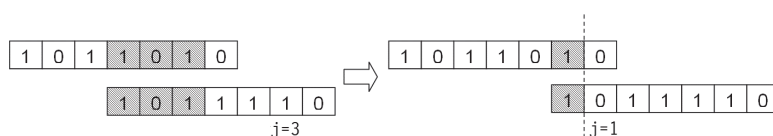


Jeśli zostawimy indeks i w spokoju, to modyfikując wyłącznie j , możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? Przesuwając go wolno w prawo (rysunek 9.4), doprowadzamy w pewnym momencie do nałożenia się ciągów 123 przed kreską — cała strefa niezgodności została wyprowadzona na prawo i ewentualne dalsze testowanie może być kontynuowane!

Analogiczny przykład znajduje się na rysunku 9.5.

Rysunek 9.5.

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

⁵ Licząc indeksy tablicy tradycyjnie od zera.



Listing

kmp.cpp (fragment)

```
#include <iostream>
#include <string.h> //z uwagi na strlen()
using namespace std;

int kmp(char w[], char t[]){
    int i,j, N=strlen(t);
    for(i=0,j=0; (i<N) && (j<M); i++,j++){
        while( (j>=0) && (t[i]!=w[j]) )
            j=shift[j];
        if (j==M)
            return i-M;
        else
            return -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 `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 `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!

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

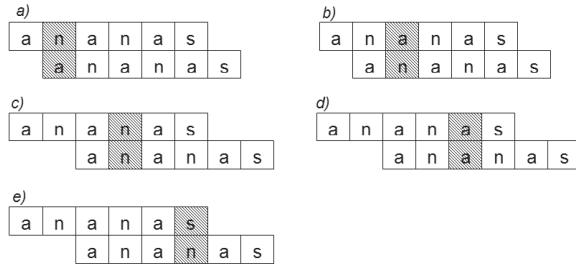
Sens tego algorytmu jest następujący: tuż po inkrementacji i oraz j wiemy, że pierwsze j znaków wzorca jest zgodne ze znakami na pozycjach $p[i-j-1] \dots p[i-1]$ (ostatnie j pozycji w pierwszych i znakach wzorca). Ponieważ jest to największe j spełniające powyższy warunek, zatem aby nie ominąć *potencjalnego*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

Popatrzmy, jaki będzie efekt zadziałania funkcji `init_shifts` na słowie *ananas* (rysunek 9.6). Zacięniowane litery oznaczają miejsca, w których wystąpiła niezgodność wzorca z tekstem. W każdym przypadku graficznie przedstawiono efekt przesunięcia wzorca — widać wyraźnie, które strefy pokrywają się przed strefą zacięniowaną (porównaj z rysunkiem 9.5).

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

Rysunek 9.6.

Optymalne przesunięcia
wzorca „ananas”
w algorytmie KMP



Algorytm KMP można zoptymalizować, jeśli znamy z góry wzorce, których będziemy poszukiwać. Jeśli przykładowo bardzo często zdarza nam się szukać w tekstach słowa *ananas*, w funkcję `kmp` można wbudować tablicę przesunięć:

**ananas.cpp**

```
#include <iostream>
using namespace std;
int kmp_ananas(char t[]){
    int i=-1;
    start: i++;
    et0: if (t[i]!='a')
        goto start;
        i++;
    et1: if (t[i]!='n')
        goto et0;
        i++;
    et2: if (t[i]!='a')
        goto et0;
        i++;
    et3: if (t[i]!='n')
        goto et1; i++;
        if (t[i]!='a')
        goto et2; i++;
        if (t[i]!='s')
        goto et3; i++;
    return i-6;
}

