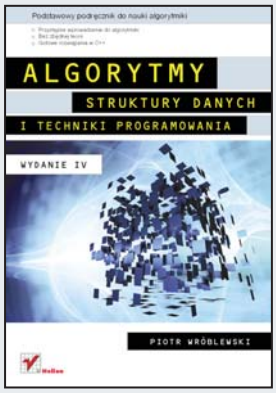
Algorytmy, struktury danych i techniki programowania. Wydanie IV



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

Rozdział 8.

Przeszukiwanie tekstów

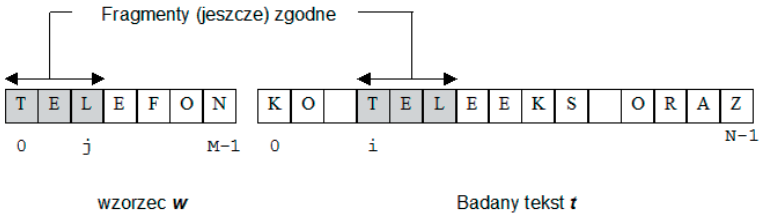
Zanim na dobre zanurzymy się w lekturę nowego rozdziału, należy wyjaśnić pewne nieporozumienie, które może towarzyszyć jego tytułowi. Otóż za *tekst* będziemy uważali ciąg znaków w sensie informatycznym. Nie zawsze będzie to miało cokolwiek wspólnego z ludzką „pisaniną"! Tekstem będzie na przykład również ciąg bitów[1], który tylko przez umowność może być podzielony na równej wielkości porcje, którym przyporządkowano pewien kod liczbowy[2].

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

Algorytm typu brute-force

Zadaniem, które będziemy usiłowali wspólnie rozwiązać, jest poszukiwanie wzorca[3] *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 zrównać się 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:

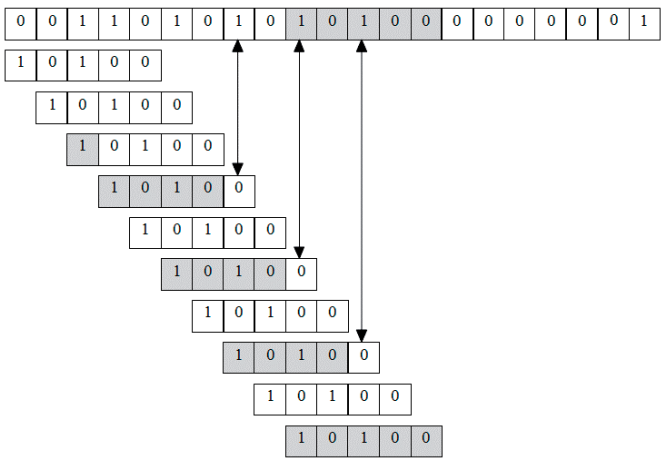
txt-1.cpp

```
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 wzorzec, 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. *„Fałszywe 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 wzorzec, dla których proces poszukiwania będzie trwał możliwie najdłużej. Chodzi oczywiście zarówno o tekst, jak i wzorzec złożone z samych zer i zakończone jedynką[4].

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)-(M-2)=N-M+1` razy, i wiedząc, że podczas każdego cyklu jest konieczne wykonanie `M` porównań, otrzymujemy natychmiast `M(N-M+1)`, czyli około[5] `M·N`.

Zaprezentowany w tym paragrafie algorytm wykorzystuje komputer jako bezmyślne, ale sprawne liczydło[6]. 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 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 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 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 jedynki symbolizują tu dwa różne od siebie znaki.

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

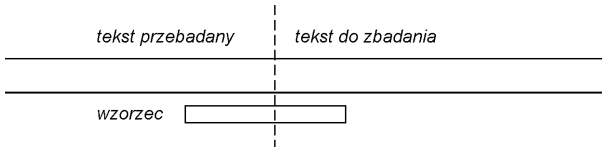
[6] Termin *brute-force* jeden z moich znajomych ślicznie przetłumaczył jako „metodę mastodonta".

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 ogóle[7], 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` wzorca[8] 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 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 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 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.

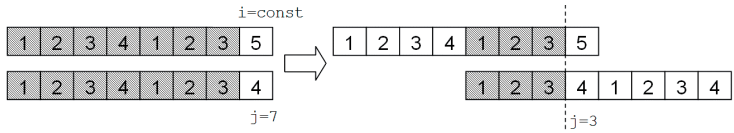
Od czego zależy ewentualne pokrycie się oglądanych fragmentów tekstu i wzorca? Otóż dość paradoksalnie badany tekst nie ma tu nic do powiedzenia — jeśli można to tak określić. Informacja o tym, jaki on był, jest ukryta w stwierdzeniu „`j-1` znaków było zgodnych" — w tym sensie można zupełnie o badanym tekście zapomnieć i analizując wyłącznie sam wzorzec, 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 wzorca[9] (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.

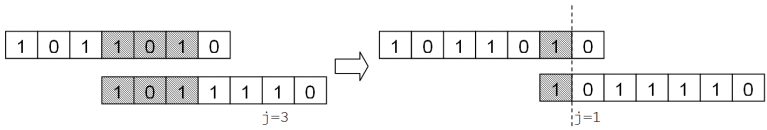
[9] Licząc indeksy tablicy tradycyjnie od zera.



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 przesunięć `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*:

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;
}
```

