Piotr **Wróblewski**

ALGORYTMY

struktury danych i techniki programowania dla programistów Java

Helion

Redaktor prowadzący: Małgorzata Kulik

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

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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

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

ISBN: 978-83-283-5465-4

Printed in Poland.

Spis treści

Rozdział 13.

Przeszukiwanie tekstów

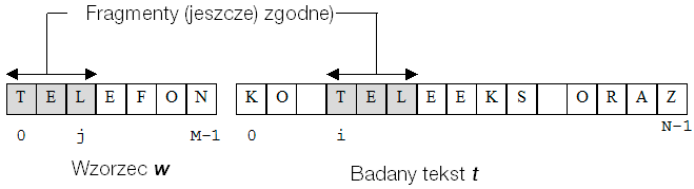
Przeszukiwanie tekstów traktuje się jako odrębną dziedzinę z uwagi na szeroką gamę zastosowań praktycznych. Tekst jest tutaj definiowany jako ciąg znaków w sensie informatycznym (nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną”!) i może być ciągiem bitów, który oczywiście można interpretować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

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

Algorytm typu brute force

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

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



Rysunek 13.1. *Algorytm typu brute force przeszukiwania tekstu*

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

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

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

SzukajTxt.java

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

Wynik uruchomienia programu jest następujący:

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

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

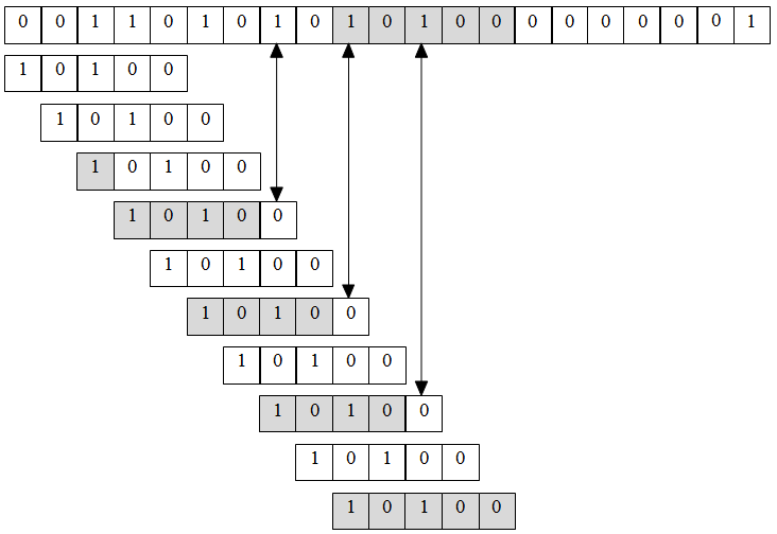
> W Javie możesz ewentualnie użyć obiektów klasy `String` zamiast tablic znaków, ale ponieważ w tej klasie znajdują się już gotowe metody pozwalające wyszukiwać wzorce, realizacja algorytmu wyszukiwania byłaby nadmiarowa — w ostatniej linijce można by użyć instrukcji `tS.indexOf(w2S)` zamiast `szukaj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

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

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

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

Rysunek 13.2. *„Fałszywe starty" podczas poszukiwania*



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

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

[1] Zwykle `M` będzie znacznie mniejsze niż `N`.

Nowe algorytmy poszukiwań

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

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

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

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

Algorytm KMP

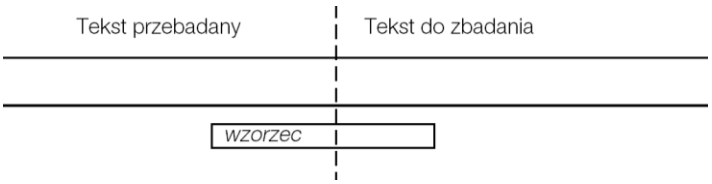
Wadą algorytmu *brute force* jest jego uwrażliwienie na konfigurację danych: fałszywe restarty są tu bardzo kosztowne; w analizie tekstu cofamy się o całą długość wzorca, zapominając po drodze o wszystkim, co przetestowaliśmy do tej pory. Narzuca się tu niejako chęć skorzystania z informacji, które już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