int main() {
    char t[]="w sklepie można było kupić ananasy i pomarańcze";
    cout << t << endl;
    cout << "Wynik poszukiwań słowa 'ananas'=" << kmp_ananas(t) << endl;
    return 0;
}
```

W celu właściwego odtworzenia etykiet należy oczywiście co najmniej raz wykonać funkcję `init_shifts` lub obliczyć samemu odpowiednie wartości. W każdym razie gra jest warta świeczki: powyższa funkcja charakteryzuje się bardzo zwięzłym kodem wynikowym assemblerowym, jest zatem bardzo szybka. Posiadacze kompilatorów, które umożliwiają generację kodu wynikowego jako tzw. „assembly output”⁶, mogą z łatwością sprawdzić różnicę

⁶ Aby wygenerować kod assemblera w kompilatorze GNU C++, użyj w linii poleceń komendy `c++ plik.cpp -S`.

pomiędzy wersjami `kmp` i `kmp_ananas`! Dla przykładu mogę podać, że w przypadku wspomnianego kompilatora GNU klasyczna wersja procedury `kmp` (wraz z `init_shifts`) miała objętość ok. 310 linii kodu asemblerowego, natomiast `kmp_ananas` zmieściła się w ok. 100 liniach (przykładowe pliki z rozszerzeniem `s` utworzone w kompilatorze GNU C++ znajdziesz w archiwum ZIP w katalogu *Dodatki*).

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.

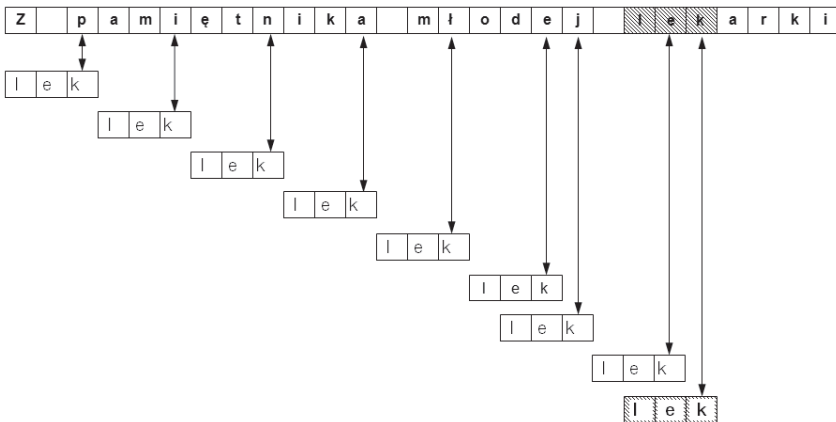
Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik `i` w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez cofania się w nim. W niektórych systemach może to mieć istotne znaczenie praktyczne — przykładowo mamy zamiar analizować bardzo długi plik tekstowy i charakter wykonywanych operacji nie pozwala na cofnięcie się w tej czynności (`i` w odczytywanym na bieżąco pliku).

Algorytm Boyera-Moore'a

Kolejny algorytm, który omówię, jest ideowo znacznie prostszy do zrozumienia niż algorytm KMP. W przeciwieństwie do metody KMP porównywaniu ulega ostatni znak wzorca. To niekonwencjonalne podejście ma kilka istotnych zalet:

- ◆ Jeśli podczas porównywania okaże się, że rozpatrywany aktualnie znak nie wchodzi w ogóle w skład wzorca, możemy „skoczyć” w analizie tekstu o całą długość wzorca do przodu! Ciężar algorytmu przesunął się zatem z analizy ewentualnych zgodności na badanie niezgodności, a te ostatnie są statystycznie znacznie częściej spotykane.
- ◆ Skoki wzorca są zazwyczaj znacznie większe od 1 — porównaj z metodą KMP!

Zanim przejdę do szczegółowej prezentacji kodu, omówię na przykładzie jego działanie. Spójrzmy w tym celu na rysunek 9.7, gdzie przedstawione zostało poszukiwanie ciągu znaków „lek” w tekście „Z pamiętnika młodej lekarki”⁷.



Rysunek 9.7. Przeszukiwanie tekstu metodą Boyera-Moore'a

⁷ Tytuł kultowego cyklu skeczy radiowych autorstwa Ewy Szumańskiej (1921 – 2011).

Pierwsze pięć porównań trafia na litery p, i, n, a i ł, które we wzorcu nie występują! Za każdym razem możemy zatem przeskoczyć w tekście o trzy znaki do przodu (długość wzorca). Porównanie szóste trafia jednak na literę e, która w słowie „lek” występuje. Algorytm wówczas przesuwając wzorec o tyle pozycji do przodu, aby litery e nałożyły się na siebie, i porównywanie jest kontynuowane.

Następnie okazuje się, że litera j nie występuje we wzorcu — mamy zatem prawo przesunąć się o kolejne trzy znaki do przodu. W tym momencie trafiamy już na poszukiwane słowo, co następuje po jednokrotnym przesunięciu wzorca, tak aby pokryły się litery k.

Algorytm jest jak widać klarowny, prosty i szybki. Jego realizacja także nie jest zbyt skomplikowana. Podobnie jak w metodzie poprzedniej, także tu musimy wykonać pewną prekompilację w celu utworzenia tablicy przesunięć. Tym razem jednak tablica ta będzie miała tyle pozycji, ile jest znaków w alfabecie — wszystkie znaki, które mogą wystąpić w tekście, plus spacja.

Będziemy również potrzebowali prostej funkcji indeks, która zwraca w przypadku spacji liczbę zero (w pozostałych przypadkach numer litery w alfabecie). Poniższy przykład uwzględni jedynie kilka polskich liter — z łatwością można go uzupełnić o brakujące znaki. Numer litery jest oczywiście zupełnie arbitralny i zależy od programisty. Ważne jest tylko, aby nie pominąć w tablicy żadnej litery, która może wystąpić w tekście. Jedną z możliwych wersji funkcji indeks przedstawiono poniżej:



bm.cpp (fragment)

```
#include <iostream>
#include <string.h> //z powodu strlen()
using namespace std;
int kmp_anas(char t[]){

const int K=26*2+2*2+1; // znaki ASCII + polskie litery + odstęp
int shift[K];
int indeks(char c){
switch(c){
case ' ':return 0; // odstęp = 0
case 'ę':return 53;
case 'Ę':return 54; // polskie litery
case 'ł':return 55;
case 'Ł':return 56; // itd. dla pozostałych polskich liter
default:
if(islower(c)) // czy c jest małą literą alfabetu
return c-'a'+1;
else
return c-'A'+27;
}
}
```

Funkcja indeks ma jedynie charakter usługowy. Służy ona m.in. do właściwej inicjacji tablicy przesunięć. Mając za sobą analizę przykładu z rysunku 9.7, nie powinien być zbyt zdziwiony sposobem inicjacji:

```
void init_shifts(char w[]){
int M=strlen(w);
for(int i=0; i<K; i++)
shift[i]=M;
```

```

for(int i=0; i<M; i++)
    shift[indeks(w[i])] = M-i-1;
}

```

Teraz przejdziemy do prezentacji listingu algorytmu z przykładem wywołania:

```

int bm (char w[], char t[]){
    init_shifts(w);
    int i, j, N=strlen(t), M=strlen(w);
    for(i=M-1, j=M-1; j>0; i--, j--){
        while(t[i] != w[j]){
            int x=shift[indeks(t[i])];
            if(M-j>x)
                i+=M-j;
            else
                i+=x;
            if (i>=N)
                return -1;
            j=M-1;
        }
        return i;
    }
}

int main(){
    char t[]="Z pamiętnika młodej lekarki";
    cout << "Wynik poszukiwań=" << bm("lek", t) << endl;
    return 0;
}

```

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 przesunięć⁸ 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 8. 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 znaków⁹. 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 pokręcić w tym miejscu z powątpiewaniem głową i stwierdzić, że przecież to nie ma prawa działać szybko! Istotnie pomysł wy-

⁸ Rozważ np. wielokrotne występowanie takich samych liter we wzorcu.

⁹ Na samym początku będzie to oczywiście M pierwszych znaków tekstu.

liczenia 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 słowowy typu *brute force*). Tym bardziej, że do tej pory nie powiedziałem ani słowa na temat funkcji H ! Z poprzedniego rozdziału pamiętasz zapewne, że jej wybór wcale nie był 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 zobaczymy, jak zmieni się wartość x :

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

Jeśli dobrze przyjrzyysz 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:

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

Jako funkcji H użyjemy dobrze znanej z poprzedniego rozdziału $H(x) = x \% p$, gdzie p jest dużą liczbą pierwszą. Założmy, 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:

$$(5 \cdot 100 + 6 \cdot 10 + 8) \% 7 = 568 \% 7 = 1.$$

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

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

co jest też łatwe do weryfikacji.

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

**rk.cpp**

