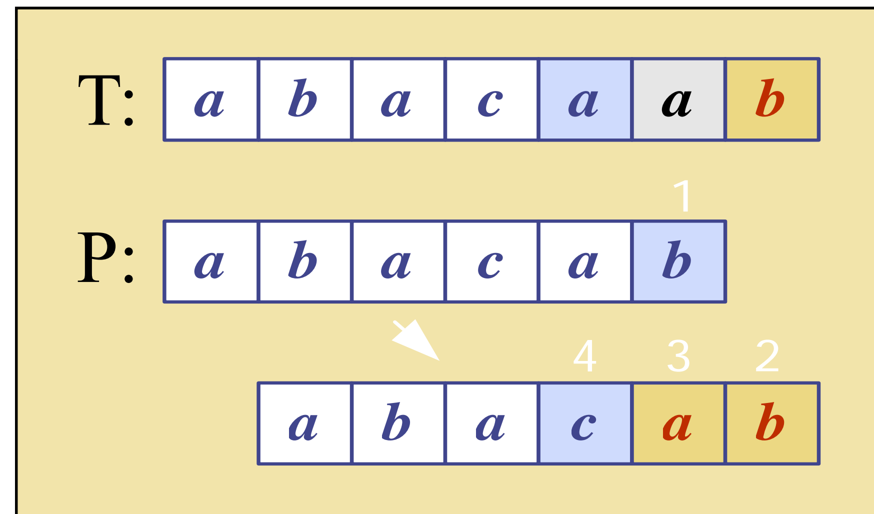
Vyhledávání řetězců (Pattern Matching)



Přehled

1. Co je vyhledávání řetězců
2. Algoritmus „hrubé síly“ (Brute-force)
3. Algoritmus Boyer-Moore
4. Knuth-Morris-Pratt algoritmus
5. Rabin-Karp algoritmus

1. Co je vyhledávání řetězců ?

- Definice:
  - Pro daný textový řetězec T and a vzorový řetězec P, hledáme vzor P uvnitř textu
    - T: “the rain in spain stays mainly on the plain”
    - P: “n th”
- Aplikace:
  - textové editory, webové vyhledávače (např. Google), analýza obrazů, strukturní rozpoznávání

Základní terminologie

- Předpokládejme, že S řetězec velikosti m.

- *podřetězec* S[i .. j] S je část řetězce mezi indexy i a j.

- *prefix (předpona)* S je podřetězec S[0 .. i]
- *suffix (přípona)* S je podřetězec S[i .. m-1]
  - i libovolný index mezi 0 a m-1

Příklad

S

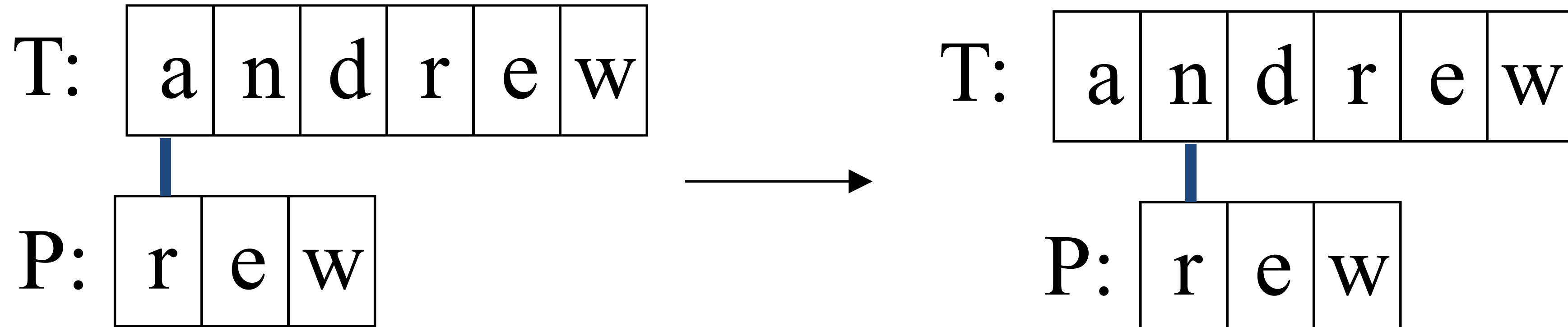
a	n	d	r	e	w

0 5

- Podřetězec S[1..3] == "ndr"
- Všechny možné prefixy S:
  - "andrew", "andre", "andr", "and", "an”, "a"
- Všechny možné suffixy S:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

2. Algoritmus „hrubé síly“ (Brute Force Algorithm)

- Pro každou pozici v textu T kontrolujeme zda v ní nezačíná vzor P.



