Wykład 6

Wyszukiwanie wzorca w tekście

Wyszukiwanie wzorca (przegląd)

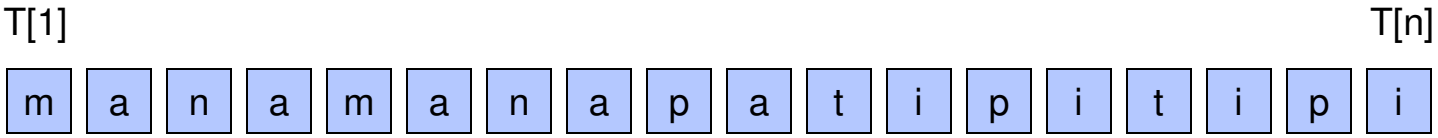
- **Porównywanie łańcuchów**
- **Algorytm podstawowy „siłowy” (naive algorithm)**
  - Jak go zrealizować?
- **Algorytm Rabina-Karpa**
  - Inteligentne wykorzystanie własności liczb pierwszych i teorii liczb
- **Algorytm Knutha-Morrisa-Pratta**
  - Wykorzystuje deterministyczny automat skończony
  - Optymalny
- **Algorytm Boyera-Moore’a**
  - Niezgodne (niepasujące) znaki pozwalają na przeskakiwanie fragmentów tekstu
  - W praktyce (średnio) okazuje się nawet lepszy od KMP

Literatura

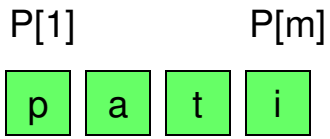
- Cormen, Leiserson, Rivest, “Wprowadzenie do algorytmów”, WNT, 1999
  rozdział 34

Porównywanie łańcuchów

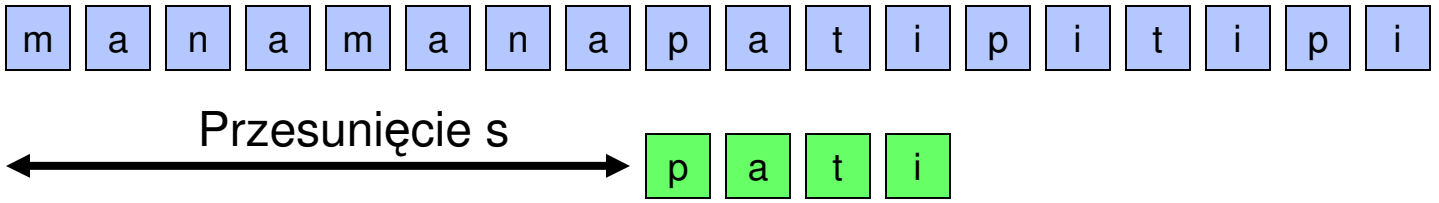
- **Wejście**
  - tekst T o długości n ze skończonego alfabetu Σ:



  - wzorzec P o długości m ze skończonego Σ:



- **Wyjście**
  - Wszystkie wystąpienia P w T T[s+1..s+m] = P[1..m]



Algorytm „siłowy”

Naive-String-Matcher(T,P)

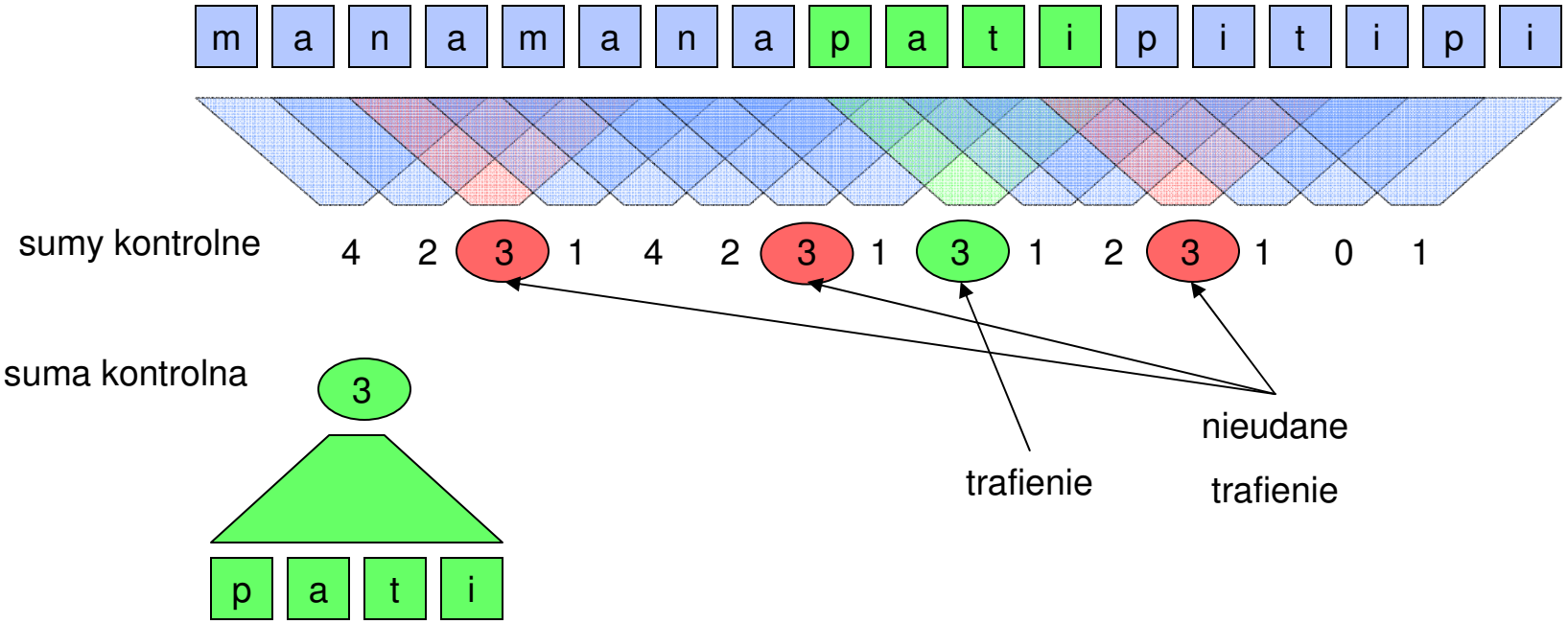
1. **n ← length(T)**
2. **m ← length(P)**
3. **for s ← 0 to n-m do**
4. **if P[1..m] = T[s+1 .. s+m] then**
5. **return “Wzorzec występuje z przesunięciem” s**