```

#include <iostream>
#include <string.h>
using namespace std;

const long p=33554393; // duża liczba pierwsza
const int b=64;       // duże + małe znaki + „coś jeszcze”

int rk(char w[], char t[]){
  unsigned long i,bM_1=1, Hw=0, Ht=0, M=strlen(w), N=strlen(t);
  for(i=0; i<M; i++){
    Hw=(Hw*b+indeks(w[i]))%p; // inicjacja funkcji H dla wzorca
    Ht=(Ht*b+indeks(t[i]))%p; // inicjacja funkcji H dla tekstu
  }
  for(i=1; i<M; i++)
    bM_1=(b*bM_1)%p;

  for(i=0; Hw!=Ht; i++){ // przesuwanie się w tekście
    Ht=(Ht+b*p-indeks(t[i])*bM_1)%p; // *
    Ht=(Ht*b+indeks(t[i+M]))%p;
    if (i>N-M)
      return -1; // porażka poszukiwań
  }
  return i;
}

```

Na pierwszym etapie następuje wyliczenie początkowych wartości H_t i H_w . Ponieważ ciągi znaków trzeba interpretować jako liczby, konieczne będzie zastosowanie znanej już doskonale funkcji `indeks`. Wartość H_w jest niezmienna i nie wymaga uaktualniania. 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`, co jest wykonywane w trzeciej pętli `for`. Dodatkowego wyjaśnienia wymaga być może linia listingu `rk.cpp` oznaczona (*). Otóż dodawanie wartości $b*p$ do H_t pozwala uniknąć przypadkowego wskoczenia w liczby ujemne. Gdyby istotnie tak się stało, przeniesiona do następnego wyrażenia arytmetycznego wartość `modulo` byłaby nieprawidłowa i sfałszowałaby końcowy wynik!

Kolejne uwagi należą się parametrom p i b . Zaleca się, aby p było dużą liczbą pierwszą¹⁰, 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 ostrożny programista powinien wykonać dodatkowy test zgodności w `t[i]... t[i+M-1]` po zwróceniu przez funkcję `rk` pewnego indeksu i .

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

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

¹⁰ W naszym przypadku jest to liczba 33 554 393.

Skorowidz

255.cpp, 330
3DES, 332

A

- A*, 321
- A* (strategia), 295
- A.cpp, 174
- abstrakcyjne struktury danych, 83
- acykliczny graf skierowany, 273
- Adleman L., 333
- Aiken H., 22
- alfabet Braille'a, 340
- alfabet Morse'a, 340
- algebra Boole'a, 60, 61
- algorytm, 17
 - A*, 321
 - Bellmana-Forda, 289
 - Boyera i Moore'a, 240
 - cechy, 18
 - cięcie α - β , 321, 322
 - DES, 332
 - deterministyczny, 19
 - Dijkstry, 287
 - Fleury'ego, 275
 - Floyda, 286
 - Floyda-Warshalla, 284
 - generowanie automatyczne, 30
 - genetyczny, 266
 - heurystyczny, 265
 - operatory, 265
 - stany, 265
 - Huffmana, 343, 345
 - język prezentacji, 11
 - Kruskala, 290
 - kryteria wyboru, 155
 - LZW, 348, 349
 - mini-max, 321, 325
 - niedeterministyczny, 19
 - niestabilny, 50
 - opis słowny, 26
 - poprawność, 29, 30
 - poziom abstrakcji opisu, 26
 - prefiksowy, 343
 - Prima, 290, 291
 - Quicksort, 205
 - Rabina i Karpa, 242
 - Roy-Warshalla, 281
 - sposób prezentacji, 26
 - SSS*, 321
 - Strassena, 251
- algorytmika, 12, 20
- algorytmika grafów, 269
- algorytmy numeryczne, 301
 - całkowanie funkcji metodą Simpsona, 307
 - interpolacja funkcji metodą Lagrange'a, 304
 - iteracyjne obliczanie wartości funkcji, 303
 - metoda Gaussa, 308
 - metoda Stirlinga, 305
 - poszukiwanie miejsc zerowych funkcji, 301
 - rozwiązywanie układów równań liniowych, 308
 - różniczkowanie funkcji, 305
- algorytmy przeszukiwania, 217
 - hashing, 220
 - przeszukiwanie binarne, 218
 - przeszukiwanie liniowe, 217
 - transformacja kluczowa, 220
- algorytmy sortowania, 199
 - duże pliki wejściowe, 212
 - Heap sort, 206
 - klasa $O(N^2)$, 200, 201
 - kryteria wyboru algorytmu, 214
 - Quicksort, 203
 - Shaker-sort, 203
 - sortowanie bąbelkowe, 201
 - sortowanie przez kopcowanie, 206
 - sortowanie przez scalanie, 209

- algorytmy sortowania
 - sortowanie przez wstawianie, 200
 - sortowanie przez wytrząsanie, 203
 - sortowanie zewnętrzne, 211
- algorytmy żarłoczne, 253
 - problem plecakowy, 254
 - schemat algorytmu, 253
- analiza
 - bezpieczeństwa sieci, 270
 - obwodów elektronicznych, 270
 - wyrażenia beznawiasowego, 130
 - złożoności algorytmów, 155
- ananas.cpp, 239
- AND, 62
- argc, 370
- argv[0], 370
- array (klasa STL), 141
- arytmetyka dużych liczb, 329, 334
- ASCII, 68, 328, 343
- assembler, 26
- assembly output, 239
- atoi, 213
- automat skończony, 272
- automatyczne generowanie algorytmów, 30

B

- bajt, 61
- baza wiedzy, 315, 316
- BCD (kod), 64
- Bellman R. E., 258, 289
- BFS, 291
- binary-i.cpp, 218
- binary-s.cpp, 54
- binarys_stl.cpp, 219
- bit-operations.cpp, 63
- blok kodu, 361
- bm.cpp, 241
- BMP, 71
- bool, 76
- Boyer R. S., 235
- breadthf.cpp, 295
- breadth-first search, 291
- brute-force, 339
- bubble.cpp, 202
- BULL Gamma3, 22

C

- C++, 11, 26
 - blok, 361
 - deklaracje nagłówkowe, 368
 - dziedziczenie własności, 376
 - funkcje, 367
 - include (dyrektywa), 362
 - iteracje, 369