Brute Force v Javě

Vrací pozici, ve které začíná vzor, nebo -1

```
public static int bfMatching(String text,String template,int i){
     int j,
     int ret_val=-1;
     int n=text.length();
     boolean find=false;
    m=template.length();
    while (i<=n-m && !find) {
  j=0;
     while ((j<m) && (text.charAt(i+j)==template.charAt(j))) {
             j=j+1;
     }
     if (j==m) { ret_val=i;
                 find=true;
               }
    i=i+1;
    }
    return(ret_val);
  }
```

Použití

```
public static void main(String[] args) {
     String text="pokus pohled pohoda podpora";
     String tmpl="po";
     int i;
     boolean nalezen=true;
     i=0;
     do { i=bfMatching(text,tmpl,i);
          if (i>=0) System.out.println("Nalezen v pozici
                                        i="+i);
             else nalezen=false;
          i=i+1;
     } while (nalezen);
}
```

Analýza

- Časová složitost Brute force algoritmu je O(mn) – nejhorší případ
- Většina vyhledáávání v běžném textu má složitost O(m+n).

continued

- Brute force algoritmus je rychlý, pokud je abeceda textu „velká“
  - např. A..Z, a..z, 1..9, atd.

- Algoritmus je pomalý pro „malou“ abecedu
  - tj. 0, 1 (binární soubory, obrázkové soubory, atd.)

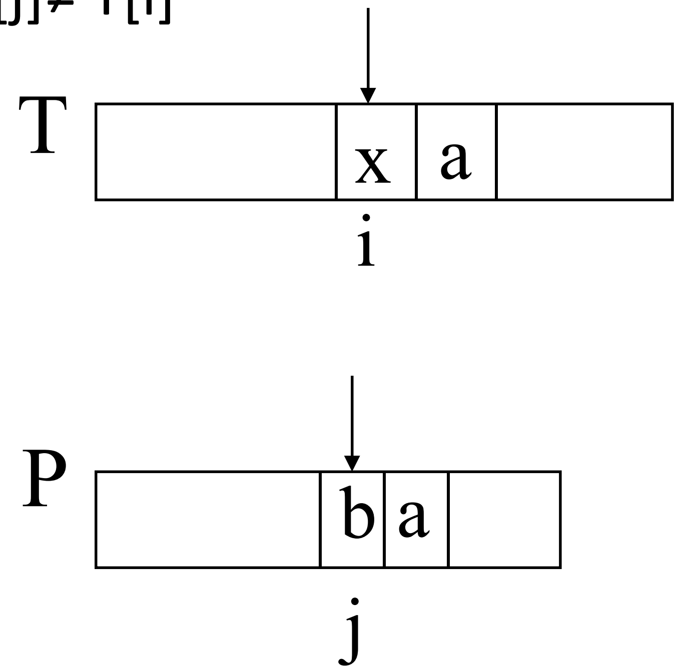
- Příklad – nejhorší případ:
  - T: "aaaaaaaaaaaaaaaaaaaaaaaaah"
  - P: "aaah"

3. Boyer-Moore Algoritmus

- Boyer-Moore algoritmus vyhledávání je založen na

- 1. Zrcadlovém přístupu k vyhledávání
  - hledáme P v T tak, že začínáme na konci P a postupujeme zpět k začátku

- 2. Přeskočením skupiny znaků, které se neshodují (pokud takové znaky existují)
  - Tento případ se řeší v okamžiku kdy P[j]≠ T[i]
  - mohou nastat celkem 3 případy.



Případ 1

- Pokud P obsahuje x, pak zkusíme posunout *P* doprava tak, aby se poslední výskyt x dostal proti x obsaženému v T[i].

