Piotr **Wróblewski**

Algorytmy, struktury danych i techniki programowania

Wydanie VI

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

ISBN: 978-83-283-5374-9

Printed in Poland.

Spis treści

Rozdział 9.

Przeszukiwanie tekstów

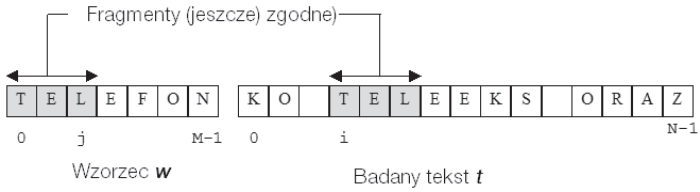
Przeszukiwanie tekstów traktuje się jako odrębną dziedzinę z uwagi na szeroką gamę zastosowań praktycznych. Tekst jest tutaj definiowany jako ciąg znaków w sensie informatycznym (nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną”!) i może być ciągiem bitów, który oczywiście można interpretować 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 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 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 procedury[1]:

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)`.

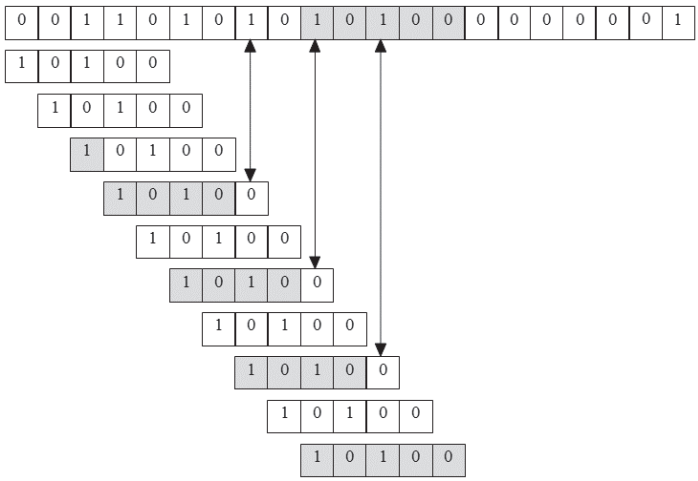
Jako wynik funkcji zwracana jest pozycja w tekście, od której zaczyna się wzorzec, lub `-1`, gdy poszukiwany tekst nie został odnaleziony — to znana już doskonale konwencja.

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 jedynką (umawiamy się, że zera i jedynki 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`, `rfing`), 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·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.

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.

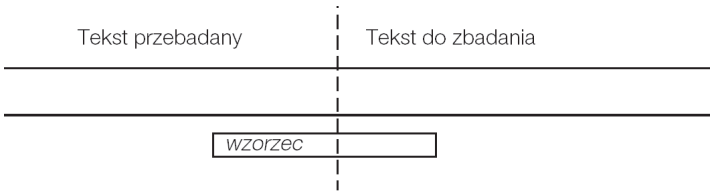
W następnych trzech punktach szczegółowo omówię algorytmy wspomniane w tym historycznym przeglądzie.

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 pojawia[3], 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` wzorca[4] 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.

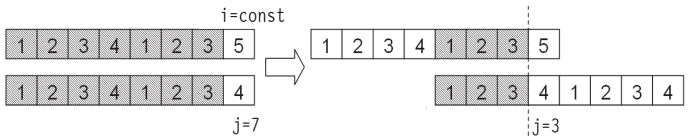
[4] Lub *i* w przypadku badanego tekstu.

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 wzorca[5] (którym jest dość abstrakcyjny ciąg 12341234) została stwierdzona niezgodność.

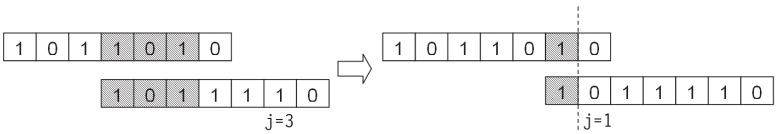
Rysunek 9.4. *Przesuwanie się wzorca w algorytmie KMP (1)*



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.

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 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`.

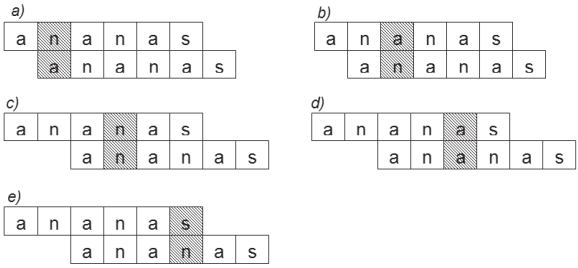
Pozostaje do omówienia sposób konstrukcji tablicy `shift[M]`. Jej obliczenie powinno nastąpić przed wywołaniem funkcji `kmp`, co sugeruje, że w przypadku wielokrotnego poszukiwania tego samego wzorca nie musimy już powtarzać inicjacji tej tablicy. Funkcja inicjująca tablicę jest przewrotna — jest ona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

```
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] ... 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). 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 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:

Listing

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 asemblerowym, jest zatem bardzo szybka. Posiadacze kompilatorów, które umożliwiają generację kodu wynikowego jako tzw. „assembly output"[6], mogą z łatwością sprawdzić różnice

[6] Aby wygenerować kod asemblera 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.

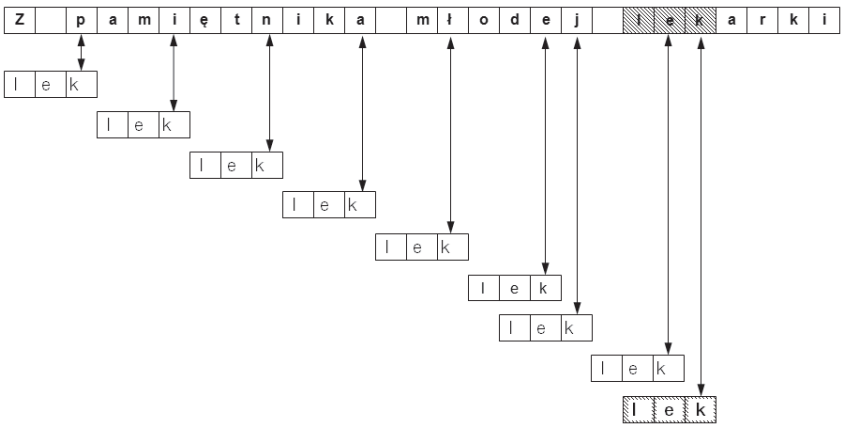
Użycie tego algorytmu jest jednak niezbędne w tych aplikacjach, w których następuje liniowe przeglądanie tekstu — bez buforowania. Jak łatwo zauważyć, wskaźnik `i` w funkcji `kmp` nigdy nie jest dekrementowany, co oznacza, że plik można przeglądać od początku do końca bez cofania się w nim. W niektórych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


Listing

bm.cpp (fragment)

```
#include <iostream>
#include <string.h> // z powodu strlen()
using namespace std;
int kmp_ananas(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 powinieneś być zbytnio 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 przesunięć[8] 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:

- 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[9]. 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 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`! 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.

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:

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

Jako funkcji `H` użyjemy dobrze znanej z poprzedniego rozdziału `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:

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

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

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

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 pierwszą[10], 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 i 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

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

T

U

V

W

X

Z

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

ISBN 978-83-283-5374-9

9 788328 353749

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł