

» Idź do

- Spis treści
- Przykładowy rozdział

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

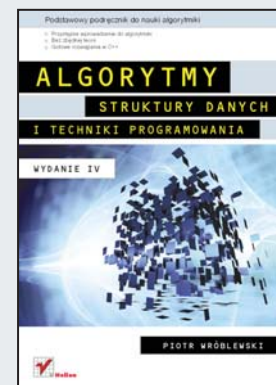
- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Algorytmy, struktury danych i techniki programowania. Wydanie IV

Autor: [Piotr Wróblewski](#)
ISBN: 978-83-246-2306-8
Format: 158×235, stron: 452



Podstawowy podręcznik do nauki algorytmiki

- Przystępne wprowadzenie do algorytmiki
- Bez zbędnej teorii
- Gotowe rozwiązania w C++

Oto kolejne wydanie sprawdzonej i cenionej przez programistów, wykładowców oraz studentów książki, będącej podstawowym podręcznikiem do nauki algorytmiki. W pierwszej kolejności autor zapozna Cię z elementarnymi zagadnieniami z tej dziedziny oraz wyjaśni, skąd bierze się tak szybki postęp w tej dyscyplinie nauki. Podczas dalszej lektury poznasz takie pojęcia, jak rekurencja, analiza złożoności oraz algorytmy sortowania i przeszukiwania czy algorytmy numeryczne. Szybko opanujesz metody optymalizacji algorytmów, sposoby kodowania i kompresji danych oraz elementy algorytmiki grafów. Przedstawione w książce algorytmy zilustrowane zostały przykładowymi kodami źródłowymi w C++ , ułatwiającymi zrozumienie poznawanych zagadnień. Przejrzysta forma, praktyczne przykłady oraz przystępny język sprawiają, że książka ta pozwala szybko, a także bezboleśnie opanować zarówno algorytmy, jak i struktury danych oraz najlepsze techniki programowania.

- Historia algorytmiki
- Wykorzystanie rekurencji
- Analiza złożoności algorytmów
- Algorytmy sortowania
- Algorytmy przeszukiwania
- Przeszukiwanie tekstów
- Struktury danych i ich implementacja
- Optymalizacja algorytmów
- Zaawansowane techniki programowania
- Wykorzystanie grafów
- Wprowadzenie do sztucznej inteligencji
- Kodowanie i kompresja danych
- Algorytmy numeryczne
- Poradnik kompilacji i uruchamiania programów (GCC, DevC++, Microsoft Visual C++ Express Edition).

Szybko i bezboleśnie opanuj wszystkie zagadnienia algorytmiki!

Spis treści

Przedmowa	9
Rozdział 1. Zanim wystartujemy	17
Jak to wcześniej bywało, czyli wyjątki z historii maszyn algorytmicznych	18
Jak to się niedawno odbyło, czyli o tym, kto „wymyślił” metodologię programowania	21
Proces koncepcji programów	22
Poziomy abstrakcji opisu i wybór języka	23
Poprawność algorytmów	24
Rozdział 2. Rekurencja	27
Definicja rekurencji	27
Ilustracja pojęcia rekurencji	28
Jak wykonują się programy rekurencyjne?	30
Niebezpieczeństwa rekurencji	31
Ciąg Fibonacciego	31
Stack overflow!	33
Pułapek ciąg dalszy	34
Stąd do wieczności	34
Definicja poprawna, ale...	35
Typy programów rekurencyjnych	36
Myślenie rekurencyjne	38
Przykład 1.: Spirala	38
Przykład 2.: Kwadraty „parzyste”	40
Uwagi praktyczne na temat technik rekurencyjnych	41
Zadania	42
Rozwiązania i wskazówki do zadań	44
Rozdział 3. Analiza złożoności algorytmów	49
Definicje i przykłady	50
Jeszcze raz funkcja silnia	53
Zerowanie fragmentu tablicy	57
Wpadamy w pułapkę	58
Różne typy złożoności obliczeniowej	59
Nowe zadanie: uprościć obliczenia!	61

Analiza programów rekurencyjnych	62
Terminologia i definicje	62
Ilustracja metody na przykładzie	63
Rozkład logarytmiczny	64
Zamiana dziedziny równania rekurencyjnego	66
Funkcja Ackermanna, czyli coś dla smakoszy	66
Złożoność obliczeniowa to nie religia!	68
Techniki optymalizacji programów	68
Zadania	69
Rozwiązania i wskazówki do zadań	70
Rozdział 4. Algorytmy sortowania	73
Sortowanie przez wstawianie, algorytm klasy $O(N^2)$	74
Sortowanie bąbelkowe, algorytm klasy $O(N^2)$	75
Quicksort, algorytm klasy $O(N \log N)$	77
Heap Sort — sortowanie przez kopcowanie	80
Scalanie zbiorów posortowanych	82
Sortowanie przez scalanie	83
Sortowanie zewnętrzne	84
Uwagi praktyczne	87
Rozdział 5. Typy i struktury danych	89
Typy podstawowe i złożone	89
Ciągi znaków i napisy w C++	90
Abstrakcyjne struktury danych	92
Listy jednokierunkowe	93
Tablicowa implementacja list	115
Stos	119
Kolejki FIFO	123
Stery i kolejki priorytetowe	125
Drzewa i ich reprezentacje	131
Zbiory	143
Zadania	145
Rozwiązania zadań	146
Rozdział 6. Derekursywacja i optymalizacja algorytmów	147
Jak pracuje kompilator?	148
Odrobina formalizmu nie zaszkodzi!	150
Kilka przykładów derekursywacji algorytmów	151
Derekursywacja z wykorzystaniem stosu	154
Eliminacja zmiennych lokalnych	154
Metoda funkcji przeciwnych	156
Klasyczne schematy derekursywacji	158
Schemat typu while	159
Schemat typu if-else	160
Schemat z podwójnym wywołaniem rekurencyjnym	162
Podsumowanie	163
Rozdział 7. Algorytmy przeszukiwania	165
Przeszukiwanie liniowe	165
Przeszukiwanie binarne	166
Transformacja kluczowa (hashing)	167
W poszukiwaniu funkcji H	169
Najbardziej znane funkcje H	169
Obsługa konfliktów dostępu	171

Powrót do źródeł	172
Jeszcze raz tablice!	173
Próbkowanie liniowe	173
Podwójne kluczowanie	175
Zastosowania transformacji kluczowej	176
Podsumowanie metod transformacji kluczowej	176
Rozdział 8. Przeszukiwanie tekstów	179
Algorytm typu brute-force	179
Nowe algorytmy poszukiwań	181
Algorytm K-M-P	182
Algorytm Boyera i Moore'a	185
Algorytm Rabina i Karpa	187
Rozdział 9. Zaawansowane techniki programowania	191
Programowanie typu „dziel i zwyciężaj”	192
Odszukiwanie minimum i maksimum w tablicy liczb	193
Mnożenie macierzy o rozmiarze $N \times N$	195
Mnożenie liczb całkowitych	197
Inne znane algorytmy „dziel i zwyciężaj”	198
Algorytmy „żarłoczne”, czyli przekąsić coś nadszedł już czas... ..	199
Problem plecakowy, czyli niełatwe jest życie turysty piechura	200
Programowanie dynamiczne	202
Ciąg Fibonacciego	203
Równania z wieloma zmiennymi	204
Najdłuższa wspólna podsekwencja	205
Inne techniki programowania	208
Uwagi bibliograficzne	210
Rozdział 10. Elementy algorytmiki grafów	211
Definicje i pojęcia podstawowe	212
Cykle w grafach	214
Sposoby reprezentacji grafów	217
Reprezentacja tablicowa	217
Słowniki węzłów	218
Listy kontra zbiory	219
Podstawowe operacje na grafach	220
Suma grafów	220
Kompozycja grafów	220
Potęga grafu	220
Algorytm Roy-Warshalla	221
Algorytm Floyda-Warshalla	224
Algorytm Dijkstry	227
Algorytm Bellmana-Forda	228
Drzewo rozpinające minimalne	228
Algorytm Kruskala	229
Algorytm Prima	230
Przeszukiwanie grafów	230
Strategia „w głąb” (przeszukiwanie zstępujące)	231
Strategia „wszerz”	232
Inne strategie przeszukiwania	234
Problem właściwego doboru	235
Podsumowanie	239
Zadania	239

Rozdział 11. Algorytmy numeryczne	241
Poszukiwanie miejsc zerowych funkcji	241
Iteracyjne obliczanie wartości funkcji	243
Interpolacja funkcji metodą Lagrange'a	244
Różniczkowanie funkcji	245
Całkowanie funkcji metodą Simpsona	247
Rozwiązywanie układów równań liniowych metodą Gaussa	248
Uwagi końcowe	251
Rozdział 12. Czy komputery mogą myśleć?	253
Przegląd obszarów zainteresowań Sztucznej Inteligencji	254
Systemy eksperckie	255
Sieci neuronowe	256
Reprezentacja problemów	257
Ćwiczenie 1.	258
Gry dwuosobowe i drzewa gier	259
Algorytm mini-max	260
Rozdział 13. Kodowanie i kompresja danych	265
Kodowanie danych i arytmetyka dużych liczb	267
Kodowanie symetryczne	267
Kodowanie asymetryczne	268
Metody prymitywne	274
Łamanie szyfrów	275
Techniki kompresji danych	275
Kompresja poprzez modelowanie matematyczne	277
Kompresja metodą RLE	278
Kompresja danych metodą Huffmana	279
Kodowanie LZW	283
Idea kodowania słownikowego na przykładach	284
Opis formatu GIF	286
Rozdział 14. Zadania różne	289
Teksty zadań	289
Rozwiązania	291
Dodatek A Poznaj C++ w pięć minut!	295
Elementy języka C++ na przykładach	295
Pierwszy program	295
Dyrektywa #include	296
Podprogramy	296
Procedury	296
Funkcje	297
Operacje arytmetyczne	298
Operacje logiczne	298
Wskaźniki i zmienne dynamiczne	299
Referencje	300
Typy złożone	300
Tablice	300
Rekordy	301
Instrukcja switch	301
Iteracje	302
Struktury rekurencyjne	303
Parametry programu main()	303
Operacje na plikach w C++	303

Programowanie obiektowe w C++	304
Terminologia	304
Obiekty na przykładzie	305
Składowe statyczne klas	308
Metody stałe klas	308
Dziedziczenie własności	308
Kod warunkowy w C++	311
Dodatek B Systemy obliczeniowe w pigułce	313
Kilka definicji	313
System dwójkowy	313
Operacje arytmetyczne na liczbach dwójkowych	315
Operacje logiczne na liczbach dwójkowych	315
System ósemkowy	317
System szesnastkowy	317
Zmienne w pamięci komputera	317
Kodowanie znaków	318
Dodatek C Kompilowanie programów przykładowych	321
Zawartość archiwum ZIP na ftp	321
Darmowe kompilatory C++	321
Kompilacja i uruchamianie	322
GCC	322
Visual C++ Express Edition	323
Dev C++	327
Literatura	329
Spis tabel	331
Spis ilustracji	333
Skorowidz	339

Rozdział 8.

Przeszukiwanie tekstów

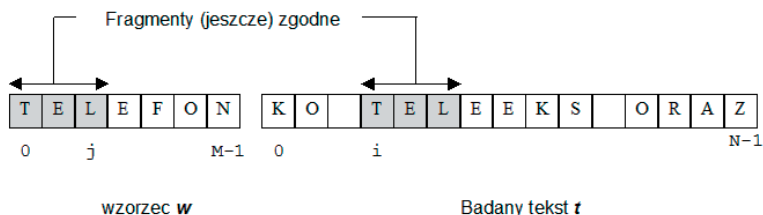
Zanim na dobre zanurzymy się w lekturę nowego rozdziału, należy wyjaśnić pewne nieporozumienie, które może towarzyszyć jego tytułowi. Otóż za *tekst* będziemy uważali ciąg znaków w sensie informatycznym. Nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną”! Tekstem będzie na przykład również ciąg bitów¹, który tylko przez umowność może być podzielony na równej wielkości porcje, którym przyporządkowano pewien kod liczbowy².

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

Algorytm typu brute-force

Zadaniem, które będziemy usiłovali wspólnie rozwiązać, jest poszukiwanie wzorca³ w o długości M znaków w tekście t o długości N . Z łatwością możemy zaproponować dość oczywisty algorytm rozwiązujący to zadanie, bazując na pomysłach symbolicznie przedstawionych na rysunku 8.1.

Rysunek 8.1.
Algorytm typu
brute-force
przeszukiwania tekstu



Zarezerwujmy indeksy j i i do poruszania się odpowiednio we wzorcu i 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 się jednak powinno stać z indeksami i oraz j podczas stwierdzenia niezgodności znaków? W takiej sytuacji całe poszukiwanie kończy się porażką, co zmusza nas do anulowania „szarej strefy” zgodności. Czynimy to poprzez cofnięcie się w tekście o to, co było zgodne, czyli o $j-1$

¹ Reprezentujący np. pamięć ekranu.

² Np. ASCII lub dowolny inny.

³ Ang. *pattern matching*.

znaków, wyzerowując przy okazji j . Omówmy jeszcze moment stwierdzenia całkowitej zgodności wzorca z tekstem. Kiedy to nastąpi? Otóż nietrudno zauważyć, że podczas stwierdzenia zgodności ostatniego znaku j powinno zwrócić się z M . Możemy wówczas łatwo odtworzyć pozycję, od której wzorec startuje w badanym tekście: będzie to oczywiście $i-M$.

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



txt-1.cpp

```
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;
}
```

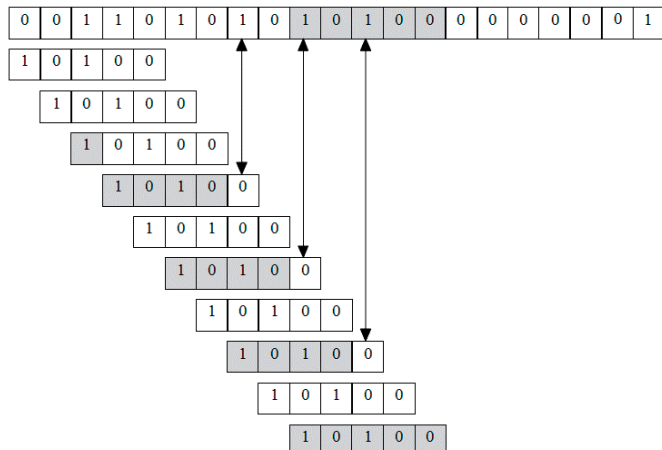
Sposób korzystania z funkcji `szukaj` jest przedstawiony na przykładzie następującej funkcji `main`:

```
int main()
{
    char *b="abrakadabra",*a="rak";
    cout << szukaj(a,b) << endl;
}
```

Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorec, lub -1 w przypadku, gdy poszukiwany tekst nie został odnaleziony — jest to znana nam już doskonale konwencja. Przypatrzmy się dokładniej przykładowi poszukiwania wzorca `10100` w pewnym tekście binarnym (patrz rysunek 8.2).

Rysunek 8.2.

„Falszywe starty”
podczas poszukiwania



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

Na podstawie zobrazowanego przykładu możemy spróbować wymyślić taki najgorszy tekst i wzorec, dla których proces poszukiwania będzie trwał możliwie najdłużej. Chodzi oczywiście zarówno o tekst, jak i wzorec złożone z samych zer i zakończone jedyneką⁴.

Spróbujmy obliczyć klasę tego algorytmu dla opisanego przed chwilą ekstremalnego najgorszego przypadku. Obliczenie nie należy do skomplikowanych czynności: zakładając, że restart algorytmu będzie konieczny $(N-1) \cdot (M-2) = N \cdot M - 1$ razy, i wiedząc, że podczas każdego cyklu jest konieczne wykonanie M porównań, otrzymujemy natychmiast $M(N-M+1)$, czyli około⁵ $M \cdot N$.

Zaprezentowany w tym paragrafie algorytm wykorzystuje komputer jako bezmyślne, ale sprawne liczydło⁶. 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 na tyle 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 1970 roku S. A. 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 D. E. Knuth i V. R. Pratt otrzymali na jego podstawie algorytm, który można już było zaimplementować w komputerze — ukazując przy okazji, iż pomiędzy praktycznymi realizacjami a rozważaniami teoretycznymi nie istnieje wcale aż tak ogromna przepaść, jakby się to mogło wydawać. W tym samym czasie J. H. Morris odkrył dokładnie ten sam algorytm jako rozwiązanie problemu, który napotkał podczas praktycznej implementacji edytora tekstu. Algorytm $K-M-P$ — bo tak będziemy go dalej zwali — jest jednym z przykładów dość częstych w nauce odkryć równoległych: z jakichś niewiadomych powodów nagle kilku pracujących osobno ludzie 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 roku. W międzyczasie pojawił się kolejny „cudowny” algorytm, tym razem autorstwa R. S. Boyera i J. S. Moore’a, który okazał się w pewnych zastosowaniach znacznie szybszy od algorytmu $K-M-P$. Został on również równolegle wynaleziony (odkryty?) przez R. W. Gospera. Oba te algorytmy są jednak dość trudne do zrozumienia bez pogłębionej analizy, co utrudniło ich rozpropagowanie.