- kod warunkowy, 362
- komentarze, 362
- operacje
 - arytmetyczne, 363
 - logiczne, 363
 - na plikach, 370
- parametry programu main(), 370
- pętle, 369
- pliki dołączane, 362
- podprogramy, 367
- procedury, 367
- programowanie obiektowe, 371
- protected, 377
- przeciążanie, 378
- przedefiniowanie operatorów, 374
- referencje, 365
- składowe statyczne klas, 376
- struktury, 366
- struktury rekurencyjne, 369
- switch, 368
- tablice, 366, 367
- tekst, 362
- void, 367
- wskazniki, 364
- zmiennie dynamiczne, 364

- cache, 177
- calloc(), 86
- całkowanie funkcji, 307
 - metoda Simpsona, 307
- Carnot L. M., 20
- char, 76
- ciąg Fibonnaciego, 38, 178, 259
 - programowanie dynamiczne, 259
- ciągi znaków, 78
 - reprezentacja, 67
- cięcie α - β , 321, 322
- COBOL, 181
- Complex, 373
- complex.cpp, 374
- complex.h, 373
- continuous integration, *Patrz* systemy ciągłej integracji
- Cook S. A., 235
- COPACOBANA, 332
- cout, 362
- CPU, *Patrz* procesor
- Cygwin, 383
- cykl Eulera, 274
- cykl skierowany, 273
- czas jednostkowy wykonania instrukcji, 165
- czas wykonania, 156, 161
- cząstkowy dobór, 297

D

DAG, 273, *Patrz* acykliczny graf skierowany
 DARPA, 314
 Data Encryption Standard, 332
 debugger, 30
 dekompozycja problemu, 47, 249
 delete, 92
 depthf.cpp, 292
 depth-first search, 291
 deque (klasa STL), 145
 derekursywacja, 181, 185, 190, 197
 eliminacja zmiennych lokalnych, 188
 metoda funkcji przeciwnych, 190
 schemat, 192
 typu if-else, 194
 typu while, 193
 z podwójnym wywołaniem rekurencyjnym, 196
 stos, 188
 DES, 332
 destruktor, 372
 destruktor (metoda), 92
 Dev C++, 395
 DFS, 291
 Diffie W., 333
 digraf, 271
 Dijkstra E., 24, 30, 287
 directed acyclic graph, 273
 directed graph, 271
 długość ciągów znakowych, 79
 do...while, 369
 dobor.cpp, 298
 domknięcie przechodnie grafu, 281
 double, 76
 drzewa, 125
 głębokość węzła, 125
 gry, 320
 kodowe, 344
 korzeń, 125
 liść, 125
 m-drzewa, 126
 ojciec, 128
 potomek lewy, 128
 potomek prawy, 128
 realizacja tablicowa, 128
 reprezentacja C++, 126
 rozpinające minimalne, 289
 syn, 128
 Uniwersalna Struktura Słownikowa, 133
 uporządkowane, 126
 węzły końcowe, 125
 węzły, 125
 wysokość węzła, 125

drzewa binarne, 84, 119, 125, 126, 129
 komórka, 126
 wypisywanie drzewa w postaci wrostkowej, 131
 wyrażenia arytmetyczne, 129
 wysokość, 126
 dyn.cpp, 261
 dziedzic.cpp, 377
 dziedzic.h, 377
 dziedziczenie własności, 376
 dziel i zwyciężaj, 209, 211, 246
 algorytm Strassena, 251
 mnożenie liczb całkowitych, 252
 mnożenie macierzy, 249
 odnajdywanie największego i najmniejszego elementu w tablicy, 247
 Quicksort, 253

E

Eckert J. P., 22
 EDVAC, 22
 eliminacja Gaussa, 308, 309
 eliminacja zmiennych lokalnych, 190, 192
 end-recursion, 184
 ENIAC, 22
 eratost.cpp, 357
 etapy konstrukcji programu, 19, 25, 28
 eukl.cpp, 27
 Euler L., 269

F

false, 76
 fib.cpp, 40
 fib-dyn.cpp, 260
 FIFO, 116, 117, *Patrz* kolejka FIFO
 First In First Out, 116
 Fleury H., 275
 float, 76
 Floyd R., 24
 floyd.cpp, 285
 floyd2.cpp, 286
 for, 33
 Ford L. R. Jr., 289
 format GIF, 351
 Forth, 130
 Fortran, 181
 FORTRAN, 312
 forward_list (klasa STL), 145
 fraktale, 33
 friend, *Patrz* funkcja zaprzyjaźniona
 funkcja
 Ackermanna, 174
 McCarthy'ego, 41, 50

O, 161, 162
 odwrotna, 191
 zaprzyjaźniona, 94, 373
 funkcje, 367
 H, 221, 222, 243
 mnożenie, 224
 suma modulo 2, 223
 inline, 102
 modulo, 243
 nagłówek, 46
 parametry domyślne, 46
 przeciążanie, 102
 wskaźniki do funkcji, 99
 wywołanie przez wartość, 44
 zaprzyjaźnione, 375
 fuzja list, 93

G

gauss.cpp, 310
 GCC, 382, 387
 GCD, 53
 generator liczb losowych, 19
 GIF, 70, 329, 348, 351
 GIF87a, 351
 GIG89a, 351
 głębokość węzła, 125
 głowa, 85
 GNU C++, 10, 239
 GNU Scientific Library, 311
 Go, 321
 Gödel K., 21
 Goldman A., 274
 Gosper R. W., 236
 goto, 182
 GPL (licencja), 382
 gra w „kółko i krzyżyk”, 321
 grafika rastrowa, 70
 grafika wektorowa, 70
 grafy, 84, 269
 acykliczne skierowane, 273
 algorytm Bellmana-Forda, 289
 algorytm Dijkstry, 287
 algorytm Fleury'ego, 275
 algorytm Floyd'a, 286
 algorytm Floyd'a-Warshalla, 284
 algorytm Kruskala, 290
 algorytm Prima, 290, 291
 algorytm Roy-Warshalla, 281
 cykl Eulera, 274
 cykl Hamiltona, 273
 cykle, 273
 DAG, 273
 diagonalne, 280
 digraf, 271
 domknięcie przechodnie, 281

eulerowskie, 274
 implementacja, 276
 kompozycja, 280
 liczba chromatyczna grafu, 271
 macierz sąsiedztwa, 276
 minimalizowanie konfliktów, 296
 minimalne drzewo rozpinające, 289
 nieskierowane, 272
 operacje, 279
 operacje matematyczne, 279
 parzysty stopień, 274
 planarne, 271
 poszukiwanie drogi, 283
 problem doboru, 296
 problem komiwojażera, 274
 przechodnie, 280
 przeszukiwanie, 291
 w głąb, 291, 292
 wszerz, 291, 294
 zstępujące, 292
 regularne, 272, 273, 274
 reprezentacja tablicowa, 276
 reprezentacja, 276
 skierowane, 271
 słownik węzłów, 276, 278
 spójne, 272
 stanów, 319
 stopień wejściowy wierzchołka, 271
 suma, 279
 symetria, 280
 tablica dwuwymiarowa, 276
 węzeł początkowy, 271
 węzły, 271
 wierzchołki, 271
 zbiory, 279
 greedy, *Patrz* algorytmy żarłoczne
 greedy.cpp, 256
 gry dwuosobowe, 320
 GSL, *Patrz* GNU Scientific Library, *Patrz* GNU Scientific Library

H

H (funkcja), 221, 243
 Hanoi, 185, 190
 hanoi.cpp, 186
 hanoi_it.cpp, 197
 hash.cpp, 230
 hashing, 220
 heap, *Patrz* sterta
 Heap sort, 206
 heap.cpp, 208
 Hellman M., 333
 hermetyzacja, 373
 heurystyka, 264, 291
 Hilbert D., 21

hill-climbing, 296
 historia maszyn algorytmicznych, 20
 Hoare C. A. R., 24
 Hollerith H., 21
 horner.cpp, 335
 Huffman(), 347

I

IBM, 21
 IBM 604, 22
 IBM 650, 22
 ICO, 70
 IFIP, 24
 iloczyn logiczny, 62
 implementacja grafów, 276
 include (dyrektywa), 362
 indukcja matematyczna, 30
 inline, 102
 insert.cpp, 201
 int, 76
 interfejs użytkownika, 25
 interpol.cpp, 305
 interpolacja funkcji, 304
 interpolacja funkcji metodą Lagrange'a, 304
 ISO 8859-2, 69
 ISO Latin-2, 69
 iteracje, 33, 369
 iteracyjne obliczanie wartości funkcji, 303
 iterator, 141, 143

J

Jacquard J. M., 20
 Java, 25
 język
 assemblera, 26
 C++, 11, 361
 prezentacji programów, 26
 programowania, 11, 26
 strukturalny, 26
 JPEG, 70

K

Karp R. M., 236
 klasa, 83, 373
 algorytmu, 158
 destruktor, 92
 O, 162
 klasy
 funkcja zaprzyjaźniona, 94
 metody statycznej, 376
 pola statyczne, 376
 składowe statyczne, 376
 szablonowe, 113

klasyfikacja problemów algorytmicznych, 29
 klucz, 221, 242, 278
 prywatny, 333
 publiczny, 333
 K-M-P, tablica przesunięć, 237
 K-M-P (algorytm), 235, 236
 kmp.cpp, 238
 Knuth D. E., 235
 kod
 ASCII, 68, 343
 kody sterujące, 68
 znaki podstawowe, 69
 Huffmana, 329, 343, 345
 nadmiarowy, 329
 nierównomierny, 329
 równomierny, 329
 warunkowy, 362
 wykonywalny, 25
 źródłowy, 25
 Kod Znak-Moduł, 66
 kodowanie danych, 327, 329
 3DES, 332
 asymetryczne, 332
 DES, 332
 klucz prywatny, 333
 LZW, 329, 348
 podpis cyfrowy, 333
 problem transmisji klucza, 332
 RSA, 333
 symetryczne, 331
 z kluczem publicznym, 329, 333
 za pomocą słownika, 348
 kodowanie liter za pomocą 5 bitów, 222
 kodowanie znaków, 68, 70
 kody ASCII, 329
 kolejka, 119, 294
 FIFO, 116
 LIFO, 112
 priorytetowa, 119
 kolejka.h, 117
 kolejka-string.cpp, 118
 komentarze, 362
 kompilacja, 10, 381, 387
 kompilator, 25, 182, 270, 382
 C++, 382
 GCC, 382, 387
 GNU C++, 239
 kompletne drzewo binarne, 119
 kompoz.cpp, 277, 280
 kompozycja grafów, 280
 kompresja, 327, 328, 340
 algorytm Huffmana, 345
 algorytmy, 341
 bezstratna, 341
 danych, 134
 GIF, 329, 348, 351

LZW, 348
 metoda Huffmana, 343
 poprzez modelowanie matematyczne, 341
 redundancja, 341
 RLE, 342
 stopień kompresji, 341
 stratna, 341
 szybkość działania, 341
 komputer, 22
 komputer kwantowy, 19, 339
 konstruktor, 372
 kontener (C++), 140
 kontroler wyводу, 316
 kopiec, 206
 korzeń (drzewa binarnego), 125
 kółko i krzyżyk, 320, 321, 323
 generowanie listy możliwych ruchów, 325
 kodowanie listy węzłów potomnych, 325
 linie otwarte, 323
 Kruskal J., 290
 kryptosystem RSA, 333
 kwadraty (Visual C++), 49
 kwadraty „parzyste”, 48
 kwadraty.cpp, 391

L

Last In First Out, 112
 LCS, 262
 lcs.cpp, 263
 Leibniz G. W., 20
 Lempel A., 348
 LF, 328
 liczby
 całkowite, 76
 reprezentacja, 67
 zespolone, 373
 zmiennopozycyjne, 76
 LIFO, 112
 linear.cpp, 217
 lineto(), 48
 Lisp, 316
 LISP, 27, 94
 list (klasa STL), 145
 lista sąsiedztwa, 276
 lista.cpp, 88, 103
 lista.h, 87
 lista2.cpp, 101, 103
 lista2.h, 98, 102
 lista-tab.cpp, 107
 listy, 84
 cykliczne, 111
 implementacja tablicowa, 106
 listy dwukierunkowe, 110
 struktura wewnętrzna, 110
 usuwanie elementów, 111

listy jednokierunkowe, 85
 dołączanie elementu, 108
 fuzja list, 93
 głowa, 85, 86
 metoda tablic równoległych, 108
 ogon, 85
 rekord natury informacyjnej, 85
 rekord o charakterze roboczym, 85
 reprezentacja tablicowa, 107
 struktura informacyjna, 85
 usuwanie elementów, 91, 100, 108
 usuwanie wskaźników, 104
 wady i zalety, 96
 wyszukiwanie, 100, 102
 listy rankingowe, 297
 liść (drzewa binarnego), 125
 long, 76
 longest common subsequence, 262
 Lovelace A., 21
 Lucas É., 185
 LZW, 329, 348
 kodowanie, 349

Ł

łamanie szyfrów, 339
 brute-force, 339
 dedukcja na podstawie fragmentu lub całości
 treści, 339
 kradzież klucza, 339
 metoda „na siłę”, 339

M

macierz
 kierowania ruchem, 282
 sąsiedztwa, 276
 trójkątna, 309
 main (funkcja), 361
 maksimum w tablicy liczb, 247
 malloc(), 86
 map (klasa STL), 149
 MARK 1, 22
 Markow A. A., 21
 maszyna