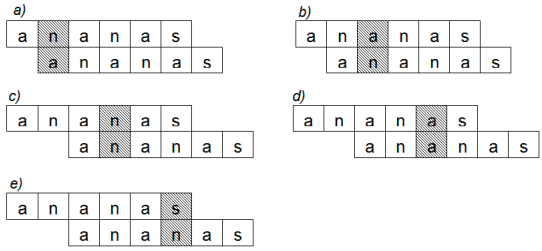
Szczególnym przypadkiem jest wystąpienie niezgodności na pozycji zerowej: z założenia niemożliwe jest tu przesuwanie wzorca w celu uzyskania nałożenia się znaków. Z tego powodu chcemy, aby indeks `j` pozostał niezmieniony przy jednoczesnej progresji indeksu `i`. Jest to możliwe do uzyskania, jeśli umówimy się, że `shift[0]` zostanie zainicjowany wartością `-1`. Wówczas podczas kolejnej iteracji pętli `for` nastąpi inkrementacja `i` oraz `j`, co wyzeruje 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]... 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). 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 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 zwięzłym 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 dyskietce 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 pozwala 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].

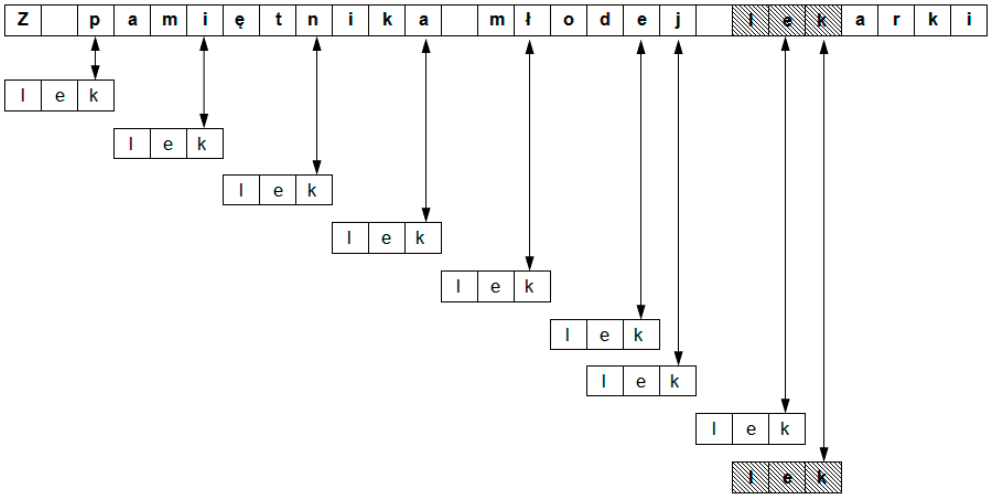
Pierwsze pięć porównań trafia na litery: `p`, `i`, `n`, `a` i `ł`, które we wzorcu nie występują! Za każdym razem możemy zatem przeskoczyć w tekście o trzy znaki do przodu (długość wzorca). Porównanie szóste trafia jednak na literę `e`, która w słowie „lek” występuje. Algorytm wówczas przesuwa wzorzec 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 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 -Ixxx 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:

Listing

bm.cpp

```
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))
         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 zbytnio zdziwiony 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 *Boyera* 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 przesunięć[12] 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:

- Wzorzec `w` (do odszukania) jest *kluczem* (patrz terminologia transformacji kluczowej w rozdziale 7.) 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 w taki sposób odczytywany, aby na bieżąco znać `M` ostatnich znaków[13]. 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? Spostrzegawczy 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 zwykłe 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.

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

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

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

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

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. Załóżmy, 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:

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 `Ht` i `Hw`. Ponieważ ciągi znaków trzeba interpretować jako liczby, konieczne będzie zastosowanie znanej nam już doskonale funkcji `indeks` (patrz strona 186). Wartość `Hw` jest niezmienna i nie wymaga uaktualniania. 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` — co jest dokonywane w trzeciej pętli `for`. Dodatkowego wyjaśnienia wymaga być może linia oznaczona (*). Otóż dodawanie wartości `b*p` do `Ht` pozwala nam 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[14], 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]... 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 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* i *Karpa* wykonuje się w czasie proporcjonalnym do `M+N`.

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