uwagi:

- **czas wykonania tego algorytmu (w pesymistycznym przypadku) wynosi O((n-m+1) m)**
- **dla n = 2m daje to $O(n^2)$**
- **Algorytm ten nie jest optymalny – zadanie to można rozwiązać w czasie O(m + n)**

Algorytm Rabina-Karpa

- **Pomysł: policzyć**
  - Sumę kontrolną dla wzorca P oraz
  - Sumę kontrolną dla każdego podłańcuch w T o długości m



Algorytm Rabina-Karpa

- **Obliczanie sumy kontrolnej:**
  - Wybieramy liczbę pierwszą q
  - niech d = |Σ|

$$S_m(P) = \sum_{i=1}^{m} d^{m-i} P[i] \bmod q = P[m] + dP[m-1] + \ldots + d^{m-2} P[2] + d^{m-1} P[1] \bmod q$$

- **Przykład:**
  - Σ={0,1,2,3,4,5,6,7,8,9}
  - wtedy d = 10, q = 13
  - niech P = 0815

$S_4(0815) = (0 \cdot 1000 + 8 \cdot 100 + 1 \cdot 10 + 5 \cdot 1) \mod 13 = 815 \mod 13 = 9$

Metoda szybkiego obliczania sumy kontrolnej (schemat Hornera)

- **obliczamy**

$$S_m(P) = \sum_{i=1}^{m} d^{m-i} P[i] \bmod q$$

- **wykorzystując**

$$S_m(P) \equiv \sum_{i=1}^{m} d^{m-i} P[i] \equiv d \left(\sum_{i=1}^{m-1} d^{m-i-1} P[i] \right) + P[m] \equiv d S_{m-1}(P[1..m-1]) + P[m] \pmod{q}$$

- **przykład:**
  - Σ={0,1,2,3,4,5,6,7,8,9}
  - wtedy d = 10, q = 13
  - niech P = 0815

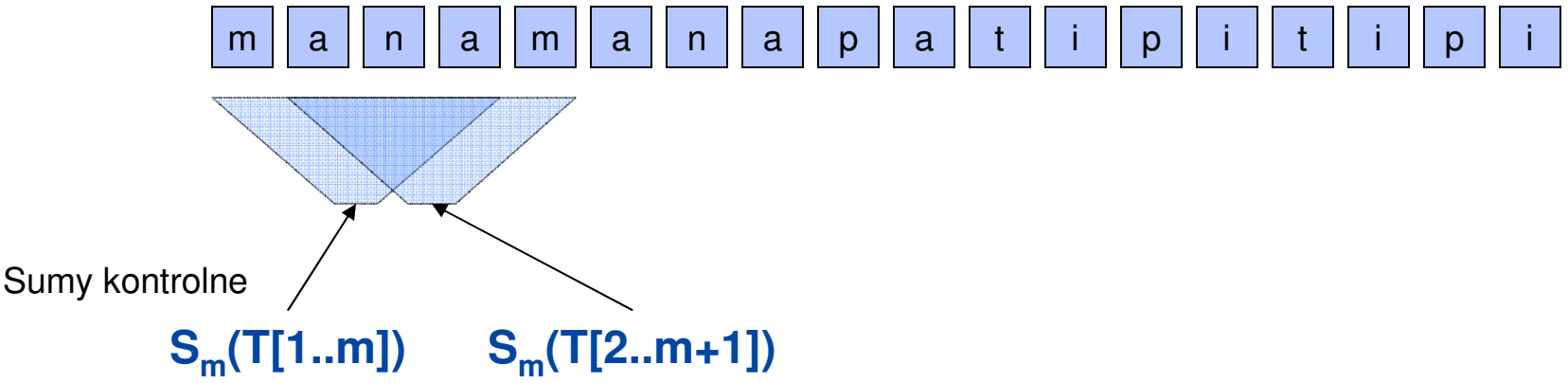
$S_4(0815)$ = ((((**0**·10+**8**)·10)+**1**)·10)+**5** mod 13 =

((((**8** ·10)+**1**)·10)+**5** mod 13 =

(3 ·10)+**5** mod 13 = 9

Jak szybko obliczać sumy kontrolne dla kolejnych podłańcuchów?

