Vyhledávání řetězců

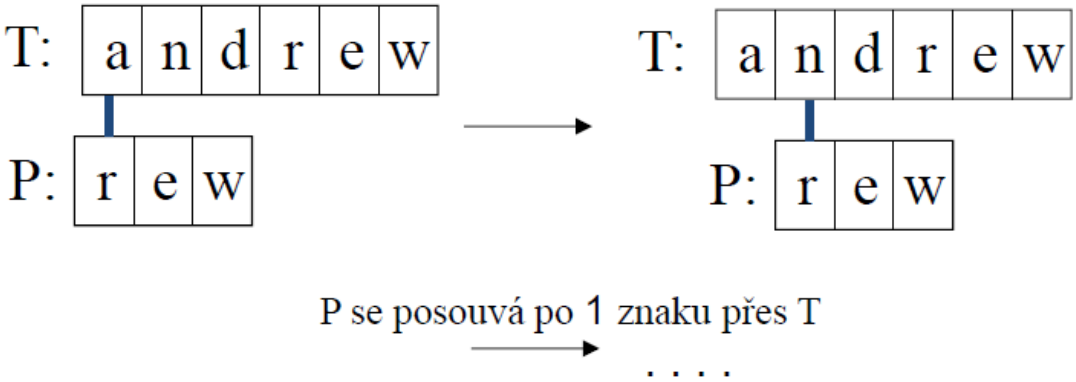
T – textový řetězec(v čem to hledáme)

P – vzorový řetězec (to co hledáme)

- Řetězec S velikosti m
- Podřetězec S[i…j] je část řetězce mezi indexy i a j
- Prefix (předpona) S je podřetězec S[0…i]
- Sufix (přípona) S je podřetězec S[I … m-1]

Brute force (algoritmus hrubé síly)

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

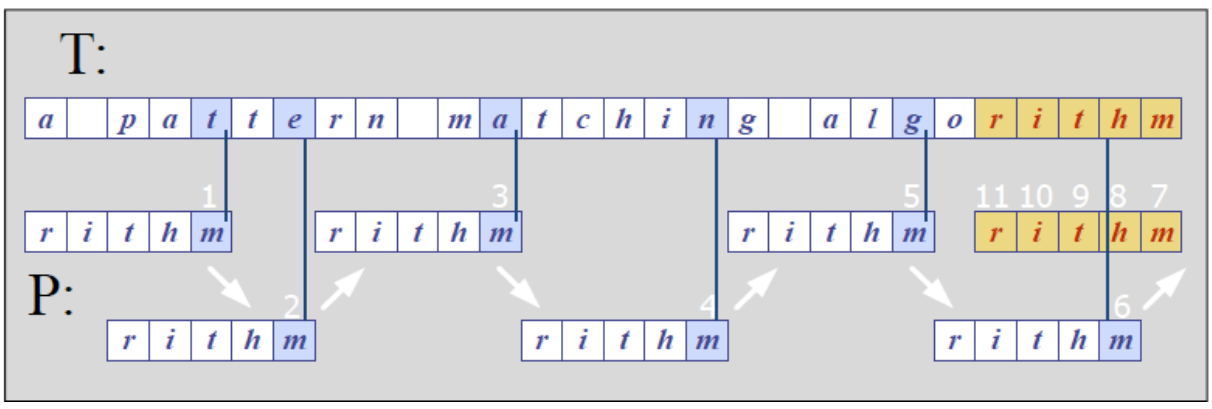


- Časová složitost Brute force algoritmu je O(mn) – nejhorší případ
- Většina vyhledávání v běžném textu má složitost O(m+n)
- Je rychlý, pokud je abeceda textu velká
- Algoritmus je pomalý pro malou abecedu

Boyer-Moore algoritmus

1. Zrcadlový přístup k vyhledávání. (hledáme P v T tak, že začínáme na konci P a postupujeme zpět k začátku)
2. Přeskočíme skupinu znaků, které se neshodují (pokud takové znaky existují)

Př:

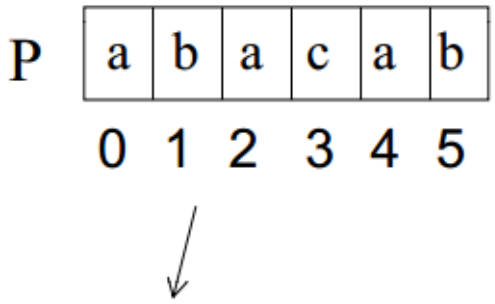


Funkce Last()

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

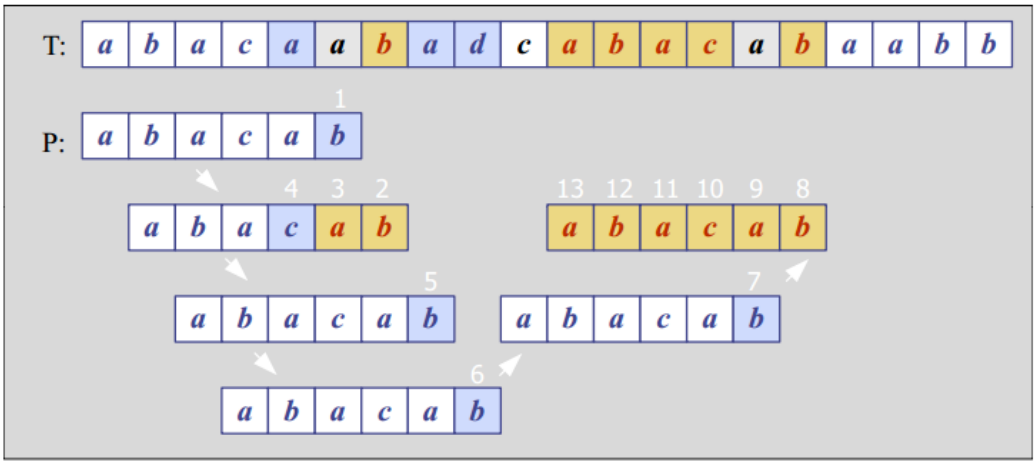
Příklad funkce Last()

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



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

Boyer-Moore příklad (2)



x	*a*	*b*	*c*	*d*
L(x)	4	5	3	−1

Boyer-Moore v Javě

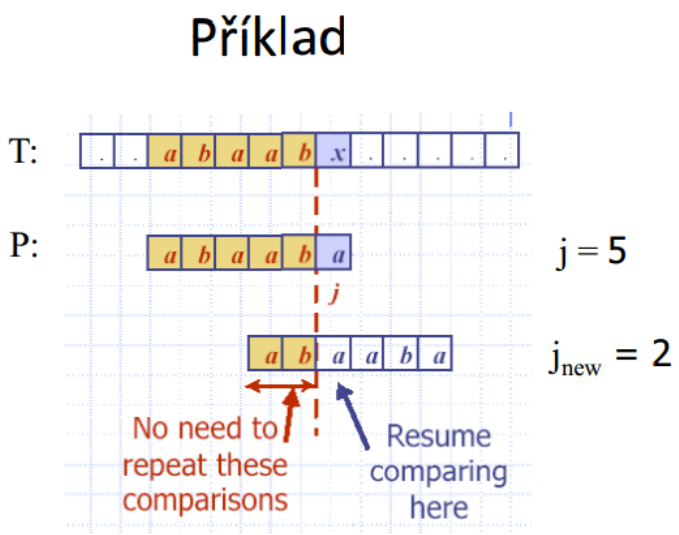
```
Public static int bmMatch(String text, String pattern({
int last[] = buildLast(pattern);
int n = text.length();
int m = pattern.length();
int i = m int i = m-1;
if (i > n-1)
return -1; // není shoda není shoda – vzor je
 // delší než text
Int j = m-1;
do {
if (pattern.charAt{j)==text.charAt(i))
         if(j==0){
         return I; //match
         else{ //zpetny pruchod
         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 //neni shoda
}//konec

Last:
publi t ti i t[] blic static int[] b ildL t buildLast(St i tt ) (String pattern)
 /* vrací pole indexů posledního výskytu každého
znaku ve vzoru */
{
int last[] = new int[128]; // ASCII znaky
for(int i= for(int i=0;i< 128; i++)
last[i] = -1; // initializace
for (int i = 0; i < p g attern.length(); i++)
last[pattern.charAt(i)] = i;
return last;
} // d f b ild t() } // end of buildLast()
```

- 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.

Knuth-Morris-Pratt (KMP) algoritmus

- Vyhledává vzor v textu zleva do prava (jako BF algo.)
- Posun vzoru jje řešen mnohem inteligentněji než v brute force algoritmu.
- Pokud se vyskytne neshoda mezi textem a vzorem P v P[j], jaký je největší možný posun vzoru abychom se vyhnuly zbytečnému posouvání?
- ->největší prefix P[0 … j-1] je sufixem P[1 … j-1]



Chybová funkce

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

(k == j-1)

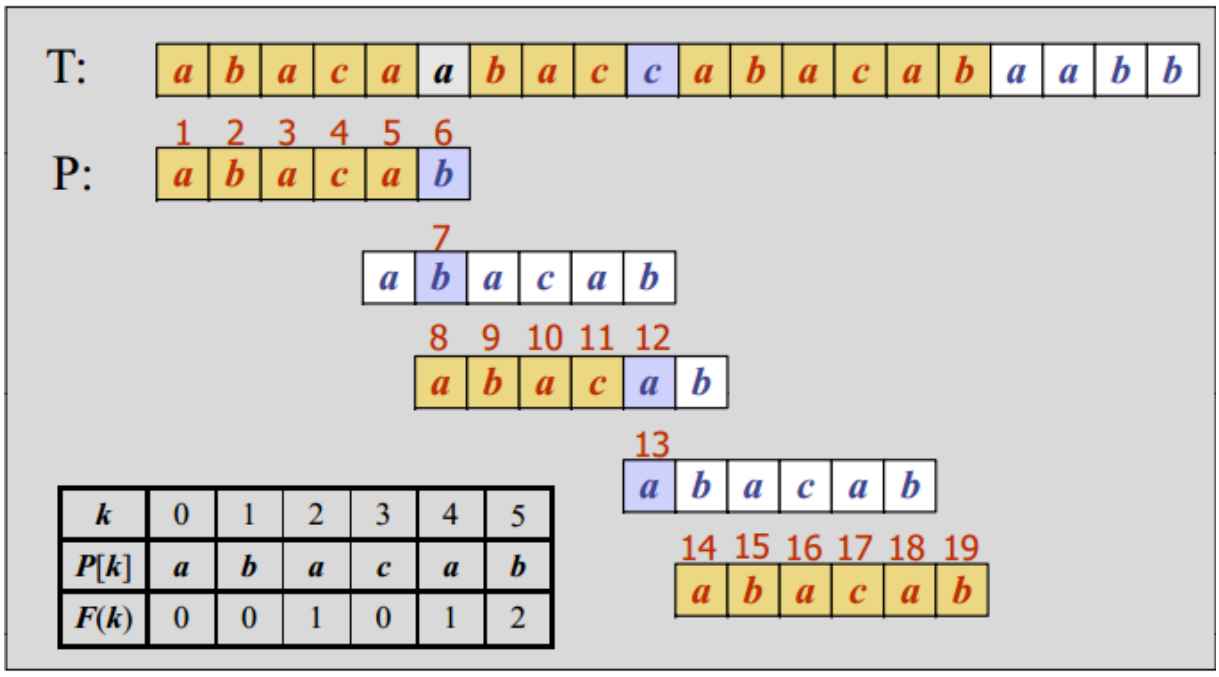
- P: "abaaba"

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

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

Použití: KMP modifikuje BF algo

- Pokud se vyskytne neshoda v P[j] (i.e.P[j]!=T[i]), pak
- K=j-1;
- J=F(k); // získání nové hodnoty j

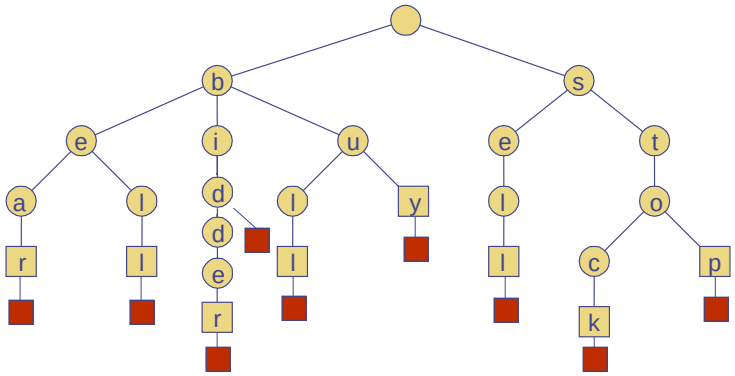


k	0	1	2	3	4	5
P[k]	a	b	a	c	a	b
F(k)	0	0	1	0	1	2

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

Standardní Trie

- Standardní trie pro množinu řetězců S je k-ární (k je velikost použité abecedy) uspořádaný strom, pro který platí:
  - Každý uzel, kromě kořene, je ohodnocen znakem
  - Následníci uzlu jsou abecedně uspořádány
  - Symboly v uzlech na cestě z kořene do externího uzlu tvoří řetězec množiny S
- Příklad: standardní trie pro množinu řetězců
  S = { bear, bell, bid, bull, buy, sell, stock, stop }

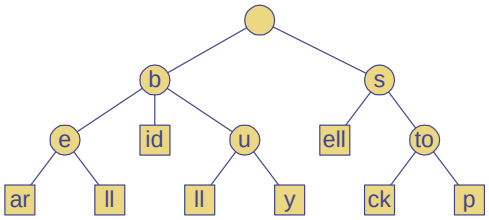


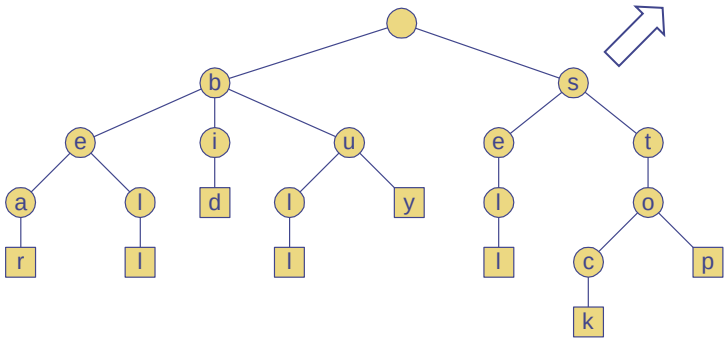
Použití:

- Vyhledávání slov [O(m)]
- Vyhledávání prefix [O(m)]

Komprimovaná Trie

- Komprimovaná trie má vnitřní uzly stupně nejméně 2
- Získáva se ze standardní trie, kompresí řetězců tzv. „redundantních" uzlů tj. uzlů, které mají pouze jednoho následníka





SCS – Shortest common super-sequence

Algoritmus nalezení nejkratšího společného „nadřetězce"

- Podobný algoritmu LCS
- **Definice:** Nechť X a Y jsou dva řetězce znaků. Řetězec Z je „nadřetězec" (**super-sequence**) řetězců X a Y pokud jsou oba řetězce X a Y podřetězcem (subsequence) Z.
- **Shortest common super-sequence algoritmus:**

  Vstup: dva řetězce X aY.

  Výstup: nejkratší společný „nadřetězec" X a Y.
- Příklad: X=**abc** a Y=**abb**. Oba řetězce **abbc abcb** jsou nejkratším společným „nadřetězcem" řetězců X a Y.

nejkratsi spolecnej nadretezec

tabulka – prvni radek a sloupec – indexy sloucpu (radku), cisluje se od 0

shoda: vezmeme cislo vlevo nahore od porovnavane pozice a zvednem o jedna, sipka doleva nahoru

neshoda: vezmeme mensi cislo z hodnot vlevo a nahore od soucasne pozice, zvedneme o jednicku a sipka timto smerem

LCS – Longest common subsequence

Algoritmus nalezení nejdelšího společného podřetězce

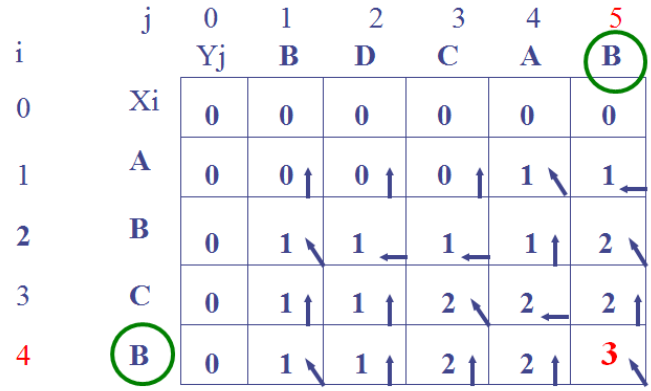
- LCS algoritmus je jedním ze způsobů jak posuzovat podobnost mezi dvěma řetězci
- algoritmus se často využívá v biologii k posuzování podobnosti DNA sekvencí (řetězců obsahujících symboly A,C,G,T)
- Příklad X = AGTCAACGTT, Y=GTTCGACTGTG
- Podřetězce jsou např. S = AGTG and S'=GTCACGT
- Jak lze tyto podřetězce nalézt ?
  - Použitím hrubé síly : pokud |X| = m, |Y| = n, pak existuje 2^m podřetězců x, které musíme porovnat s Y (n porovnání) tj. časová složitost vyhledání je O(n 2^m)
  - Použití dynamického programování – složitost se sníží na O(nm)

nejdelsi spolecnej podretezec

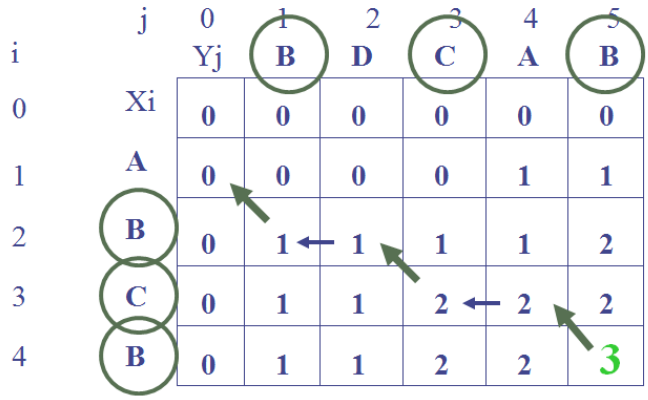
tabulka – prvni radek a sloupec nuly

shoda: vezmeme cislo vlevo nahore od porovnavane pozice a zvednem o jedna, sipka doleva nahoru

neshoda: vezmeme vetsi cislo z hodnot vlevo a nahore od soucasne pozice a sipka timto smerem

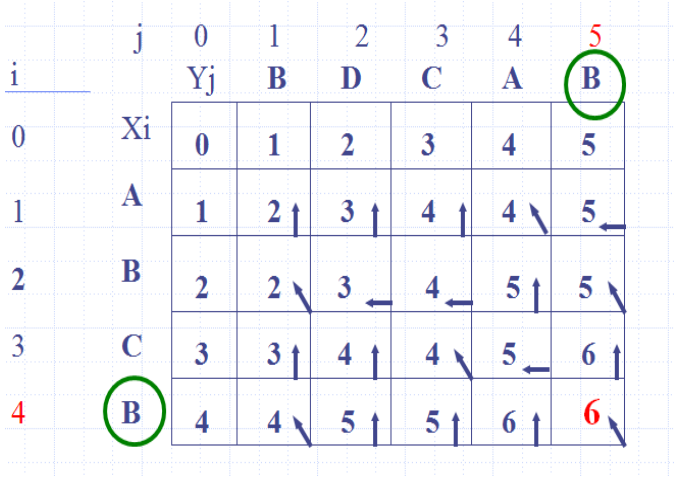


i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

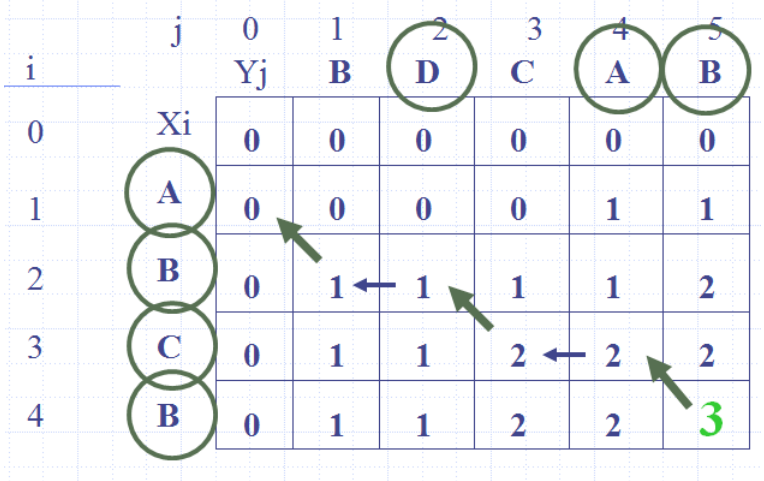


i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

SCS



i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	1	2	3	4	5
1	A	1	2	3	4	4	5
2	B	2	2	3	4	5	5
3	C	3	3	4	4	5	6
4	B	4	4	5	5	6	6



i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3