W roku 1980 R. M. Karp i M. O. 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 przez nas algorytmu typu *brute-force*. Na dodatek jest to algorytm, który można względnie łatwo uogólnić na przypadek poszukiwania w tablicach 2-wymiarowych, co czyni go potencjalnie użytecznym w obróbce obrazów.

W następnych trzech sekcjach szczegółowo omówimy sobie wspomniane w tym przeglądzie historycznym algorytmy.

⁴ Zera i jedyнки symbolizują tu dwa różne od siebie znaki.

⁵ Zwykle M będzie znacznie mniejsze niż N .

⁶ Termin *brute-force* jeden z moich znajomych ślicznie przetłumaczył jako „metodę mastodonta”.

Algorytm K-M-P

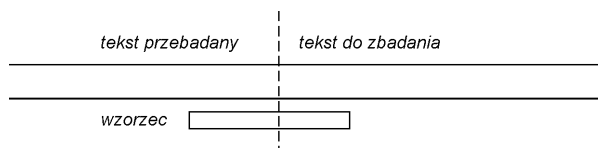
Wadą algorytmu *brute-force* jest jego czułość na konfigurację danych: fałszywe restarty są tu bardzo kosztowne; w analizie tekstu cofamy się o całą długość wzorca, zapominając po drodze wszystko, co przetestowaliśmy do tej pory. Narzuca się tu niejako chęć skorzystania z informacji, które już w pewien 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. Przykładowo: jeśli wiemy na pewno, iż w poszukiwanym wzorcu jego pierwszy znak nie pojawia się już w nim w ogóle⁷, to w razie restartu nie musimy cofać wskaźnika i o $j-1$ pozycji, jak to było poprzednio (patrz listing *txt-1.cpp*). W tym przypadku możemy po prostu zinkrementować i , wiedząc, że ewentualne powtórzenie poszukiwań na pewno nic by już nie dało. Owszem, można się łatwo zgodzić z twierdzeniem, iż 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 *K-M-P* 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 wzorca⁸ należy zmodyfikować ten indeks, wykorzystując informację zawartą w już zbadanej „szarej strefie” zgodności.

Brzmi to wszystko (zapewne) niesłuchanie tajemniczo, pora więc jak najszybciej wyjaśnić tę sprawę, aby uniknąć możliwych nieporozumień. Popatrzmy w tym celu na rysunek 8.3.

Rysunek 8.3.

Wyszukiwanie optymalnego przesunięcia w algorytmie *K-M-P*



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 umieszczony jest powyżej wzorca. W pewnym momencie może okazać się, że następuje pokrycie obu tych części. Zatrzymujemy wówczas przesuwanie i kontynuujemy testowanie (znak po znaku) zgodności obu części znajdujących się za kreską pionową.

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 on był, jest ukryta w stwierdzeniu „ $j-1$ znaków było zgodnych” — w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie sam wzorec, odkryć poszukiwane optymalne przesunięcie. Na tym właśnie spostrzeżeniu opiera się idea algorytmu *K-M-P*. 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 owych przesunięć, popatrzmy na efekt ich działania na kilku kolejnych przykładach. Na rysunku 8.4 możemy dostrzec, iż na siódmej pozycji wzorca⁹ (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

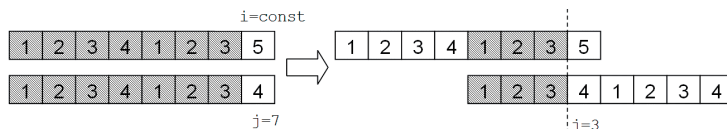
⁷ Przykład: „ABBBBBBB” — znak ‘A’ wystąpił tylko jeden raz.

⁸ Lub i w przypadku badanego tekstu.

⁹ Licząc indeksy tablicy tradycyjnie od zera.

Rysunek 8.4.

Przesuwanie się wzorca
w algorytmie *K-M-P* (1)

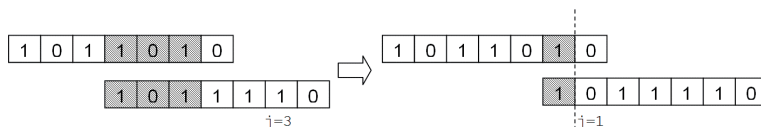


Jeśli zostawimy indeks i w spokoju, to — modyfikując wyłącznie j — możemy bez problemu kontynuować przeszukiwanie. Jakie jest optymalne przesunięcie wzorca? Ślizgając go wolno w prawo (patrz rysunek 8.4), doprowadzamy w pewnym momencie do nałożenia się ciągów 123 przed kreską — cała strefa niezgodności została wyprowadzona na prawo i ewentualne dalsze testowanie może być kontynuowane!

Analogiczny przykład znajduje się na rysunku 8.5.

Rysunek 8.5.

Przesuwanie się wzorca
w algorytmie *K-M-P* (2)



Tym razem niezgodność wystąpiła na pozycji $j=3$. Dokonując — podobnie jak poprzednio — przesuwania wzorca w prawo, zauważamy, iż 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 potrzeb algorytmu *K-M-P* konieczne okazuje się wprowadzenie tablicy przesunień `int shift[M]`. Sposób jej zastosowania będzie następujący: jeśli na pozycji j wystąpiła niezgodność znaków, to kolejną wartością j będzie `shift[j]`. Nie wnikając chwilowo w sposób inicjalizacji tej tablicy (odmiennej oczywiście dla każdego wzorca), możemy natychmiast podać algorytm *K-M-P*, który w konstrukcji jest niemal dokładną kopią algorytmu typu *brute-force*:



Listing

kmp.cpp

```
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 nam j .

Pozostaje do omówienia sposób konstrukcji tablicy `shift[M]`. Jej obliczenie powinno nastąpić przed wywołaniem funkcji `kmp`, co sugeruje, iż w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona praktycznie identyczna z `kmp` z tą tylko różnicą, iż 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];
}

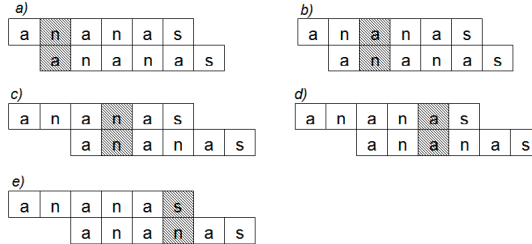
```

Sens tego algorytmu jest następujący: tuż po inkrementacji i i 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* (patrz rysunek 8.6). 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 rysunek 8.5).

Rysunek 8.6.

Optymalne przesunięcia wzorca „ananas” w algorytmie *K-M-P*



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

Algorytm *K-M-P* można zoptymalizować, jeśli znamy z góry wzorce, których będziemy poszukiwać. Przykładowo: jeśli bardzo często zdarza nam się szukać w tekstach słowa *ananas*, to w funkcję `kmp` można wbudować tablicę przesunięć:



Listing

ananas.cpp

```

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;
}

```

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 zwinym kodem wynikowym asemblerowym, jest zatem bardzo szybka. Posiadacze kompilatorów, które umożliwiają generację kodu wynikowego jako tzw. „assembly output”¹⁰, mogą z łatwością sprawdzić różnice 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 około 170 linii kodu asemblerowego, natomiast `kmp_ananas` zmieściła się w ok. 100 liniach. (Patrz: pliki z rozszerzeniem `s` na dyskiecie dla kompilatora GNU lub `asm` dla kompilatora Borland C++ 5.5).

Algorytm *K-M-P* 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 *K-M-P* 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 pozwalała na cofnięcie się w tej czynności (`i` w odczytywanym na bieżąco pliku).

Algorytm Boyera i Moore’a

Kolejny algorytm, który będziemy omawiali, jest ideowo znacznie prostszy do zrozumienia niż algorytm *K-M-P*. W przeciwieństwie do metody *K-M-P* porównywaniu ulega ostatni znak wzorca. To niekonwencjonalne podejście niesie ze sobą kilka istotnych zalet:

- ♦ Jeśli podczas porównywania okaże się, że rozpatrywany aktualnie znak nie wchodzi w ogóle w skład wzorca, wówczas możemy „skoczyć” w analizie tekstu o całą długość wzorca do przodu! Ciężar algorytmu przesunął się więc 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 *K-M-P*!