W pewnych szczególnych przypadkach przy znajomości struktury analizowanego tekstu możliwe jest ulepszenie algorytmu. Jeśli przykładowo wiemy na pewno, że w poszukiwanym wzorcu pierwszy znak w ogóle się nie pojawia[2], to w razie restartu nie musimy cofać wskaźnika `i` o `j-1` pozycji, jak to było poprzednio (patrz listing *SzukajTxt.java*). W tym przypadku możemy po prostu inkrementować `i`, wiedząc, że ewentualne powtórzenie poszukiwań na pewno nic by nie dało. Owszem, można się łatwo zgodzić z twierdzeniem, że tak wyspecjalizowane teksty zdarzają się relatywnie rzadko, jednak powyższy przykład ukazuje, iż ewentualne manipulacje algorytmami poszukiwań są ciągle możliwe — wystarczy się tylko rozejrzeć. Idea algorytmu KMP polega na wstępnym zbadaniu wzorca w celu obliczenia liczby pozycji, o które należy cofnąć wskaźnik `i` w przypadku stwierdzenia niezgodności

[2] Przykład: „ABBBBBBB" — znak „A" wystąpił tylko jeden raz.

badanego tekstu ze wzorcem. Oczywiście można również rozumować w kategoriach przesuwania wzorca do przodu — rezultat będzie ten sam. To właśnie tę drugą konwencję będziemy stosować dalej. Wiemy już, że powinniśmy przesuwać się po badanym tekście nieco inteligentniej niż w poprzednim algorytmie. W przypadku zauważenia niezgodności na pewnej pozycji `j` wzorca[3] 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 13.3.

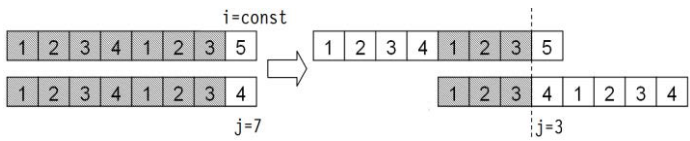


Rysunek 13.3. *Wyszukiwanie optymalnego przesunięcia w algorytmie KMP*

Moment niezgodności został zaznaczony poprzez narysowanie przerywanej pionowej kreski. Otóż wyobraźmy sobie, że przesuwamy teraz 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 13.4 możemy dostrzec, że na siódmej pozycji wzorca[4] (którym jest dość abstrakcyjny ciąg `12341234`) została stwierdzona niezgodność.



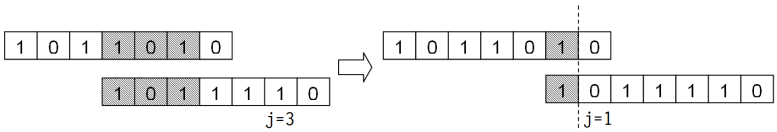
Rysunek 13.4. *Przesuwanie się wzorca w algorytmie KMP (1)*

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

Analogiczny przykład znajduje się na rysunku 13.5.

[3] Lub *i* w przypadku badanego tekstu.

[4] Licząc indeksy tablicy tradycyjnie od zera.



Rysunek 13.5. *Przesuwanie się wzorca w algorytmie KMP (2)*

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

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

Listing

KMP.java (fragment)

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

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

Wynik uruchomienia programu jest następujący:

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

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

powodu chcemy, aby indeks `j` pozostał niezmieniony, 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`.