 analityczna, 21
 Moore’a, 272
 różnicowa, 21
 stanów skończonych, 273
 tkacka Jacquarda, 20
 Turinga, 21, 28, 29
 Mathcad, 301
 Mathematica, 301
 Matlab, 301
 Mauchly J. W., 22

McCarthy J., 24, 91, *Patrz* funkcja McCarthy'ego
 McCarthy.cpp, 41
 m-drzewa, 126
 merge.cpp, 210
 metadane, 71
 metoda

- C++, 372
- klasy, 83
- Floyda, *Patrz* metoda niezmienników
- funkcji przeciwnych, 190, 191
- Gaussa, 308, 309
- Huffmana, 343
- Newtona, 301
- niezmienników, 30
- programowania, 245
- Simpsona, 307, 308
- statyczna, 376
- Stirlinga, 305
- tablic równoległych, 108, 109
- transformacji kluczowej, 230

 metodologia programowania, 24
 metody prymitywne, 329
 miara złożoności obliczeniowej, 157
 miejsca zerowe funkcji, 301
 MinGW, 382
 minimalizowanie konfliktów, 296
 minimalne drzewo rozpinające, 289
 mini-max, 321, 325
 minimum w tablicy liczb, 247
 min-max.cpp, 247, 248
 Mitnick, 339
 mnożenie liczb całkowitych, 252
 mnożenie macierzy, 249
 model neuronu, 317
 model sortowania zewnętrznego, 213
 modelowanie

- matematyczne, 341
- obiektowe, 26
- problemów, 270
- rozwiązywanego zagadnienia, 318

 modulo, 243, 338
 moduł dialogowy, 316
 Moore J. S., 235
 Morris J. H., 235
 Morse S., 340
 motor wyводу, 316
 Muhammad ibn Musa al-Chuwarizmi, 17
 multiset (klasa STL), 149
 MYCIN, 316
 myślenie rekurencyjne, 46

N

n.cpp, 44
 najbardziej czasochłonna operacja, 160, 165
 najdłuższa wspólna podsekwencja, 261

największy wspólny dzielnik, 17
 napisy, 78
 negacja, 62
 Neumann, Johannes von, 22
 neuron, 317
 new, 86
 newton.cpp, 302
 niezmiennik, 30
 notacja dużego O, 162
 notacja Landaua, 162
 NWD, 53, *Patrz* największy wspólny dzielnik
 nwd.cpp, 57

O

O(1), 161
 O(2ⁿ), 162
 O(log n), 162
 O(n), 162
 O(N²), 162
 O(N³), 162
 obiekt, 83, 372
 oblicz.cpp, 359
 obliczanie wartości funkcji, 303
 obwód zamknięty, 282
 odnajdywanie największego i najmniejszego
 elementu w tablicy, 247
 odpluskwanie, 24
 odwrot2.cpp, 196
 Odwrotna Notacja Polska, 130, 132
 odwrotna.cpp, 191
 odwr-tab.cpp, 54
 ogon, 85
 ogrodzenie (Visual C++), 56
 ojciec, 128
 O-notacja, 161, 162, 163
 ONP, *Patrz* Odwrotna Notacja Polska
 operacje

- arytmetyczne, 363
- arytmetyczne na liczbach dwójkowych, 61
- logiczne, 363
- logiczne na liczbach dwójkowych, 62
- na grafach, 279
- na plikach, 370

 operandy, 129
 operator --, 363
 operator ++, 363
 operatory, 129
 opis słowny, 26
 optymalizacja algorytmów, 181
 optymalizacja programów, 176
 OR, 62

P

- palindro.cpp, 359
- pamięć, 199
- parametry domyślne, 46
- parametry programu main(), 370
- pętle, 33, 369
- pliki, 370
 - dołączane, 362
 - GIF, 352
 - wykonywalne, 25
- pliki.cpp, 371
- PNG, 70
- pochozna.cpp, 306
- podjęcie iteracyjne, 33
- podjęcie komponentowe, 23
- podjeżdżanie (metoda), 296
- podpis cyfrowy, 333
- podprogramy, 367
- podstawa systemu obliczeniowego, 60
- podwójne kluczowanie, 228, 229
- pola statyczne, 376
- pop(), 112, 114
- poprawność algorytmów, 29
- porównywanie sekwencji kodów, 262
- post2.cpp, 55
- post2-dod.cpp, 55
- postać uwikłana, 303
- Postscript, 130
- poszukiwanie miejsc zerowych funkcji, 301
- pot.cpp, 358
- potęga grafu, 280
- potomek lewy, 128
- potomek prawy, 128
- poziomy abstrakcji opisu, 26
- Pratt V. R., 235
- prawdopodobieństwa występowania liter w języku polskim, 345
- prefiks, 344
- prezentacja algorytmów, 26
- Prim R. C., 291
- private, 94
- private (sekcja), 88
- problem
 - 8 hetmanów, 266
 - doboru, 296, 298
 - cząstkowy dobór, 297
 - listy rankingowe, 297
 - komiwojażera, 274
 - konika szachowego, 319
 - mostów w Królewcu, 269
 - NP-zupełny, 274
 - optymalnego doboru, 270
 - plecakowy, 254
 - transmisji klucza, 332
 - wyrażania preferencji, 297
- procedury, 367
- procesor, 28
- program, 10, 361
 - etapy konstrukcji, 25
 - rekurencyjny, 36, 45, 169
 - warunki końcowe, 30
 - warunki wstępne, 30
- programowanie dynamiczne, 40, 259
 - ciąg Fibonacciego, 259
 - etapy, 258
 - najdłuższa wspólna podsekwencja, 261
 - równania z wieloma zmiennymi, 260
- programowanie obiektowe, 371
- PROLOG, 27, 94, 316
- prostota algorytmu, 155
- protected, 94, 377
- próbkowanie liniowe, 226, 230
- przeciążanie, 378
- przeciążenie funkcji, 102
- przedefiniowanie operatora, 93, 374
- przedrostek, 344
- przepięnienie stosu, 40, 175
- przesunięcie bitowe, 62
- przeszukiwanie, 217
 - binarne, 51, 171, 218, 253
 - klasa algorytmu, 172
 - grafów, 291
 - A*, 295
 - BFS, 291
 - DFS, 291
 - ekspansja węzłów, 292
 - metoda podjeżdżania, 296
 - strategie, 295
 - w głąb, 291, 292
 - wszerz, 291, 294
 - z powracaniem, 295
 - zstępujące przeszukiwanie, 292
 - liniowe, 217
 - tekstów
 - algorytm K-M-P, 235, 236
 - algorytm typu brute-force, 233
- przetwarzanie równoległe, 19