Zanim przejdziemy do szczegółowej prezentacji kodu, omówimy sobie na przykładzie jego działanie. Spójrzmy w tym celu na rysunek 8.7, gdzie przedstawione jest poszukiwanie ciągu znaków „*lek*” w tekście „*Z pamiętnika młodej lekarki*”¹¹.

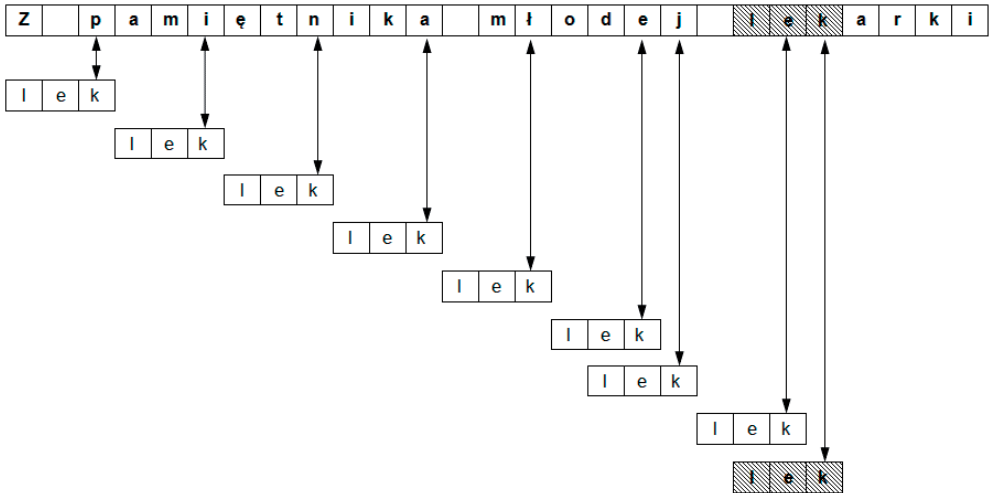
Pierwsze pięć porównań trafia na litery: `p`, `i`, `n`, `a` i `ł`, które we wzorcu nie występują! Za każdym razem możemy zatem przeskoczyć w tekście o trzy znaki do przodu (długość wzorca). Porównanie szóste trafia jednak na literę `e`, która w słowie „*lek*” występuje. Algorytm wówczas przesuwa wzorec 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 3 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 przypadku metody poprzedniej, także i tu musimy wykonać pewną prekompilację w celu stworzenia 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.

¹⁰ W przypadku kompilatorów popularnej serii Borland C++ należy skompilować program ręcznie poprzez polecenie `bcc32 -S -lxxx plik.cpp`, gdzie `xxx` oznacza katalog z plikami typu `H`; identyczna opcja istnieje w kompilatorze GNU C++, należy wystukać: `c++ -S plik.cpp`.

¹¹ Tytuł znakomitego cyklu autorstwa Ewy Szumańskiej.



Rysunek 8.7. Przeszukiwanie tekstu metodą Boyera i Moore'a

Będziemy również potrzebowali prostej funkcji indeks, która zwraca w przypadku spacji liczbę zero — w pozostałych przypadkach numer litery w alfabecie. Poniższy przykład uwzględnia jedynie kilka polskich liter — Czytelnik uzupełni go z łatwością 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. Jedna z możliwych wersji funkcji indeks jest przedstawiona poniżej:



bm.cpp

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

Funkcja indeks ma jedynie charakter usługowy. Służy ona m.in. do właściwej inicjalizacji tablicy przesunięć. Mając za sobą analizę przykładu z rysunku 8.7, Czytelnik nie powinien być zbyt zszokowany sposobem inicjalizacji:

```
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;
}

```

Przejdźmy wreszcie do prezentacji samego 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;
}

```

Algorytm *Boyer'a i Moore'a*, podobnie jak i *K-M-P*, jest klasy $M+N$ — jednak jest on o tyle od niego lepszy, iż w przypadku krótkich wzorców i długiego alfabetu kończy się po około M/N porównaniach. W celu obliczenia optymalnych przesunięć¹² autorzy algorytmu proponują skombinowanie powyższego algorytmu z tym zaproponowanym przez Knutha, Morrisa i Pratta. Celowość tego zabiegu wydaje się jednak wątpliwa, gdyż optymalizując sam algorytm, można w bardzo łatwy sposób uczynić zbyt czasochłonnym sam proces prekompilacji wzorca.

Algorytm Rabina i Karpa

Ostatni algorytm do przeszukiwania tekstów, który będziemy analizowali, wymaga znajomości rozdziału 7. i terminologii, która została w nim przedstawiona.

Algorytm *Rabina i Karpa* polega bowiem na dość przewrotnej idei:

- ♦ Wzorec w (do odszukania) jest *kluczem* (patrz terminologia transformacji kluczowej w rozdziale 7.) o długości M znaków, charakteryzującym się pewną wartością wybraną 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 w taki sposób odczytywany, aby na bieżąco znać M ostatnich znaków¹³. 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 ? Spodziewamy Czytelnik ma jednak prawo pokręcić w tym miejscu z powątpiewaniem głową: przecież to nie ma prawa działać szybko! Istotnie pomysł wyliczenia dodatkowo funkcji H dla

¹² Rozważ np. wielokrotne występowanie takich samych liter we wzorcu.

¹³ Na samym początku będzie to oczywiście M pierwszych znaków tekstu.

każdego słowa wejściowego o długości M wydaje się tak samo kosztowny — jeśli nie bardziej! — jak zwykle sprawdzanie tekstu znak po znaku (np. stosując algorytm typu *brute-force*). Tym bardziej że jak do tej pory nie powiedzieliśmy ani słowa na temat funkcji $H!$ Z poprzedniego rozdziału pamiętamy zapewne, iż jej wybór wcale nie był taki oczywisty.

Omawiany algorytm jednak istnieje i na dodatek działa szybko! Zatem, aby to wszystko, co poprzednio zostało napisane, logicznie się ze sobą łączyło, potrzebny nam będzie zapewne 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 dokonanych krok wcześniej, co znacząco potrafi 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:

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

Przesuniemy 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 przyjrzymy się x i x' , to okaże się, że wartość x' jest w dużej części zbudowana z elementów tworzących x — pomnożonych przez b z uwagi na przesunięcie. Nietrudno jest wówczas wywnioskować, że:

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

Jako funkcji H użyjemy dobrze nam znanej z poprzedniego rozdziału $H(x) = x \% p$, gdzie p jest dużą liczbą pierwszą. Założmy, że dla danej pozycji i wartość $H(x)$ jest nam znana. Po przesunięciu się w tekście o jedną pozycję w prawo pojawia się konieczność wyliczenia wartości funkcji $H(x')$ dla tego „nowego” słowa. Czy istotnie zachodzi potrzeba powtarzania całego wyliczenia? Być może istnieje pewne ułatwienie bazujące na zależności, jaka istnieje pomiędzy x i x' ?

Na pomoc przychodzi nam tu własność funkcji *modulo* użytej w wyrażeniu arytmetycznym. Można oczywiście obliczyć *modulo* z wyniku końcowego, lecz to bywa czasami niewygodne na przykład 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 z kalkulatorem. Identyczny rezultat da nam jednak następująca sekwencja obliczeń:

$$(5 \cdot 100) \% 7 = 3 \quad (3 + 6 \cdot 10) \% 7 = 0 \quad (0 + 8) \% 7 = 1.$$

co jest też łatwe do weryfikacji.

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



rk.cpp

```
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 nam już doskonałej funkcji `indeks` (patrz strona 186). 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 dokonywane w trzeciej pętli `for`. Dodatkowego wyjaśnienia wymaga być może linia oznaczona (*). Otóż dodawanie wartości $b*p$ do H_t pozwala nam 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 należą się parametrom p i b . Zaleca się, aby p było dużą liczbą pierwszą¹⁴, 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 . Ta możliwość — mimo iż mało prawdopodobna — ciągle istnieje i ostrożny programista powinien wykonać dodatkowy test zgodności $w[i] \ t[i] \dots \ t[i+M-1]$ po zwróceniu przez funkcję `rk` pewnego indeksu i .

Co zaś się tyczy wyboru podstawy systemu (oznaczonej w programie jako b), to warto jest 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 zwykle 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* i *Karpa* wykonuje się w czasie proporcjonalnym do $M+N$.

¹⁴ W naszym przypadku jest to liczba 33 554 393.