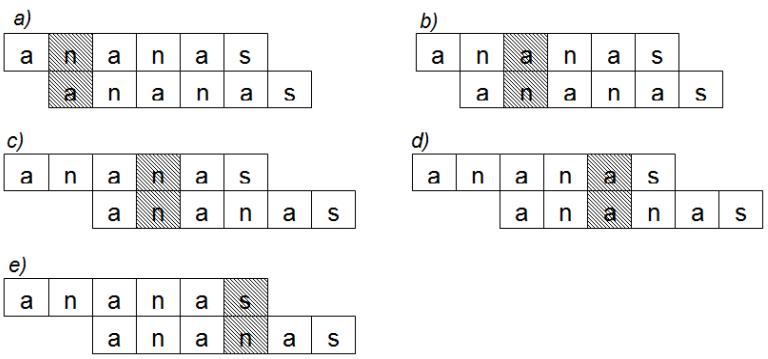
Pozostaje do omówienia sposób konstrukcji tablicy `shift[M]`. Jej obliczenie powinno nastąpić przed wywołaniem funkcji `kmp`, co sugeruje, że w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona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

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

Sens tego algorytmu jest następujący: tuż po inkrementacji `i` oraz `j` wiemy, że pierwsze `j` znaków wzorca jest zgodne ze znakami na pozycjach `p[i-j-1] ... 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 13.6). Zacieniowane 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 zacieniowaną (porównaj z rysunkiem 13.5).

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



Rysunek 13.6. *Optymalne przesunięcia wzorca „ananas" w algorytmie KMP*

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

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

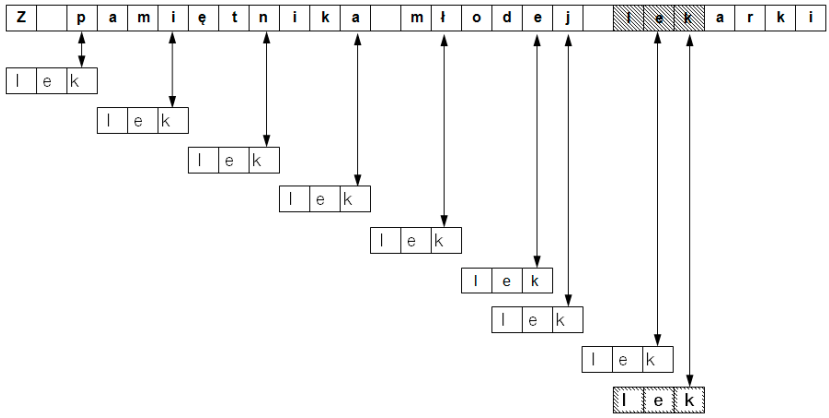
cofania się w nim. W niektórych systemach może to mieć istotne znaczenie praktyczne — przykładowo mamy zamiar analizować bardzo długi plik tekstowy i charakter wykonywanych operacji nie pozwala na cofnięcie się w tej czynności (i w odczytywanym na bieżąco pliku).

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 13.7, gdzie przedstawione zostało poszukiwanie ciągu znaków „lek" w tekście „Z pamiętnika młodej lekarki"[5].



Rysunek 13.7. *Przeszukiwanie tekstu metodą Boyera-Moore'a*

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 wzorzec 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 dysponować tablicą pomocniczą zapisującą stan bieżących niezgodności. Tablica ta powinna mieć tyle pozycji, ile jest znaków w alfabecie.

[5] Tytuł kultowego cyklu skeczy radiowych autorstwa Ewy Szumańskiej (1921 – 2011).