- **Zaczynamy od $S_m(T[1..m])$**



$$S_m(T[2..m+1]) \equiv d(S_m(T[1..m]) - d^{m-1}T[1]) + T[m+1] \pmod{q}$$

Algorytm Rabina-Karpa

Rabin-Karp-Matcher(T,P,d,q)

1. **n ← length(T)**
2. **m ← length(P)**
3. **h ← d^{m-1} mod q**
4. **p ← 0**
5. **t_0 ← 0**
6. **for i ← 1 to m do**
7. **p ← (d p + P[i]) mod q**
8. **t_0 ← (d t_0 + T[i]) mod q**
9. **for s ← 0 to n-m do**
10. **if p = t_s then**
11. **if P[1..m] = T[s+1..s+m] then return "wzorzec występuje z p." s**
12. **if s < n-m then**
13. **t_{s+1} ← (d(t_s-T[s+1]h) + T[s+m+1]) mod q**

Suma kontrolna dla wzorca P

Suma kontrolna dla T[1..m]

Jeśli sumy kontrolne są zgodne, przeprowadzamy porównywanie łańcuchów

Uaktualnianie sumy kontrolnej dla T[s+1..s+m] z wykorzystaniem T[s..s+m-1]

Złożoność obliczeniowa algorytmu Rabina-Karpa

- **W pesymistycznym przypadku O(m (n-m+1))**

- **Analiza probabilistyczna**
  - Prawdopodobieństwo nieudanego trafienia dla losowego wejścia wynosi 1/q
  - Oczekiwana ilość nieudanych trafień wynosi zatem O(n/q)

  - Stąd oczekiwany czas wykonania algorytmu Rabina-Karpa to
    O(n + m (v+n/q))
    gdzie v jest ilością wystąpień wzorca (trafień)

- **Jeśli wybierzemy q ≥ m i mamy stałą ilość wystąpień wzorca to czas wykonania algorytmu Rabina-Karpa wyniesie O(n +m).**

- **Jednak jeśli v i m są duże wtedy czas wykonania dla tego algorytmu może wynieść $O(n^2)$**

Metoda Knutha-Morrisa-Pratta (KMP)

Niech t_i i p_{j+1} będą porównywanymi znakami:

t_1	t_2	...	...	t_i	...	...
		=	= = =	≠		
p_1	...	p_j	p_{j+1}	...	p_m	

Jeśli pierwsza niezgodność nastąpiła dla znaków t_i i p_{j+1}, to:

- Ostatnie *j* znaków porównywanych w *T* jest zgodne z pierwszymi *j* znakami w *P*.
- $t_i \neq p_{j+1}$

Metoda Knutha-Morrisa-Pratta (KMP)

Pomysł:

określamy $j' = next[j] < j$ takie, że t_i może być porównywane z $p_{j'+1}$

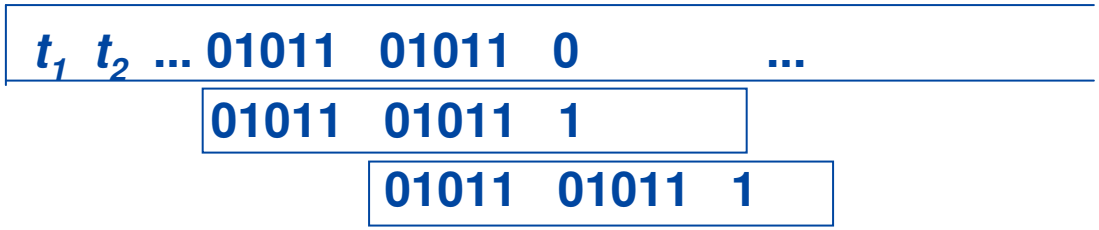
czyli: $j' < j$ takie, że $P_{1\ldots j'} = P_{j-j'+1\ldots j}$.

inaczej szukamy najdłuższego prefiksu P, który jest jednocześnie sufiksem $P_{1\ldots j}$.

t_1	t_2	...	...	t_i	...	...
		$=$	$=$ $=$ $=$	$\neq$		
p_1	...	p_j	p_{j+1}	...	p_m	

Metoda Knutha-Morrisa-Pratta (KMP)

Przykład wyznaczania *next[j]*:



***next[j]* = długość najdłuższego prefiksu P, będącego sufiksem $P_{1 \ldots j}$.**

Metoda Knutha-Morrisa-Pratta (KMP)

⇒ dla P = 0101101011, $next$ = [0,0,1,2,0,1,2,3,4,5] :

1	2	3	4	5	6	7	8	9	10
0	1	0	1	1	0	1	0	1	1
		0							
		0	1						
					0				
					0	1			
					0	1	0		
					0	1	0	1	
					0	1	0	1	1

Metoda Knutha-Morrisa-Pratta (KMP)

```
KMP := proc (T : : string, P : : string)
# Input: text T and pattern P
# Output: list L of shifts i at which P occurs in T
   n := length (T); m := length(P);
   L :=  [ ]; next := KMPnext(P);
   j  :=  0;
   for i from 1 to n do
            while j>0 and T[i] <> P[j+1] do j := next [j] od;
            if T[i]  = P[j+1]  then j := j+1 fi;
            if j = m then L := [L[ ] , i-m] ;
                              j  := next [j]
            fi;
    od;
    RETURN (L);
end;
```