T | | x | a | |

i

P | | x | c | b | a |

j

a
posunout i a
j doprava,

T | | x | a | ? | ? | |

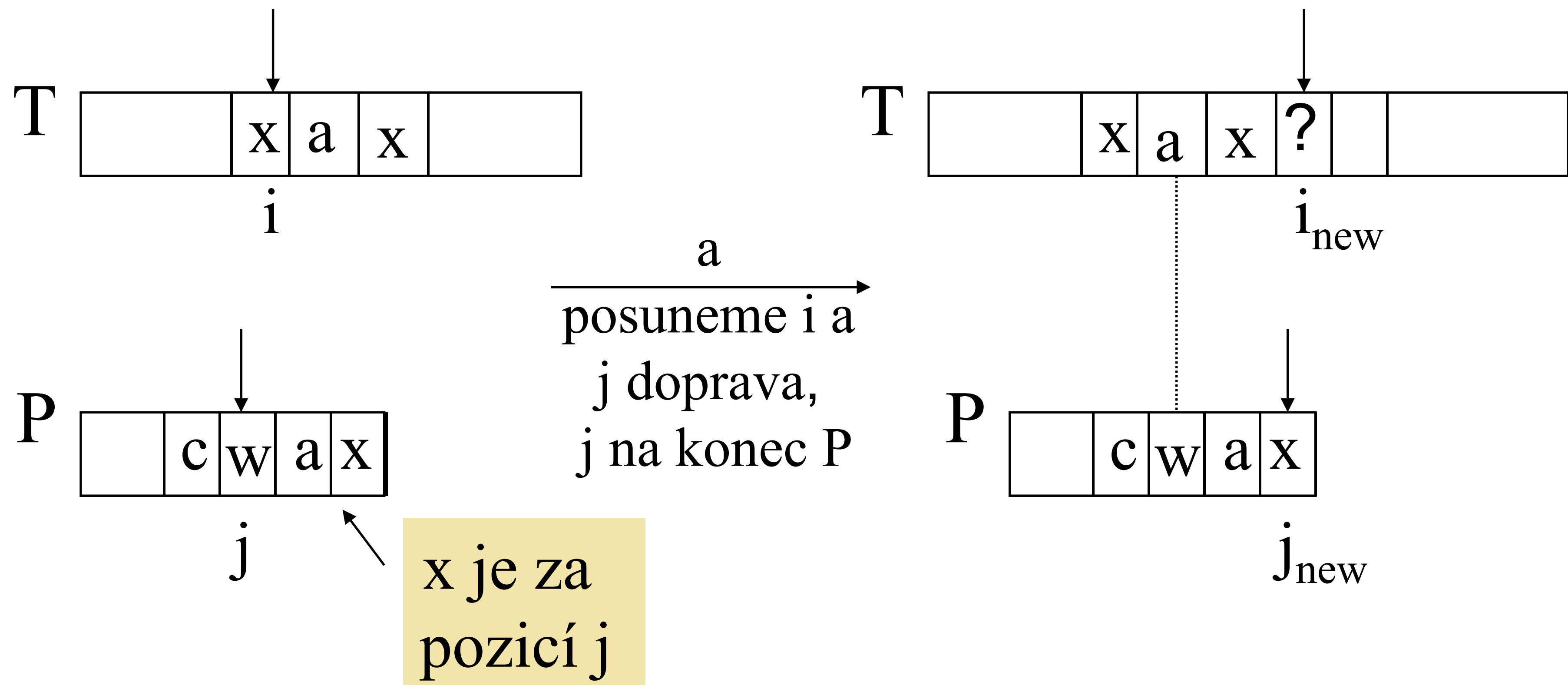
i_{new}

P | | x | c | b | a |

j_{new}

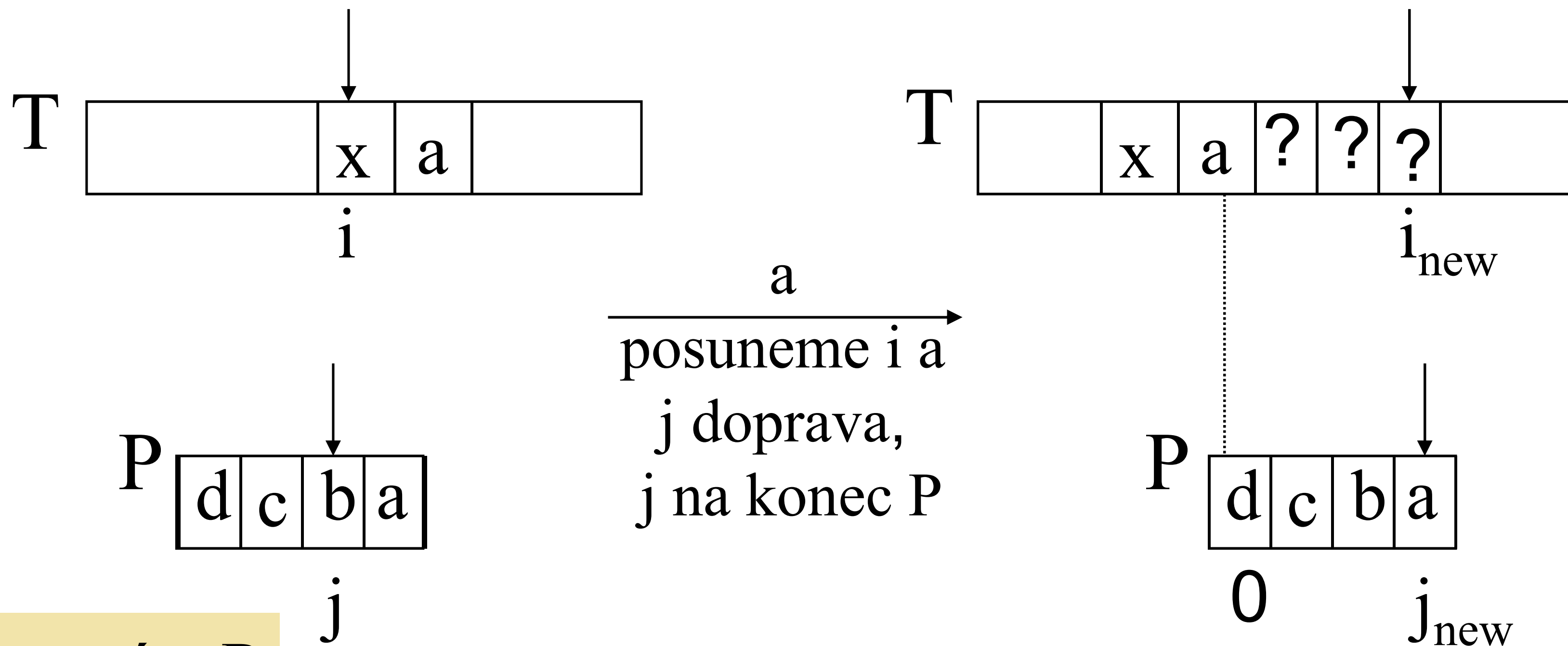
Případ 2

- P obsahuje x, ale posun doprava na poslední výskyt x není možný, pak posuneme P doprava o jeden znak k T[i+1].



Případ 3

- Pokud není možné požít případ 1 a 2, pak posuneme P tak aby bylo P[0] zarovnáno s T[i+1].



Boyer-Moore příklad (1)

T:

a		p	a	t	t	e	r	n		m	a	t	c	h	i	n	g		a	l	g	o	r	i	t	h	m

P:

1: r i t h m

2: r i t h m

3: r i t h m

4: r i t h m

5: r i t h m

6: r i t h m

11 10 9 8 7: r i t h m

Funkce Last()

- Boyer-Moore algoritmus předzpracovává vzor P a pro danou abecedu A definuje funkci Last().
  - Last() zobrazuje všechny znaky abecedy A do množiny celých číslel

- Last(x) je definována jako :   // x je znak v A
  - Největší index i pro který platí, že P[i] == x, nebo
  - -1 pokud žádný takový index v P neexistuje

Příklad funkce Last()

- A = {a, b, c, d}
- P: "abacab"

P	a	b	a	c	a	b
	0	1	2	3	4	5

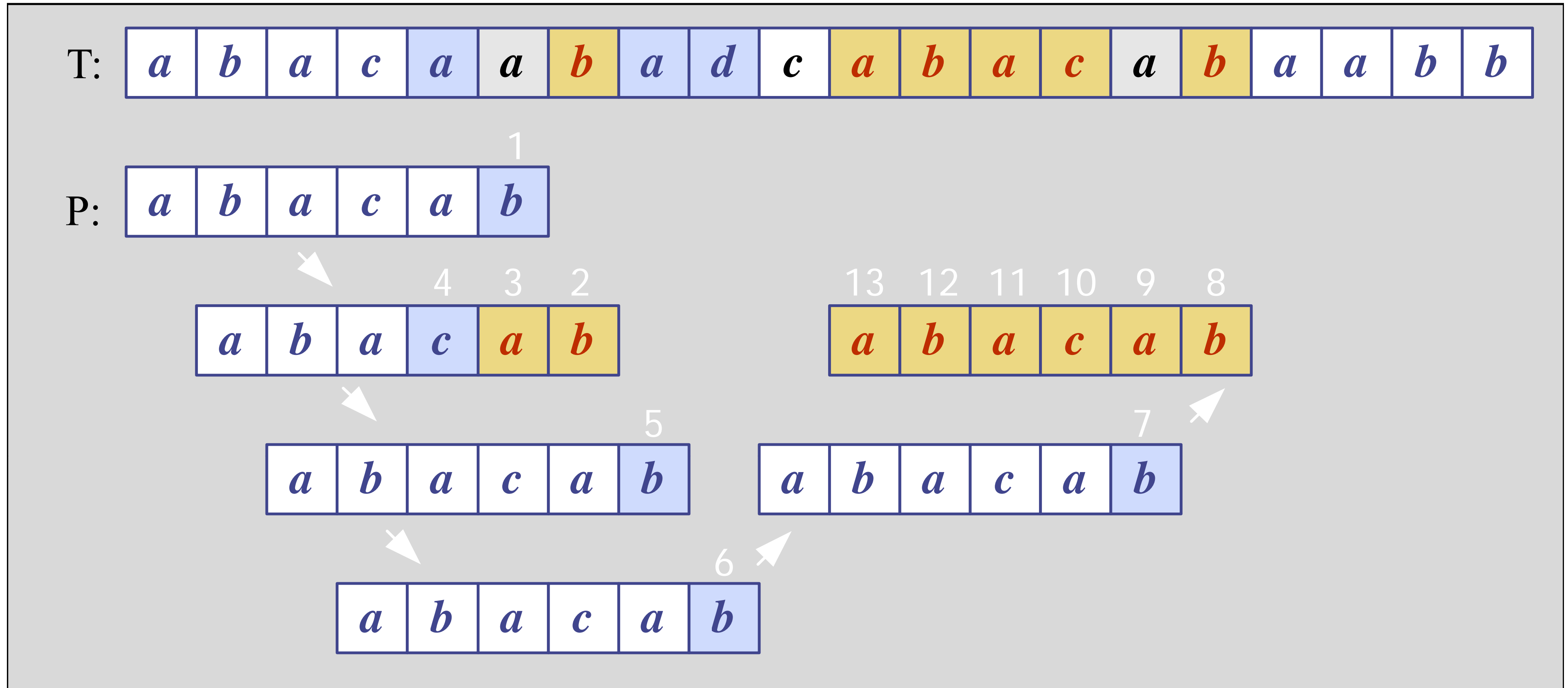
x	a	b	c	d
$Last(x)$	4	5	3	-1

Poznámka

- Last() se počítá pro každý vzor P před začátkem vyhledávání.

- Last() s obvykle uchovává jako pole (tabulka)

Boyer-Moore příklad (2)



x	a	b	c	d
$L(x)$	4	5	3	–1

Boyer-Moore in Javě

Vrací index ve kterém začíná vzor nebo -1

```
public static int bmMatch(String text,
                                  String pattern)
{
  int last[] = buildLast(pattern);
  int n = text.length();
  int m = pattern.length();
  int i = m-1;

  if (i > n-1)
    return -1; // není shoda - vzor je
       // delší než text
```

```
    int j = m-1;
    do {
      if (pattern.charAt(j) == text.charAt(i))
        if (j == 0)
          return i; // match
        else { // zpětný průchod
          i--;
          j--;
        }
      else { // přeskočení znaků
           int lo = last[text.charAt(i)];  //last occ
           i = i + m - Math.min(j, 1+lo);
           j = m - 1;
      }
    } while (i <= n-1);

    return -1; // není shoda
} // konec algoritmu
```

```
public static int[] buildLast(String pattern)
/* vrací pole indexů posledního výskytu každého
   znaku ve vzoru */
{
  int last[] = new int[128]; // ASCII znaky

  for(int i=0; i < 128; i++)
    last[i] = -1; // initializace

  for (int i = 0; i < pattern.length(); i++)
    last[pattern.charAt(i)] = i;

  return last;
} // end of buildLast()
```

Použití

```
public static void main(String args[])
 { if (args.length != 2) {
     System.out.println("Usage: java BmSearch
                            <text> <pattern>");
     System.exit(0);
   }
   System.out.println("Text: " + args[0]);
   System.out.println("Pattern: " + args[1]);

   int posn = bmMatch(args[0], args[1]);
   if (posn == -1)
     System.out.println("Pattern not found");
   else
     System.out.println("Pattern starts at posn "
                                          + posn);
}
```

Analýza

- Časová složitost Boyer-Moore algoritmu je v nejhorším případě O(nm + A)

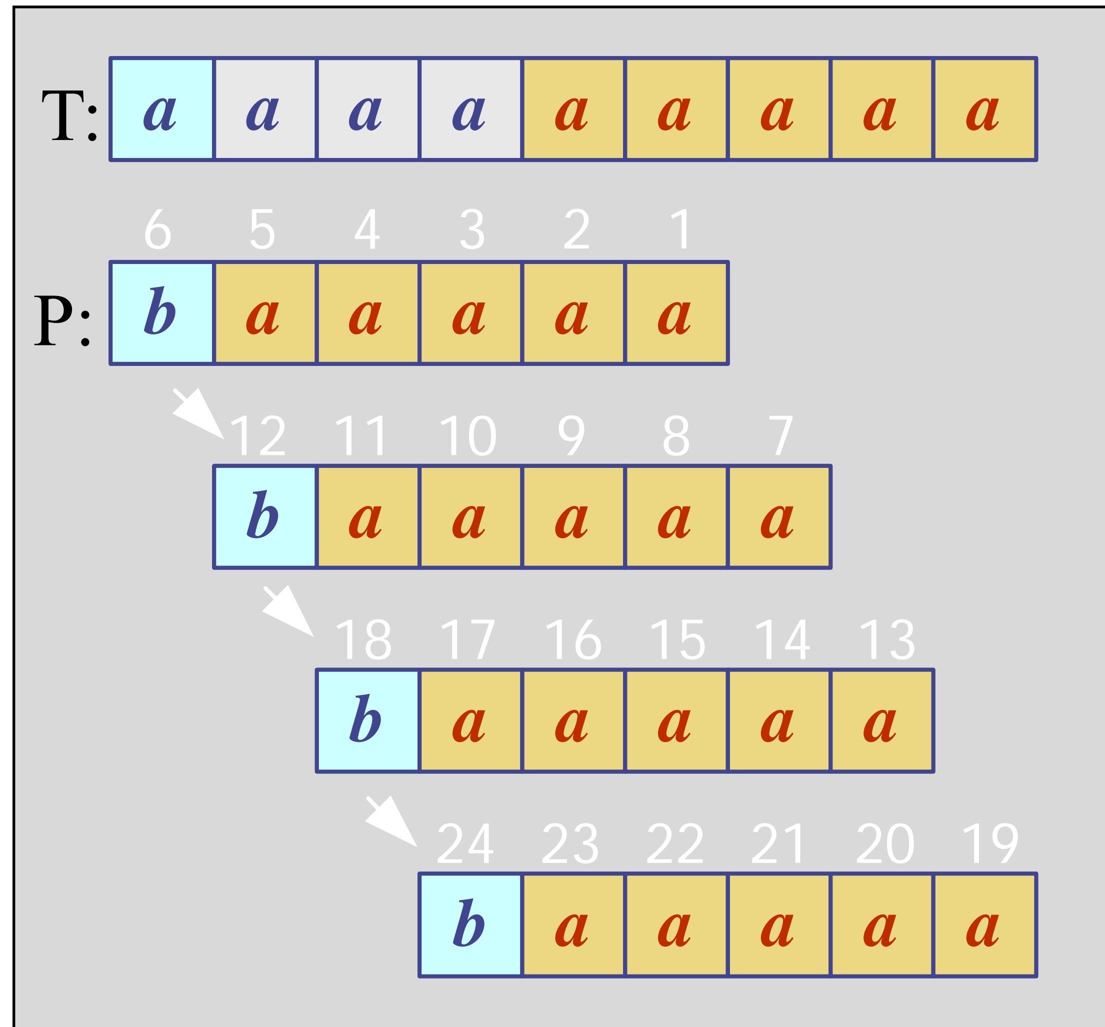
- Boyer-Moore je rychlejší pokud je abeceda (A) velká, pomalý pro malou abecedu

  tj. algoritmus je vhodný pro text, špatný pro binární vstupy

- Boyer-Moore rychlejší než *brute force* v případě vyhledávání v textu.

Příklad nejhoršího případu

- T: "aaaaa...a"
- P: "baaaaa"



4. KMP Algoritmus

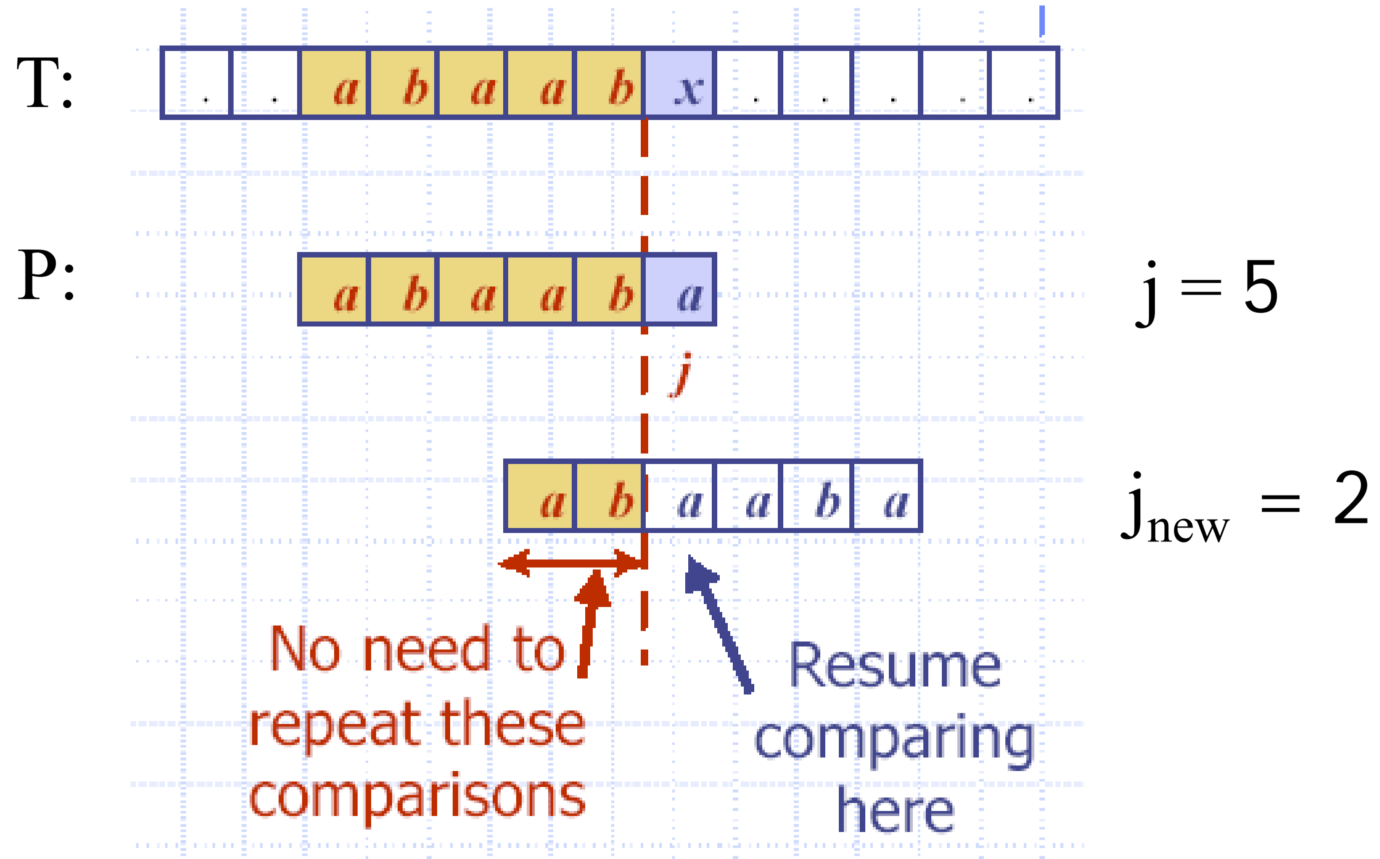
- Knuth-Morris-Pratt (KMP) algoritmus vyhledává vzor v textu *zleva do prava* (jako brute force algoritmus).

- Posun vzoru je řešen mnohem inteligentněji než v brute force algoritmu.

- Pokud se vyskytne neshoda mezi textem and vzorem P v P[j], jaký je *největší možný posun* vzoru abychom se vyhnuly zbytečnému porovnávání?

- *Odpověď*: největší prefix P[0 .. j-1], který je suffixem P[1 .. j-1]

Příklad



Příklad

$j == 5$

- Nalezneme největší prefix (start) :
  "a b a a b" (P[0..j-1])

  jehož suffix (end) :
  "b a a b" (p[1 .. j-1])

- Odpověď: "a b"
- Nastavíme j = 2 // nová hodnota j

KMP chybová funkce

- KMP předzpracovává vzor, abychom nalezli shodu prefixů vzoru se sebou samým.
- k = pozice před neshodou (j-1).
- *Chybová funkce (failure function)* F(k) definována jako nejdelší prefix P[0..k] který je také suffixem P[1..k].

Příklad chybové funkce

(k == j-1)

- P: "abaaba"

k	0	1	2	3	4	5
F(k)	0	0	1	1	2	3

F(k) velikost největšího prefixu, který je zároveň sufixem

- V programu je F() implementována,jako pole (popř. tabulka.)

Použití chybové funkce

- Knuth-Morris-Pratt algoritmus modifikuje brute-force algoritmus.
  - Pokud se vyskytne neshoda v P[j] (i.e. P[j] != T[i]), pak

```
k = j-1;
j = F(k);    // získání nové hodnoty j
```

KMP v Javě

```
public static int kmpMatch(String text,
                           String pattern)
 {
   int n = text.length();
   int m = pattern.length();

   int fail[] = computeFail(pattern);

   int i=0;
   int j=0;
     :
```

```
    while (i < n) {
      if (pattern.charAt(j) == text.charAt(i)) {
        if (j == m - 1)
          return i - m + 1; // match
        i++;
        j++;
      }
      else if (j > 0)
    j = fail[j-1];
      else
        i++;
    }
    return -1; // no match
  } // end of kmpMatch()
```

```java
public static int[] computeFail(
                            String pattern)
{
  int fail[] = new int[pattern.length()];
  fail[0] = 0;

  int m = pattern.length();
  int j = 0;
  int i = 1;
 :
```

```
while (i < m) {
    if  (pattern.charAt(j) ==
          pattern.charAt(i)) {    //j+1 chars match
      fail[i] = j + 1;
      i++;
      j++;
    }
    else if (j > 0) // j follows matching prefix
      j = fail[j-1];
    else {      // no match
      fail[i] = 0;
      i++;
    }
  }
  return fail;
} // end of computeFail()
```

Použití

```
public static void main(String args[])
{ if (args.length != 2) {
    System.out.println("Usage: java KmpSearch
                         <text> <pattern>");
    System.exit(0);
  }
  System.out.println("Text: " + args[0]);
  System.out.println("Pattern: " + args[1]);

  int posn = kmpMatch(args[0], args[1]);
  if (posn == -1)
    System.out.println("Pattern not found");
  else
    System.out.println("Pattern starts at posn "
                                    + posn);
}
```

Příklad

T: a b a c a a b a c c a b a c a b a a b b

P: a b a c a b (positions 1 2 3 4 5 6)

7: a b a c a b

8 9 10 11 12: a b a c a b

13: a b a c a b

14 15 16 17 18 19: a b a c a b

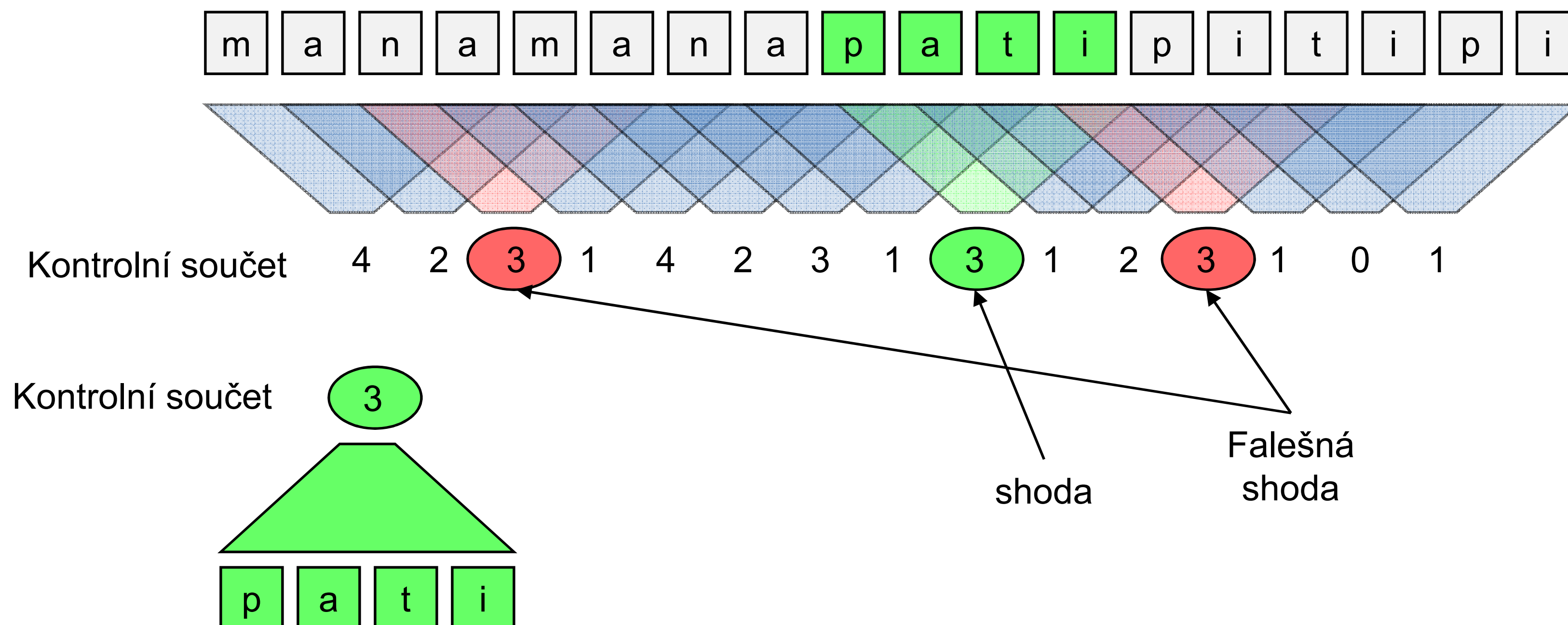
k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1	0	1	2

KMP výhody

- KMP běží v optimálním čase: O(m+n)
- Algoritmus se nikdy neposouvá zpět ve vstupním textu T
  - To činí algoritmus obzvlášť výhodný zpracování velkých souborů

5. Rabin-Karp Algoritmus

- Základní myšlenka: Vypočítat
  - kontrolní součet pro vzor P (délky m) a
  - kontrolní součet pro každý podřetězec řetězce T délky m
  - procházet řetězcem T a porovnat kontrolní součet každého podřetězce s kontrolním součtem vzoru. Pokud dojde ke shodě vzoru provést test znak po znaku.



Rabin-Karp Algoritmus

- Výpočet kontrolního součtu:
  - Zvolíme prvočíslo q
  - Zvolíme d = |Σ| - tj. počet všech možných znaků v použité abecedě

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

- Příklad:

  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Pak d = 10, q = 13
  - Nechť P = 0815

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

Jak vypočítat kontrolní součet : Hornerovo schéma

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

- Použitím

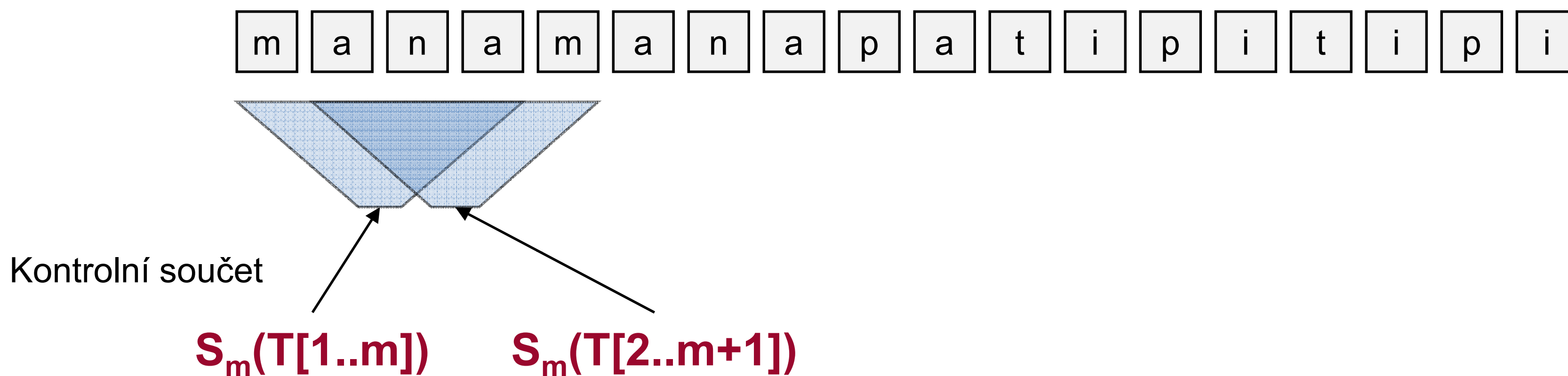
$$S_m(P) \equiv \sum_{i=1}^{m} d^{m-i} P[i] \equiv d\left(\sum_{i=1}^{m-1} d^{m-i-1} P[i]\right) + P[m] \equiv dS_{m-1}(P[1..m-1]) + P[m] \pmod q$$

- Příklad:
  - $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Pak d = 10, q = 13
  - Nechť 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 vypočítat kontrolní součet pro text

- Začneme s $S_m(T[1..m])$



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

Rabin-Karp Algoritmus

```
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
     od
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 “Pattern occurs with shift” s
             fi
12.          if s < n-m then
13.                  t_(s+1) ← (d(t_s-T[s+1]h) + T[s+m+1]) mod q
             fi
     od
```

Kontrolní součet vzoru P

Kontrolní součet textu T[1..m]

Kontrolní součty se shodují provádí se test shody řetězců

výpočet kontrolního součtu pro T[s+1..s+m] s využitím kontrolního součtu T[s..s+m-1]

Vlastnosti Rabin-Karp algoritmu

- čas běhu Rabin-Karp algoritmu je v nejhorším případě $O(m(n-m+1))$

- Pravděpodobnostní analýza
  - Pravděpodobnost falešné shody je pro náhodný vstup $1/q$
  - Předpokládaný počet falešných shod $O(n/q)$

  - Předpokládaný čas běhu Rabin-Karp algoritmu je
    $O(n + m(v+n/q))$
    kde v je počet správných posuvů

- Pokud zvolíme $q \geq m$ a očekávaný počet posuvů je malý je předpokládaná doba běhu Rabin-Karp algoritmu $O(n + m)$.