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

BM.java (fragment)

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

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

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

Teraz przejdziemy do prezentacji listingu algorytmu z przykładem wywołania:

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

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

 int przes = 0; // aktualne przesunięcie wzorca 'w' względem tekstu
  while(przes <= (N -M)) { // sytuacja, gdy wzorzec będzie znajdował się na samym końcu
     int j = M-1;
  // przesuwanie indeksu 'j' (pozycja dopasowania), gdy porównywane znaki są równe:
  while(j >= 0 && w.charAt(j) == t.charAt(przes+j))
     j--;
 // przesuwanie wzorca tak, aby kolejny znak zrównał się z ostatnim swoim wystąpieniem we wzorcu:
  if (j < 0) {
      System.out.println("Znalazłem dopasowanie na pozycji: " + przes);
      przes += (przes+M < N)? M-shift[t.charAt(przes+M)] : 1;
  } else // przesuwanie do ostatniej niezgodności z uwzględnieniem możliwego końca tekstu
         przes += max(1, j - shift[t.charAt(przes+j)]);
}
  }
 public static void main(String []args) {
  String t = "Z pamietnika mlodej lekarki podajacej w tym momencie pacjentowi nowe
  ↳ lekarstwo";
  String w = "lek";
  System.out.println(t);
  bm(t, w); // program znajdzie dopasowanie na pozycjach: 20 i 69
 }
}
```

Algorytm Boyera-Moore'a, podobnie jak KMP, jest klasy M+N, jednak jest o tyle od niego lepszy, że w przypadku krótkich wzorców i długiego alfabetu kończy się po ok. M/N porównaniach. W celu obliczenia optymalnych przesunięć[6] autorzy algorytmu proponują połączenie powyższego algorytmu z zaproponowanym przez Knutha, Morrisa i Pratta. Celowość tego zabiegu wydaje się jednak wątpliwa, gdyż optymalizując sam algorytm, można bardzo łatwo sprawić, że proces prekompilacji wzorca stanie się zbyt czasochłonny.

Algorytm Rabina-Karpa

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

Algorytm Rabina-Karpa polega na dość przewrotnej idei:

- Wzorzec 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 znaków[7]. 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 wyliczenia dodatkowo funkcji H dla każdego słowa wejściowego o długości M wydaje się tak samo kosztowny — jeśli nie bardziej — jak zwykłe sprawdzanie tekstu znak po znaku (np. stosując algorytm siłowy typu *brute force*). Tym bardziej, że do tej pory nie powiedziałem ani słowa na temat funkcji H! W rozdziale 10. mogliśmy się przekonać, że jej wybór wcale nie jest taki oczywisty.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

co jest też łatwe do weryfikacji.

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

RK.java

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

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

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

```
      if (Ht < 0) // korekta znaku (**)
        Ht = (Ht + p);
    }
   }
}
public static void main(String[] args) {
 String t ="Z pamietnika mlodej lekarki podajacej w tym momencie pacjentowi nowe
 ↳ lekarstwo";
 String w = "lek";
 System.out.println(t);
 System.out.println("Szukam: " + w);
 rk(w, t);
}
}
```

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

Kolejne uwagi dotyczą parametrów `p` i `MAX`. Zaleca się, aby `p` było dużą liczbą pierwszą[8], jednakże nie można tu przesadzać z uwagi na możliwe przekroczenie zakresu pojemności użytych zmiennych. W przypadku wyboru dużego `p` zmniejszamy prawdopodobieństwo wystąpienia kolizji spowodowanej niejednoznacznością funkcji `H`. Możliwość ta, mimo że mało prawdopodobna, ciągle istnieje i dlatego po upewnieniu się, że `Hw == Ht` sprawdzamy już klasycznie (znak po znaku), czy wyszukanie się powiodło — linia oznaczona *(**)*.

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

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

Wynik uruchomienia programu jest następujący:

```
Z pamietnika mlodej lekarki podajacej w tym momencie pacjentowi nowe lekarstwo
Szukam: lek
Znalazłem wzorzec na pozycji: 20
Znalazłem wzorzec na pozycji: 69
```

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

[8] W naszym przypadku jest to liczba 33 554 393.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

R

S

T

U

V

W

X

Z


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA Helion

Opanuj Javę jak prawdziwy profesjonalista!

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

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

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

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

Rozwiązuj problemy programistyczne w Javie!

Helion

helion.pl

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

ISBN 978-83-283-5465-4

9 788328 354654

Cena: 67,00 zł