Metoda Knutha-Morrisa-Pratta (KMP)

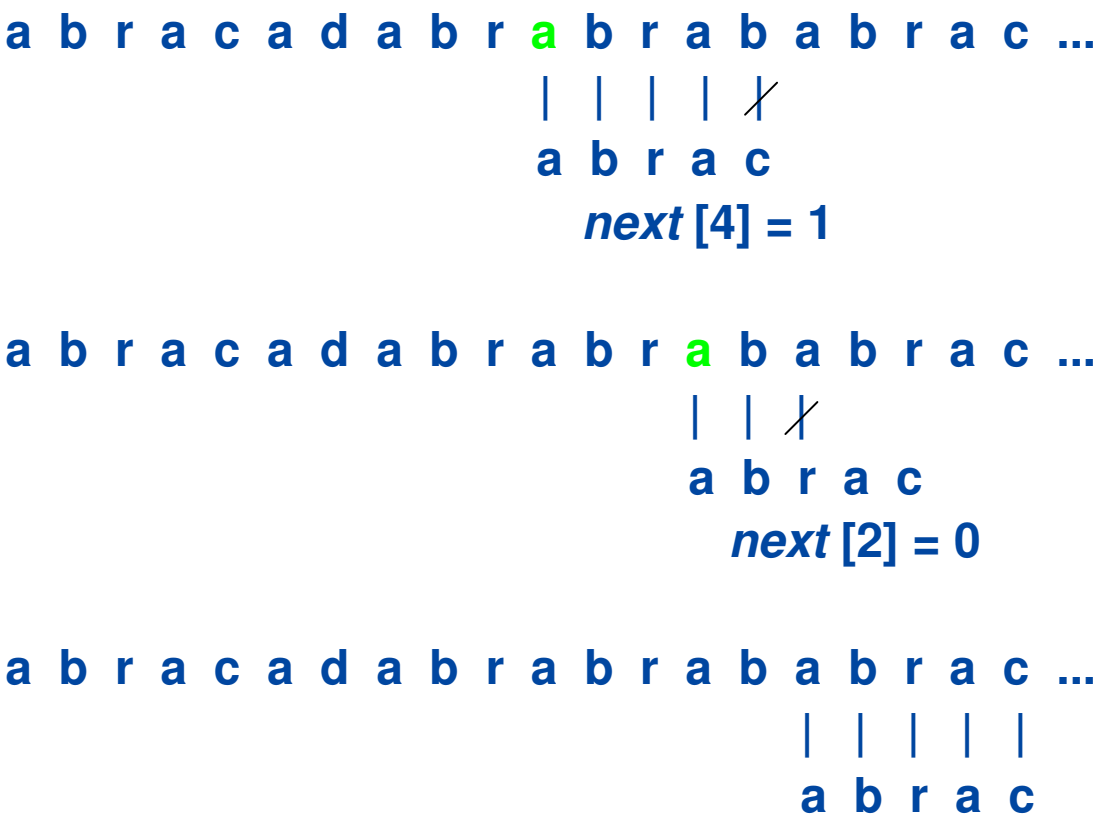
Wzorzec: abracadabra, *next* = [0,0,0,1,0,1,0,1,2,3,4]

a b r a c a d a b r a b r a b a b r a c ...
| | | | | | | | | | |
a b r a c a d a b r a

***next*[11] = 4**

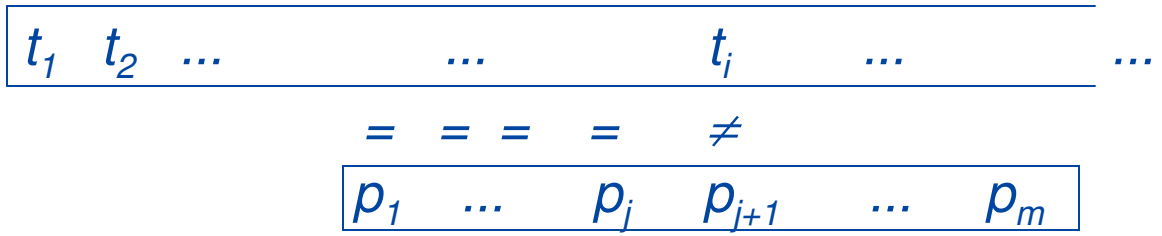
a b r a c a d a b r a b r a b a b r a c ...
- - - -
a b r a c ...
***next*[4] = 1**

Metoda Knutha-Morrisa-Pratta (KMP)



Metoda Knutha-Morrisa-Pratta (KMP)

Poprawność:



Przed pętlą *for* mamy:

$P_{1...j} = T_{i-j...i-1}$ **oraz** $j \neq m$

Jeśli $j = 0$: **jesteśmy przy pierwszym znaku w *P***

Jeśli $j \neq 0$: ***P* może zostać przesuwane dopóki** $j > 0$ **i** $t_i \neq p_{j+1}$

Metoda Knutha-Morrisa-Pratta (KMP)

Jeśli $T[i] = P[j+1]$, j oraz i mogą być zwiększone (na końcu pętli).

Jeśli porównywanie P zakończyło się ($j = m$), odnaleźliśmy wystąpienie wzorca, i możemy wykonać przesunięcie.

Metoda Knutha-Morrisa-Pratta (KMP)

Złożoność czasowa:

- **Wskaźnik *i* nigdy nie jest resetowany**
- **Wskaźniki *i* oraz *j* są zawsze zwiększane razem**
- **Zawsze: *next*[j] < *j*;**
  j może zostać zmniejszone tylko tyle razy ile razy zostało zwiększone.

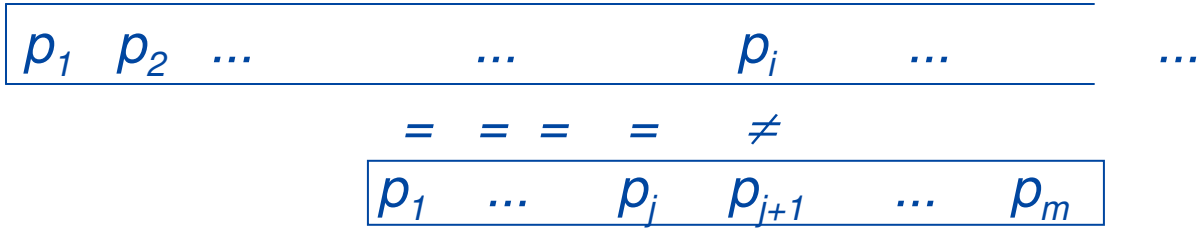
Stąd algorytm KMP można wykonać w czasie *O*(*n*), jeśli znamy tablicę *next*.

Obliczanie tablicy next

***next*[i] = długość najdłuższego prefiksu *P, będącego jednocześnie sufiksem $P_{1\cdots i}$.**

***next*[1] = 0**

Niech *next*[*i*-1] = *j*:



Obliczanie tablicy next

Rozważmy dwa przypadki:

1) $p_i = p_{j+1} \rightarrow$ ***next***[***i***] = ***j*** + 1

2) $p_i \neq p_{j+1} \rightarrow$ **zastępujemy *j* przez *next*[*j*] , aż** $p_i = p_{j+1}$ **lub** $j = 0$.
jeśli $p_i = p_{j+1}$**, to możemy przypisać *next*[*i*] = *j* + 1,**
w przeciwnym raze *next*[*i*] = 0.

Obliczanie tablicy next

```
KMPnext := proc (P : : string)
#Input    :  pattern P
#Output :   next-Array for P
   m := length (P);
   next := array (1..m);
   next [1] := 0;
   j := 0;
   for i from 2 to m do
      while j > 0 and P[i] <> P[j+1]
            do j := next [j] od;
      if P[i] = P[j+1] then j := j+1 fi;
      next [i] := j
   od;
   RETURN (next);
end;
```

Czas wykonania KMP

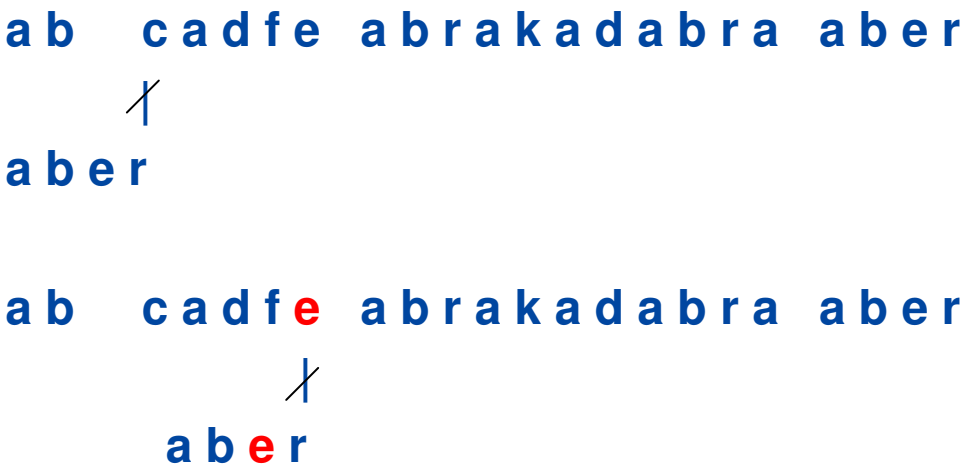
Czas obliczenia tablicy *next* wynosi O(*m*), stąd całkowity czas dla KMP to O(*n* + *m*).

Czy można to zrobić lepiej?

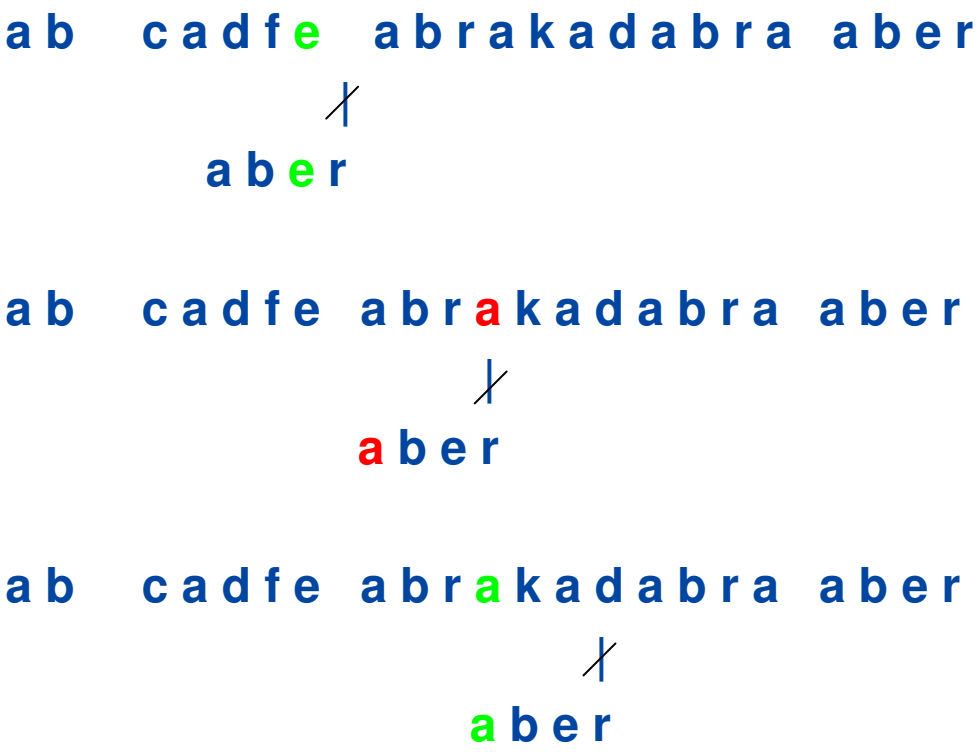
Metoda Boyera-Moore'a (BM)

Pomysł: **przykładamy wzorzec od lewej do prawej strony, ale porównujemy od prawej do lewej.**

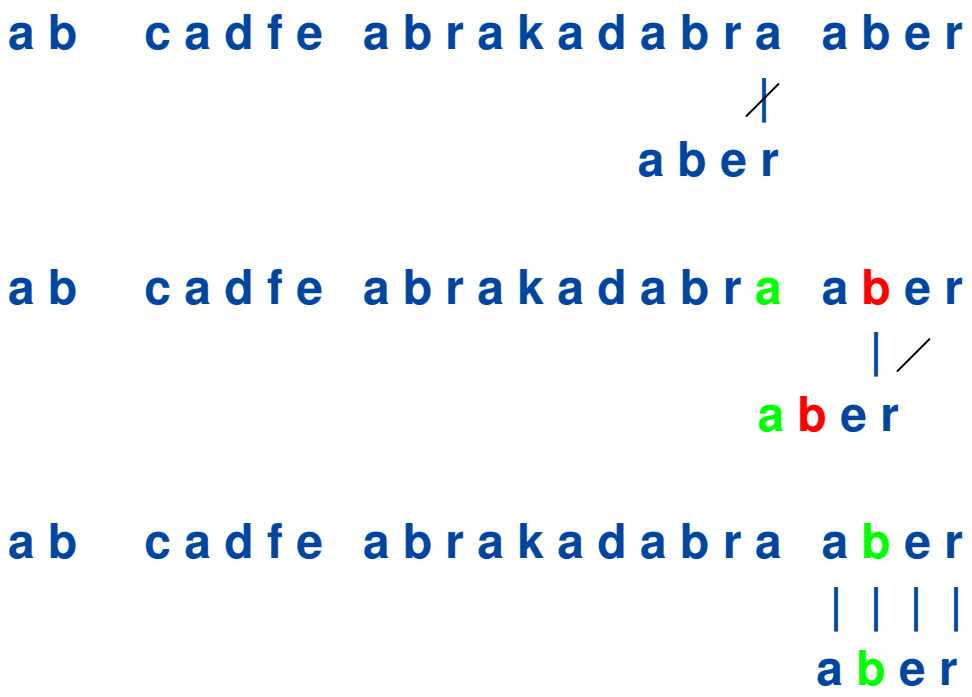
Przykład:



Metoda Boyera-Moore'a (BM)



Metoda Boyera-Moore'a (BM)



Duże przeskoki: mało porównań

Oczekiwany czas działania: $O(m + n/m)$

Metoda Boyera-Moore'a (BM)

Dla $c \in \Sigma$ i wzorca *P* określamy

$\delta(c)$:= indeks pierwszego wystąpienia znaku *c* we wzorcu *P* od prawej strony:

$$= \max \{j \mid p_j = c\}$$

$$= \begin{cases} 0 & \text{dla } c \notin P \\ j & \text{dla } c = p_j \text{ i } c \neq p, \text{gdzie } j < k \leq m \end{cases}$$

Metoda Boyera-Moore'a (BM)

Jaki jest koszt obliczenia wartości δ?

przyjmijmy

$|\Sigma| = d$:

***c* = niezgodny znak**

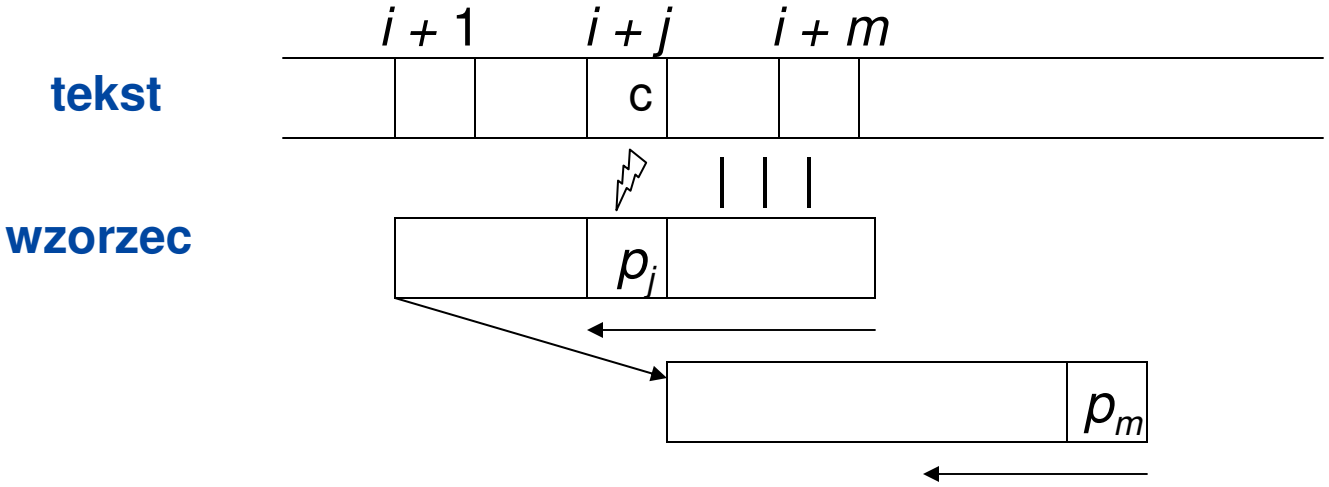
***j* = aktualna pozycja we wzorcu ($c \neq p_j$)**

Metoda Boyera-Moore'a (BM)

Obliczanie przesunięcia

Przypadek 1 ***c* nie występuje we wzorcu *P*. ($\delta(c) = 0$)**

Przesuwamy wzorzec o *j* znaków



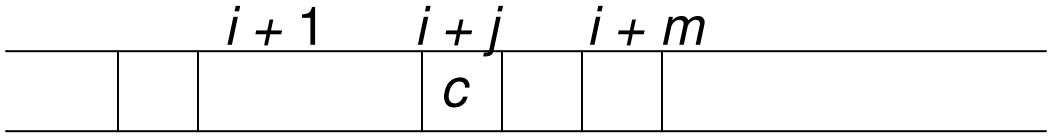
$$\Delta(i) = j$$

Metoda Boyera-Moore'a (BM)

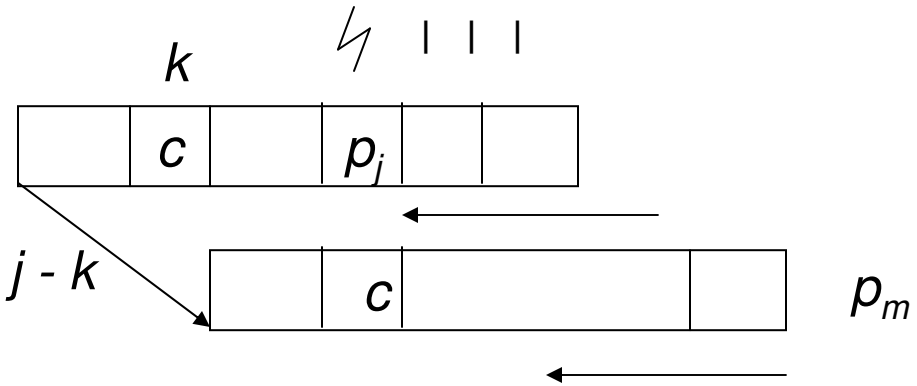
Przypadek 2: ***c* występuje we wzorcu. ($\delta(c) \neq 0$)**

Przesuwamy wzorzec w prawo, aż do pierwszego z prawej strony wystąpienia znaku *c* we wzorcu

tekst



wzorzec

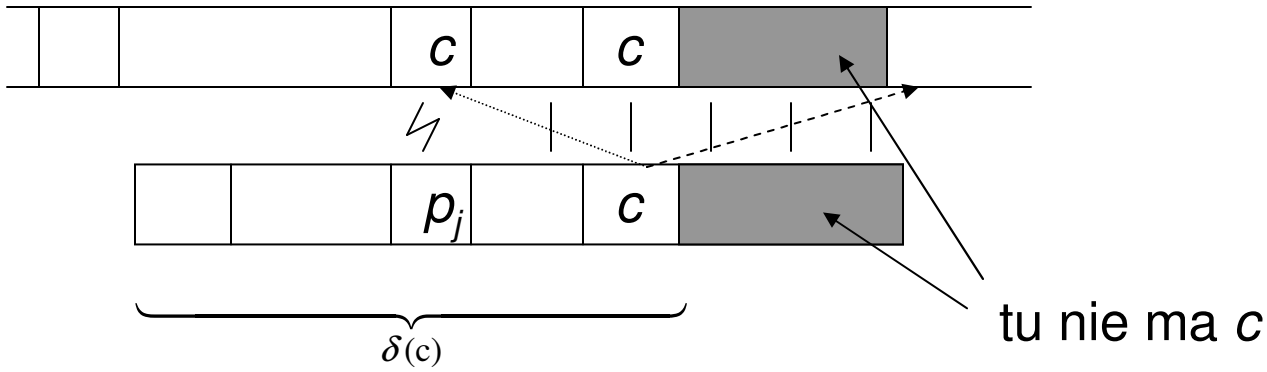


Metoda Boyera-Moore'a (BM)

Przypadek 2 (a)**: jeśli $\delta(c) > j$**

tekst

wzorzec



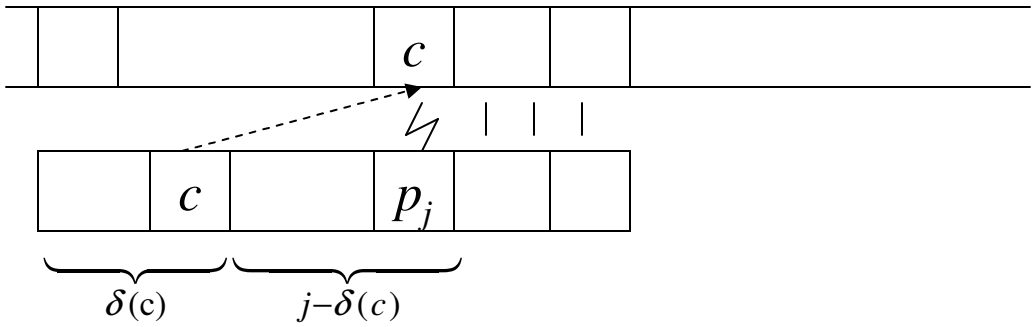
Przesuwamy o: $\Delta(i) = m - \delta(c) + 1$

Metoda Boyera-Moore'a (BM)

Przypadek 2 (b)**: $\delta(c) < j$**

tekst

wzorzec



Przesuwamy o: $\Delta(i) = j - \delta(c)$

Algorytm BM (1 wersja)

Input: **Text *T* and pattern *P***

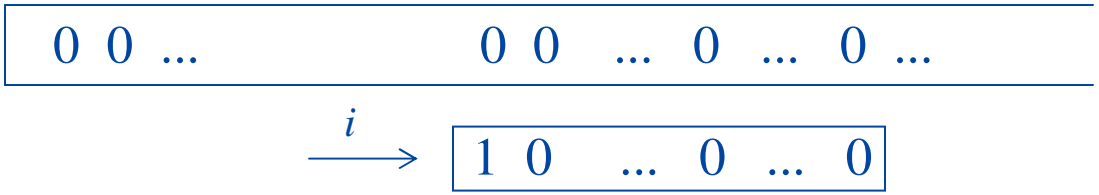
```
1  n := length(T); m := length(P)
2  compute δ
3  i := 0
4  while i ≤ n – m do
5       j := m
6       while j > 0 and P[j] = T[i + j] do
7     j := j – 1
8     end while;
9      if j = 0
10          then output shift i
11                i := i + 1
12     else if δ(T[i + j]) > j
13          then i := i + m + 1 - δ[T[i + j]]
14          else i := i + j - δ[T[i + j]]
15  end while;
```

Algorytm BM (1 wersja)

Analiza:

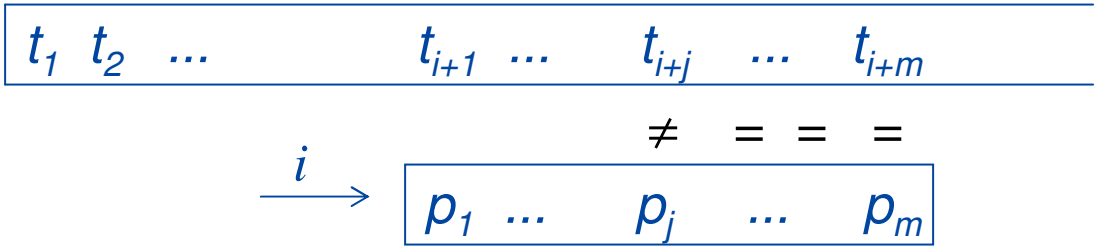
oczekiwany czas działania: $O(m + n/m)$

najgorszy czas działania: $\Omega(n \cdot m)$



Obliczanie najlepszego przesunięcia

Korzystamy z informacji zebranych przed wystąpieniem niezgodności $p_j \neq t_{i+j}$



***wrw[j]* = pozycja końca ostatniego wystąpienia sufiksu $P_{j+1 \ldots m}$ od prawej strony P_j.**

Możliwe przesunięcie: $\gamma[j] = m - wrw[j]$

Przykład

wrw[j] **= pozycja końca ostatniego wystąpienia sufiksu $P_{j+1 \ldots m}$ od prawej strony P_j.**

Wzorzec: banana

wrw[j]	Sprawdzany suffix	Błędny znak	Następne wystąpienie	Pozycja
wrw[5]	a	n	banana	2
wrw[4]	na	a	*** bana na	0
wrw[3]	ana	n	banana	4
wrw[2]	nana	a	banana	0
wrw[1]	anana	b	banana	0
wrw[0]	banana	ε	banana	0

Przykład

***wrw* (banana) = [0,0,0,4,0,2]**

$\Rightarrow$

a b a a b a b a n a n a n a n a

≠ = = =

b a n a n a

b a n a n a

Algorytm BM (2 wersja)

Input: **Text *T* and pattern *P***

```
1  n := length(T); m := length(P)
2  compute δ and γ
3  i := 0
4  while i ≤ n – m do
5       j := m
6       while j > 0 and P[j] = T[i + j] do
7            j := j – 1
8    end while;
9          if j = 0
10           then output shift i
11                     i := i + γ[0]
12           else i := i + max(γ[j], j - δ[T[i + j]])
13   end while;
```