- przydzielanie pamięci, 86
- przypadek
 - najgorszy, 167
 - najlepszy, 167
 - średni, 168
 - typowy, 168
- public (sekcja), 88
- push(), 112, 114

Q

- qsort (funkcja biblioteczna), 214
- qsort2.cpp, 215
- queue (klasa STL), 148

quick.cpp, 206
 Quicksort, 203, 253
 podział tablicy, 204

R

Rabin M. O., 236
 RAM, 199
 RC, *Patrz* równanie charakterystyczne
 redukcja wsteczna, 309
 redundancja, 341
 ref.cpp, 82
 referencje, 365
 reguła mini-max, 322
 rekordy, 80
 reksearch.cpp, 35
 rekurencja, 33, 169, 245
 ciąg Fibonacciego, 38
 dekompozycja problemu, 47
 drzewo wywołań, 37
 ilustracja, 35
 kontekst, 46
 kwadraty „parzyste”, 48
 myślenie rekurencyjne, 46
 naturalna, 45
 nieskończona ilości wywołań, 43
 pamięciożerność, 50
 poprawność definicji, 44
 poziomy, 37, 38, 46
 problemy, 38
 przebiegi, 33
 przekazywanie parametrów, 37
 przepełnienie stosu, 40
 rozkład na problemy elementarne, 34
 silnia, 45
 skrośna rekurencja, 50
 spirała, 47
 sposób wykonywania programu, 36
 typy programów, 45
 uwagi praktyczne, 50
 z parametrem dodatkowym, 45, 46, 188
 zajętość pamięci, 41
 zakończenie algorytmu, 34
 zakończenie programu, 36
 zasada działania, 34
 zmnieszanie rozmiaru problemu, 50
 rekurencja (definicja), 33
 rekursja, *Patrz* rekurencja
 relacje binarne, 280
 relaksacja, 288
 reprezentacja
 grafów, 276
 problemów, 318
 tablicowa grafów, 292

reszta.cpp, 257
 reverse-engineeringu, 332
 Reversi, 321
 Rivest R., 333
 rk.cpp, 244
 RLE, 342
 RO, *Patrz* rozwiązanie ogólne
 routing matrix, 282
 rozdzielenie i scalanie połączone
 z sortowaniem, 212
 rozkład logarytmiczny, 171
 rozmiar danych, 157
 rozwiązanie
 ogólne, 170
 równania rekurencyjnego, 170
 szczególne, 170
 rozwiązywanie układów równań, 308
 równania z wieloma zmiennymi, 260
 równanie charakterystyczne, 169
 różnica symetryczna, 62
 różniczkowanie funkcji, 305
 RS, *Patrz* rozwiązanie szczególne
 RSA, 332, 333
 ruchy dozwolone, 319
 Run Length Encoding, 342

S

scalaj.cpp, 209
 scalanie zbiorów posortowanych, 209
 schemat Hornera, 303, 335
 schematy derekursywacji, 192
 if-else, 194
 while, 193
 z podwójnym wywołaniem
 rekurencyjnym, 196
 sciezka.cpp, 283
 sdw.cpp, 43
 Segmentation fault, 43
 set (klasa STL), 148
 shaker.cpp, 203
 Shaker-sort, 203
 Shamir A., 333
 SI, *Patrz* Sztuczna Inteligencja
 sieci neuronowe, 317
 nauczanie, 318
 struktura, 318
 silnia, 37, 45, 157, 160, 170, 187
 silnia (definicja), 37
 silnia.cpp, 37, 45
 silnia_ite.cpp, 57
 simpson.cpp, 308
 sito Erastotenesa, 355, 357
 składowe statyczne klas, 376
 słownik węzłów, 278, 292

- słowniki, 133
 - słownikowa metoda kodowania, 349, 353
 - Solution Explorer, 393
 - sortowanie, 199
 - algorytm klasy $O(N^2)$, 200, 201
 - algorytmy, 199
 - bąbelkowe, 201
 - Heap sort, 206
 - przez kopcowanie, 206
 - przez scalanie, 209
 - przez wstawianie, 200
 - przez wytrząsanie, 203
 - sterta, 124
 - szybkie, 203
 - wewnętrzne, 199
 - zewnętrzne, 199, 211
 - spirala, 47, 48
 - SRL, *Patrz* szereg rekurencyjny liniowy
 - SSS*, *Patrz* algorytm SSS*
 - stack (klasa STL), 147
 - Stack overflow, 40, 43, 175
 - sterta, 119, 122
 - implementacja, 122
 - kolejka priorytetowa, 121
 - sortowanie, 124
 - tworzenie, 120
 - wstawianie elementów, 121, 122
 - sterta.cpp, 124
 - sterta.h, 122
 - STL, 12, 84, 140, 219
 - stl-array.cpp, 142, 145
 - stl-lista.cpp, 146
 - stl-map.cpp, 150
 - stl-set.cpp, 149
 - stl-stos.cpp, 147, 148
 - stl-vector.cpp, 144
 - stopień kompresji, 341
 - stopień wejściowy wierzchołka grafu, 271
 - stos, 111
 - pop(), 112, 114
 - push(), 112, 114
 - zasada działania, 112
 - stos.cpp, 115
 - stos.h, 113
 - Strassen V., 250
 - strategia gry, 320
 - strategia przeszukiwania, 321
 - string, 79
 - string.cpp, 80
 - struct.cpp, 82
 - struktury, 366
 - struktury danych, 84
 - drzewa, 125
 - grafy, 269
 - kolejka FIFO, 116
 - kolejka priorytetowa, 119
 - listy jednokierunkowe, 85
 - sterta, 119
 - stos, 111
 - zbiory, 138
 - suma grafów, 279
 - suma logiczna, 62
 - suma modulo 2, 62, 223
 - suma modulo Rmax, 223
 - SVG, 70
 - switch, 368
 - symulacja inteligentnego zachowania, 313
 - syn, 128
 - system
 - ciągłej integracji, 23
 - dwójkowy, 60
 - dziesiętny, 60
 - ekspercki, 315
 - baza wiedzy, 315
 - kontroler wyводу, 316
 - moduł dialogowy, 316
 - motor wyводу, 316
 - MYCIN, 316
 - sztuczny lekarz, 316
 - kodujący z kluczem publicznym, 333
 - operacyjny, 25
 - ósemkowy, 65
 - pozycyjny, 60
 - szesnastkowy, 65
 - uzupełnienia dwójkowego, 66
 - systemsort.cpp, 212
 - szachy, 321
 - szereg rekurencyjny liniowy, 169
 - Sztuczna Inteligencja, 313, 314
 - cele, 314
 - drzewa gier, 320
 - grafy stanów, 319
 - reprezentacja problemów, 318
 - sieci neuronowe, 317
 - sztuczny lekarz, 316
 - szukaj.cpp, 166
 - szukaj-txt.cpp, 234
 - szyfr blokowy, 332
- ## T
- tablica
 - dwuwymiarowa, 276
 - przesunięć, 237
 - różnic centralnych, 305
 - tablice, 77, 366
 - implementacja kolejki FIFO, 116
 - implementacja listy, 106, 107
 - reprezentacja drzewa, 129
 - równoległe, 128
 - tablice.cpp, 77
 - techniki optymalizacji programów, 176

techniki programowania, 245
 tekst źródłowy, 25
 teoria automatów, 272
 teoria gier, 291
 test Turinga, 29, 314
 this (C++), 372
 tictac.cpp, 321, 323, 324
 transformacja kluczowa, 220, 230

- funkcje H, 221
- podwójne kluczowanie, 228
- próbkowanie liniowe, 226
- strefa podstawowa, 226
- strefa przepełnienia, 226
- tablice, 226
- zastosowania, 229

 traveling salesman problem, 274
 true, 76
 Turing A. M., 21, 314
 tworzenie programu, 19, 25, 28
 typy danych, 76

- podstawowe, 76
- wbudowane, 76
- zakres dozwolonych wartości, 76

 typy złożoności obliczeniowej, 166

U

U2, *Patrz* system uzupełnienia dwójkowego
 Unicode, 70
 unie, 81
 union, 81
 UNIVAC 1, 22
 Uniwersalna Struktura Słownikowa, 133
 using namespace, 362
 USS, *Patrz* Uniwersalna Struktura Słownikowa
 uss.cpp, 135

V

vector (klasa STL), 76, 143
 Visual C++ Express Edition

- projekt konsolowy, 389, 391

 void, 367

W

warshall.cpp, 282
 wartf.cpp, 304
 wartości logiczne, 76
 warunki końcowe, 30
 warunki wstępne, 30
 wektory, 110
 Welch T., 348

while, 33
 wielom.cpp, 336
 wielom2.cpp, 336
 wielomiany, 335
 wielowątkowość, 177
 wieże Hanoi, 185, 190, 197, 253
 Wirth N., 24
 wsk_fun2.cpp, 100
 wskaźniki, 76, 364
 wskaźniki do funkcji, 99
 wsk-fun.cpp, 99
 wydajność oprogramowania, 157
 wyrazen.cpp, 129, 130
 wyrażanie preferencji, 297
 wyrażenia arytmetyczne, 129, 130
 wysokość drzewa binarnego, 126
 wysokość węzła, 125
 wywołanie przez wartość, 44
 wywołanie terminalne, 184
 wyznacznik Vandermonde'a, 304
 wzorce projektowe, 23
 wzór Simpsona, 308
 wzór Stirlinga, 305

X

XOR, 62, 330
 xor.cpp, 330

Z

zajętość pamięci, 156
 zamiana dziedziny równania rekurencyjnego, 174
 zbior.cpp, 139
 zbiory, 138, 276

- implementacja, 139

 zerowanie fragmentu tablicy, 163
 Ziv J., 348
 zlib1.dll, 383
 złożoność

- algorytmów, 155
- czasowa, 156
- obliczeniowa, 155, 172
- pamięciowa, 156
- praktyczna, 161
- teoretyczna, 161


 zmienne, 10, 67

- dynamiczne, 364
- globalne, 189
- lokalne, 188, 189

 znaki, 68, 70, 76
 znaki.cpp, 79

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

Algorytmy i struktury danych — szybko, łatwo, skutecznie!

- **Poznaj najważniejsze algorytmy i techniki programistyczne**
- **Naucz się skutecznie wykorzystywać typy i struktury danych**
- **Dowiedz się, jak w praktyce zastosować zdobytą wiedzę**

Algorytmika to dziedzina, która w ciągu ostatnich kilkudziesięciu lat dostarczyła wielu efektywnych narzędzi wspomagających rozwiązywanie różnorodnych zagadnień za pomocą komputera. Dla niektórych stanowi swego rodzaju książkę kucharską, do której sięgają jedynie po wybrane przepisy, a dla innych — pole do rozwinięcia umiejętności skutecznego rozwiązywania problemów i szkołę niestandardowego myślenia. Niezależnie od podejścia jest to dziedzina, z którą wypada się zapoznać, jeśli ma się ambicję zostać zawodowym programistą lub po prostu być osobą nowoczesną i wszechstronnie wykształconą.

Ten przewodnik prezentuje szerokie spektrum zagadnień algorytmicznych, najważniejsze informacje na temat struktur danych, technik rekurencyjnych i złożonych metod algorytmicznych. Teoria jest tu poparta przykładowymi programami napisanymi w języku C++, łatwymi do analizy i skompilowania z wykorzystaniem standardowych narzędzi. Autor nie poprzestaje na suchym kodzie, lecz stara się przedstawić praktyczne zastosowanie opisywanych rozwiązań. Podręcznik przyda się zarówno osobom niemającym solidnych podstaw teoretycznych, jak i specjalistom, którzy zawodowo zajmują się programowaniem. Nowe wydanie zostało gruntownie odświeżone i poprawione, a listingi dostosowane do wymagań najnowszych kompilatorów. Książka zawiera opis zasad kompilacji dla środowiska Visual Studio 2017 i kilku wybranych środowisk używających GNU C++ (Dev-C++ i Cygwin).

- **Historia algorytmiki**
- **Mechanizm rekurencji**
- **Systemy liczbowe i kodowanie**
- **Typy i struktury danych**
- **Analiza złożoności algorytmów**
- **Derekursywacja algorytmów**
- **Optymalizacja algorytmów**
- **Algorytmy sortowania i wyszukiwania**
- **Elementy algorytmiki grafów**
- **Sztuczna inteligencja**
- **Szyfrowanie i kompresja danych**
- **Biblioteka STL**

Jedyny podręcznik do algorytmiki, którego będziesz potrzebować!

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 978-83-283-5374-9



9 788328 353749

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł