Porovnávanie reťazcov

Úvod

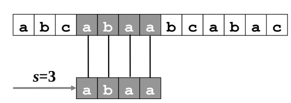
- Porovnávanie reťazcov, presnejšie hľadanie výskytov reťazca v reťazci (string-matching) je pomerne dôležitou súčasťou širokej domény zaoberajúcej sa spracovaním textu. Algoritmy na porovnávanie textov sa využívajú pri implementácií softvérových systémov, ktoré sú reálne nasadené v praxi. Takisto však hrajú dôležitú rolu v teoretickej informatike, kde môžu byť výzvou pre navrhovanie efektívnejších algoritmov.

základné pojmy

- S=“AGCTTGA”
- $|S|$=7, dĺžka reťazca S
- podreťazec: $S_{i,j}=S_i S_{i+1} \ldots S_j$
  - príklad: $S_{2,4}$=“GCT”
- podpostupnosť reťazca S: vymazaním niekoľkých (vrátane žiadneho) znakov z S
  - “ACT” and “GCTT” sú podpostupnosti.
- predpona S: $S_{1,k3}$
  - “AGCT” je predpona S.
- prípona S: $S_{h,|S|}$
  - “CTTGA” je prípona S.

základné pojmy

- Uvažujme 2 reťazce:
  - Vzor P[1...*m]*, ktorý má dĺžku *m*
  - Text T[1...*n*], ktorý má dĺžku *n*
- Vzor P sa nachádza v texte T s posunutím *s* ak platí:
  - T[*s*+*1*...*s*+*m*] = P[*1*...*m*]
- Príklad: T = abcabaabcabac, P = abaa
  - *m*=4, *n*=13, *s*=3

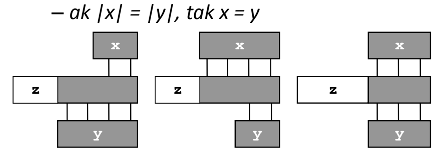


základné pojmy

- Predpona (prefix): reťazec *w* je predponou reťazca *x*, ak *x* = *wy*, kde *y* je akýkoľvek reťazec z použitej abecedy *Σ*, t.j. prvok z množiny *Σ**
  - Napr: pre(*ab,abcca)*
- Prípona (suffix): reťazec *w* je príponou reťazca *x*, ak *x* = *yw*, kde *y* je akýkoľvek reťazec z použitej abecedy *Σ*, t.j. prvok z množiny *Σ**
  - Napr: suf(*cca,abcca)*

Lema

- Predpokladajme, že “*x*”, “*y*” a “*z*” sú reťazce, pre ktoré platí suf(*x*, *z*) a suf(*y*, *z*), potom:
  - ak $|x| \leq |y|$, *tak suf(x,y)*
  - *ak* $|x| \geq |y|$, *tak suf(y,x)*
  - *ak* $|x| = |y|$, *tak* $x = y$



najznámejšie algoritmy

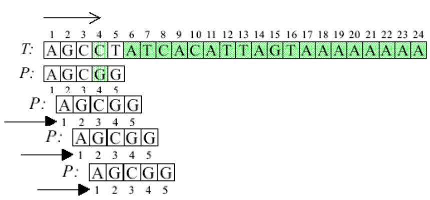
Algoritmus	fáza predspracovania	Vyhľadávacia fáza
Naivný	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Konečný automat	$O(m\|\Sigma\|)$	$\Theta(m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(m)$

- Ďalšie algoritmy, ich opisy, vizualizácie a zdrojové kódy môžete nájsť na : http://www-igm.univ-mlv.fr/~lecroq/string/

Naivné hľadanie výskytu reťazca v reťazci

- Nemá fázu predspracovania
- Vždy sa posúva len o 1 pozíciu doprava
- Porovnávanie môže prebiehať v akomkoľvek poradí
- Veľká časová zložitosť
- Vykoná sa 2n porovnávaní textu

naivné = hrubou silou



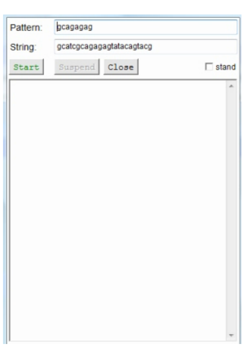
čas: O(*mn*) kde *m*=|*P*| a *n*=|*T*|.

Naivné hľadanie výskytu reťazca v reťazci

- Samotný algoritmus pozostáva z porovnávania znakov na všetkých miestach medzi 0 a *n-m*. Pri každom kroku sa posúva iba o 1 miesto doprava.

```
Naivne porovnavanie (T, P)
n ←length[T]
m ←length[P]
for s ←0 to n −m
   do if P[1 . . . m] = T[s+1. . . s+m]
        then
        print "vyskytuje sa s posunutim" s
```


Naivné hľadanie výskytu reťazca v reťazci



Naivné hľadanie výskytu reťazca v reťazci

```
C++:
void BF(char *x, int m, char *y, int n)
{
int i, j;
/* Searching */
for (j = 0; j <= n - m; ++j)
{
   for (i = 0; i < m && x[i] == y[i + j]; ++i); if (i >= m)
            OUTPUT(j);
} }
```

2 druhy algoritmov hľadania výskytu reťazca

- predspracovanie vzoru P (P sa nemení, T sa mení)
  - napr: dopyt P do databázy T
  - algoritmy:
    - Knuth – Morris - Pratt
    - Boyer – Moore
- predspracovanie textu T (T sa nemení, P sa mení)
  - napr: hľadá sa vzor P v slovníku T
  - algoritmus:
    - príponový strom

Predspracovanie vzoru

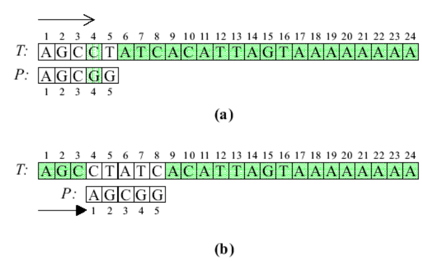
- dvojfázové algoritmy
  - fáza 1 : vygeneruj pole, ktoré bude indikovať smer pohybu.
  - fáza 2 : použi to pole na pohyb a hľadanie výskytu
- príklady
- algoritmus KMP:
  - navrhli Knuth, Morris and Pratt v 1977.
- algoritmus BM:
  - navrhli Boyer a Moore v 1977.

Knuthov-Morrisov-Prattov algoritmus

- KMP algoritmus vychádza z analýzy algoritmu naivného vyhľadávania. V určitých situáciách vie využiť informáciu získanú čiastočným porovnávaním vybraného podreťazca a vzoru a posunúť podreťazec o viac než jeden znak.

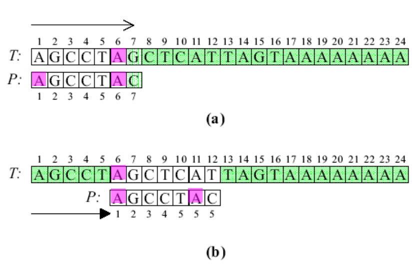
prvý prípad v KMP algoritme

- prvý symbol vzoru *P* sa viac vo vzore *P* nenachádza.
- možno sa posunúť až ku T_4, pretože $T_4 \neq P_4$ (a $T_i = P_i$ pre prvé tri pozície i=1,2,3) v (a).



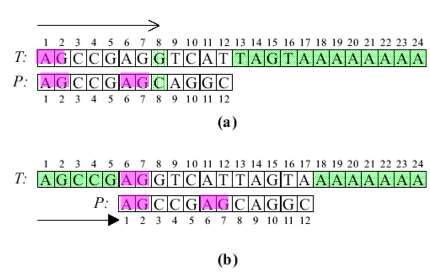
druhý prípad v KMP algoritme

- prvý symbol vo vzore *P* sa v ňom ešte nachádza na niektorom ďalšom mieste.
- $T_7 \neq P_7$ v (a). treba sa posunúť ku T_6, pretože $P_6 = P_1 = T_6$.

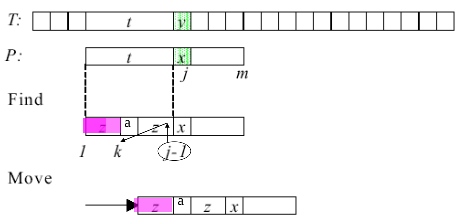


tretí prípad v KMP algoritme

- predpona vzoru *P* sa vyskytuje v *P* ešte raz.
- $T_8 \neq P_8$ in (a). Treba sa posunúť k T_6, pretože $P_{6,7} = P_{1,2} = T_{6,7}$.



základná myšlienka KMP algoritmu



Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
A	B	C	D	A	B	D
Y	Y	Y	N			

m = 0

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
A	B	C	D	A	B	D
Y	Y	Y	Y	Y	Y	N

m = 4

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]
A	B	C	D	A	B	D
		N				

m = 10

Knuth-Morris-Pratt príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]	t[11]	t[12]	t[13]
A	B	C		A	B	C	D	A	B		A	B	C

p[0]	p[1]	p[2]	..
A	B	C	..
Y	Y	Y	

m = 11

Knuthov-Morrisov-Prattov algoritmus

algorithm *kmp_search*:
Input pole znakov T (text, v ktorom sa hľadá výskyt vzoru)
pole znakov P (text, vzor, ktorého výskyt sa hľadá)
Output celé číslo (miesto v poli T, počínajúc ktorým sa našiel výskyt vzoru P)

define variables:
integer m ← 0 (začiatočné miesto súčasného výskytu v T)
integer i ← 0 (miesto súčasného znaku v P)
pole celých čísiel π (tabuľka, ktorá sa vypočíta inde)
while m + i < dĺžka π do:
if P[i] = T[m + i] **then**
let i ← i + 1
if i = dĺžka P **then** **return** m
else
let m ← m + i - π[i],
if i > 0 **then** **let** i ← π[i]

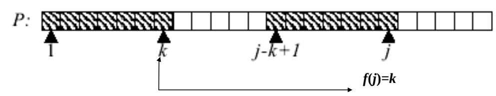
KMP algoritmus – pomocná tabuľka

```
algorithm kmp_table:
input:   pole znakov P (vzor, ktorého výskyt sa hľadá)
output:  pole celých čísiel π (tabuľka, ktorú treba vyplniť)

define variables:
         integer i ← 2 (súčasné miesto v π, pre ktoré sa počíta hodnota π)
         integer j ← 0 (pozícia znaku v P, počítajúc od nuly, ktorý je
nasledujúcim znakom v súčasnom podreťazci, ktorý je kandidátom na
zhodu)

let π[0] ← -1, π[1] ← 0
while i < dĺžka P, do:
         (prvý prípad: podreťazec pokračuje)
         if P[i - 1] = P[j]
                  then let π[i] ← j + 1, i ← i + 1, j ← j + 1
         (druhý prípad: nepokračuje, ale možno sa vrátiť)
                  else if j > 0 then let j ← π[j]
         (tretí prípad: nie sú ďalší kandidáti. Všimnime si, že j = 0)
                           else let π[i] ← 0, i ← i + 1
```


definícia funkcie počítajúcej predponu

$f(j)$= najväčšie $k < j$ také, že $P_{1,k}=P_{j-k+1,j}$

$f(j)$=0 ak také k neexistuje



j	1	2	3	4	5	6	7	8	9	10	11	12
P:	A	T	C	A	C	A	T	C	A	T	C	A
f:	0	0	0	1	0	1	2	3	4	2	3	4

príklad výpočtu hodnoty funkcie, počítajúcej predponu

j	1	2	3	4	5	6	7	8	9	10	11	12
P:	A	T	C	A	C	A	T	C	A	T	C	A
f:	0	0	0	1	0	1	2	3	4	2	3	4

urči f(5)

$$f(4)=1,\quad \text{teda}\ P_4=P_1$$
$$\text{ak}\ P_5=P_2,\ \text{tak dostaneme}\ f(5)=f(4)+1;$$
$$\text{ak}\ P_5\neq P_2,\ \text{tak skontrolujeme, či}\ P_5=P_1;$$
$$\text{pretože}\ P_5\neq P_1,\ \text{dostaneme}\quad f(5)=0$$

príklad výpočtu hodnoty funkcie, počítajúcej predponu

predpokladajme, že sa našlo $f(8)$=3.

kvôli určeniu $f(9)$:

j	1	2	3	4	5	6	7	8	9	10	11	12
P:	A	T	C	A	C	A	T	C	A	T	C	A
f:	0	0	0	1	0	1	2	3	4	2	3	4

$$f(8)=3\ \text{znamená}\ P_{6,8}=P_{1,3}$$
$$\text{teraz máme}\ P_9=P_4$$
$$\text{preto}\ f(9)=f(8)+1=4$$

príklad výpočtu hodnoty funkcie, počítajúcej predponu

kvôli určeniu $f(10)$:

j	1	2	3	4	5	6	7	8	9	10	11	12
P:	A	T	C	A	C	A	T	C	A	T	C	A
f:	0	0	0	1	0	1	2	3	4	2	3	4

$f(4)=1$

$f(9)=4$ pretože $P_9=P_{f(9-1)+1}=P_4$

$f(4)=1$ pretože $P_4=P_{f(4-1)+1}=P_1=$"A"

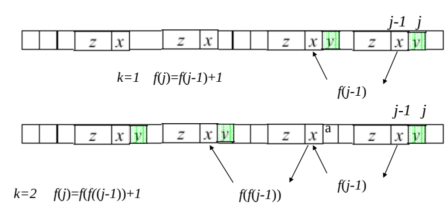
$f(10)=2$ pretože "T"$=P_{10}\neq P_{f(10-1)+1}=P_5=$"C"

$$P_{10}=P_{f^2(10-1)+1}=P_{f(f(10-1))+1}=P_{f(4)+1}=P_2=\text{"T"}$$

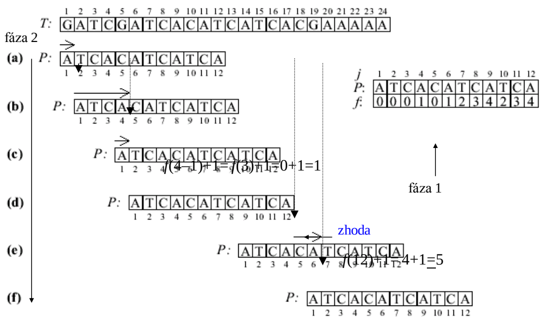
algoritmus pre funkciu predpona

$f(j)=f^k(j-1)+1$ if $j>1$ a existuje najmenšie $k\geq 1$ také, že $P_j=P_{f^k(j-1)+1}$

$f(j)=0$ inak



KMP algoritmus - príklad

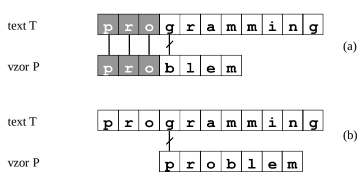


KMP algoritmus – časová zložitosť

- časová zložitosť: $O(m+n)$
  - $O(m)$ výpočet funkcie f
  - $O(n)$ hľadanie vzoru P

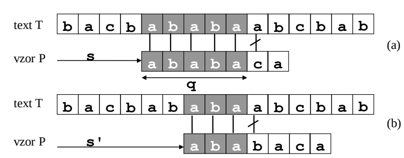
Knuthov-Morrisov-Prattov algoritmus

Porovnávanie vzoru s reťazcom začína na prvom znaku zľava (vzor je zarovnaný s reťazcom). Algoritmus postupuje, kým nenarazí na nezhodu na štvrtej pozícii medzi znakmi **b** a **g** (obr. a). Z predchádzajúcich znakov okamžite vieme, že posun vzoru o jeden alebo dva znaky nemá význam. Preto nastane posun o tri znaky. Tým sa vzor zarovná s textom nad znakom, kde nastala nezhoda. Od tohto miesta môže ďalej pokračovať porovnávane.



Knuthov-Morrisov-Prattov algoritmus

- V tomto príklade vidíme, že vzor je posunutý o ***s** a nastáva nové porovnávanie. Pri ňom sa zistilo, že nezhoda nastala na 6. pozícii reťazca, čo indikuje posun o 5 znakov (q). V tomto prípade však takýto posun nie je možný. Posunúť sa môžeme iba o 2 znaky, pretože na 3. znaku sa nachádza zhoda medzi týmto znakom a prvým znakom vzoru (na tomto mieste môže začínať vzor)*



Knuthov-Morrisov-Prattov algoritmus

- Posun pri prehľadávaní je nezávislý od prehľadávaného reťazca. Jeho veľkosť určuje tzv. predponová funkcia.
- predponová funkcia (Prefix function)

  π pre P, |P| = m:

  $\pi : \{1, 2, . . . ,m\} \rightarrow \{0, 1, . . . ,m-1\}$

  $\pi(q) = \max\{k : k < q, P_k \sqsupset P_q\}$.
- Jednotlivé posuny sa vypočítavajú vo fáze predspracovania.

Knuthov-Morrisov-Prattov algoritmus

```
KMP POROVNANIE(T, P)
n ← length(T)
m ← length(P)
π ← PREDPONOVA FUNKCIA(P)
q ← 0
for i ← 1 to n        //prehľadávaj další zľava doprava
        do while q > 0 and P[q + 1] ≠ T[i]
                do q ← π[q]        //nezhoduje sa
           if P[q + 1] = T[i]
                then q ← q + 1     //zhoduje sa
           if q = m
                then print "Vzor sa v retazci vyskytuje s posunom " i-m
                        q ← π[q]
```

Knuthov-Morrisov-Prattov algoritmus

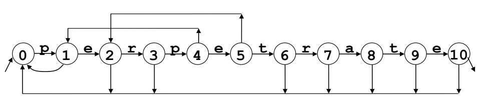
PREDPONOVA FUNKCIA (*P*)
// funkcia sa reprezentuje tabuľkou π, jej hodnoty sa tu vypočítavajú a zapisujú do π
m ← length(*P*)
π[1] ← 0
k ← 0
for *q* ← 2 **to** *m*
  do while *k* > 0 and *P*[*k* + 1] ≠ *P*[*q*]
    do *k* ← π[*k*]
  if *P*[*k* + 1] = *P*[*q*]
    then *k* ← *k* + 1
  π[*q*] ← *k*
return π

KMP algoritmus a konečný automat

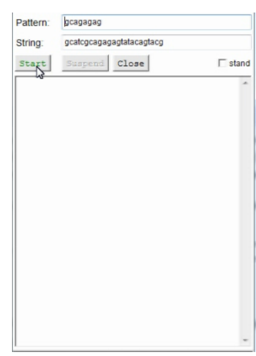
- KMP algoritmus do určitej miery súvisí s konečnými automatmi. Predpokladejme, že máme vzor P dĺžky m. Definujeme si konečný automat, ktorý bude mať m+1 stavov. Prechody medzi jednotlivými stavmi budú postupne určené jednotlivými písmenami vzoru. Teda napr. prechod mezi nultým a prvným stavom bude podľa písmena p_1, prechod mezi prvným a druhým stavom podľa p_2 atď. Ostatné prechody (teda akési chybové) bude určovať práve predponová funkcia. Vstupným stavom bude stav 0 a výstupným stav m. Samotné vyhľadávanie bude realizované ako práca takéhoto automatu so vstupom, ktorý odpovedá zadanému reťazcu. Rozdiel je iba v tom, že pokiaľ sa pomocou predponovej funkcie vrátime do niektorého predchádzajúceho stavu, okamžite skúsime cez ten istý znak (ktorý spôsobil nezhodu) prejsť do nasledujúceho stavu.

KMP algoritmus a konečný automat

- Príklad konečného automatu pre vzor perpetrate



Knuthov-Morrisov-Prattov algoritmus



Knuthov-Morrisov-Prattov algoritmus

```
void preKmp(char *x, int m, int kmpNext[])
{
    int i, j; i = 0; j = kmpNext[0] = -1;
    while (i < m)
    {
        while (j > -1 && x[i] != x[j])
            j = kmpNext[j];
        i++;
        j++;
        if (x[i] == x[j])
            kmpNext[i] = kmpNext[j];
        else
            kmpNext[i] = j;
    }
}
```


Knuthov-Morrisov-Prattov algoritmus

```
void KMP(char *x, int m, char *y, int n)
{
    int i, j, kmpNext[XSIZE];
    /* Preprocessing */
    preKmp(x, m, kmpNext);
    /* Searching */
    i = j = 0;
    while (j < n)
    {
        while (i > -1 && x[i] != y[j])
            i = kmpNext[i];
        i++;
        j++;
        if (i >= m)
        {
            OUTPUT(j - i);
            i = kmpNext[i];
        }
    }
}
```

Boyerov-Moorov algoritmus - príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	B	C	E	F	G	A	B	C	D	E

p[0]	p[1]	p[2]	p[3]
A	B	C	D

N

Vo vzore nie je žiadny znak E: z toho plynie, že vzor nemožno nájsť nikde pred znakom t[3]. Preto sa možno a teda aj treba posunúť o 4 miesta doprava.

Boyerov-Moorov algoritmus - príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	B	C	E	F	G	A	B	C	D	E

p[0]	p[1]	p[2]	p[3]
A	B	C	D

N

Nevyskytuje sa. Ale vo vzore sa B vyskytuje. Treba sa posunúť o dve miesta doprava.

Boyerov-Moorov algoritmus - príklad

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	B	C	E	F	G	A	B	C	D	E

p[0]	p[1]	p[2]	p[3]
A	B	C	D

Y Y Y Y

Boyerov-Moorov algoritmus

- porovnáva sprava doľava
- 2 predvypočítané funkcie
  - posun o dobrú príponu
  - posun o zlý znak

Boyerov-Moorov algoritmus

myšlienka č. 1: porovnávať sprava doľava

```
        12345678901234567
T:      xpbctbxabpqxctbpq
P:        tpabxab
```


Boyerov-Moorov algoritmus

```
        12345678901234567
T:      spbctbsabpqsctbpq
P:        tpabsab
P:          tpabxab
```

myšlienka č. 2: pravidlo zlého znaku

R(x): najpravejší výskyt znaku x v P. R(x)=0 ak sa x nevyskytuje. R(t)=1, R(s)=5.

i: miesto, kde došlo k nezhode v P. i=3

k: zodpovedajúce miesto v T. k=5. T[k]=t

pravidlo zlého znaku: P sa posunie doprava o max{1, i-R(T[k])}. čiže ak je najpravejší výskyt znaku T[k] v P na mieste j (j<i), tak P[j] sa ocitne pod T[k] po posunutí.

Boyerov-Moorov algoritmus

- myšlienkou pravidla zlého znaku je posunúť P o viac než o jeden znak, ak je to možné.
- pravidlo neúčinné, ak j>i
- bohužiaľ, prípad j>i je častý

```
           12345678901234567
T:         spbctbsatpqsctbpq
P:           tpabsat
P:       tpabsat
```


Boyerov-Moorov algoritmus

označme x=T[k] znak, ktorý sa nezhoduje so vzorom v T.

myšlienka č. 3: rozšírené pravidlo zlého znaku: P sa posunie doprava tak, že najbližšie x vľavo od miesta i v P bude pod T[K].

```
           12345678901234567
T:         spbctbsatpqsctbpq
P:           tpabsat
P:             tpabsat
```


Boyerov-Moorov algoritmus

aby sme mohli používať rozšírené pravidlo zlého znaku, potrebujeme poznať:
pre každé miesto i v P, pre každý znak x v abecede, miesto najbližšieho výskytu x vľavo od miesta i.

možný prístup: dvojrozmerným poľom: n*| Σ |

pamäťovo a časovo príliš náročné

Boyerov-Moorov algoritmus

iný prístup: prezrieť P sprava doľava a pre každý znak x si držať zoznam miest, kde sa x vyskytuje (v klesajúcom poradí).

```
P:  tpabsat
    t→7,1
    a→6,3  …
```

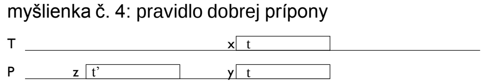
ak dôjde k nezhode medzi P[i] a T[k], (označme x=T[k]), prezri zoznam výskytov znaku x, nájdi prvé číslo (označme ho j), ktoré je menšie než i a posuň P doprava tak, aby P[j] sa dostalo pod T[k].

```
           12345678901234567
T:         spbctbsatpqsctbpq
P:           tpabsat
P:             tpabsat
```

ak také číslo j neexistuje, tak posuň P za T[k]
čas a pamäť: lineárna

Boyerov-Moorov algoritmus

myšlienka č. 4: pravidlo dobrej prípony

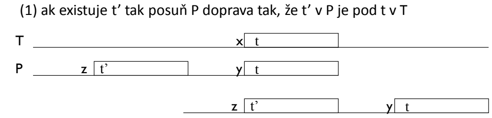


t je prípona vzoru P taká, že sa zhoduje s podreťazcom t textu T
x≠y
t' je najpravejší výskyt t v P taký, že t' nie je príponou P a z≠y

Boyerov-Moorov algoritmus

pravidlo dobrej prípony:

(1) ak existuje t' tak posuň P doprava tak, že t' v P je pod t v T




```
    123456789012345678
T:  prstabstubabvqxrst
P:    qcabdabdab
P:          qcabdabdab
P:       qcabdabdab
```

Boyerov-Moorov algoritmus

rozšírené pravidlo zlého znaku sa sústreďuje na znaky.
pravidlo dobrej prípony sa sústreďuje na podreťazce.

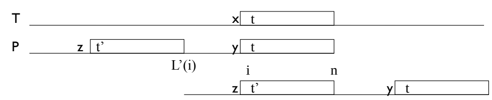
ako teraz získať informáciu, ktorú treba pre pravidlo dobrej prípony? ako pre nejaké t nájsť t`?

Boyerov-Moorov algoritmus

L’(i): pre každé i, L’(i) je najväčšie miesto menšie než n také, že podreťazec P[i,…,n] sa zhoduje s príponou reťazca P[1,…, L’(i)] a navyše znak predchádzajúci tejto prípone je rôzny od P[i-1].

ak také miesto neexistuje tak L’(i) =0.

nech t= P[i,…,n], potom L’(i) je miesto pravého konca t’.




```
T:  prstabstubabvqxrst
P:     qcabdabdab
       1234567890
L’(9)=4, L’(10)=0, L’(8)=?, L’(7)=? L’(6)=?
```

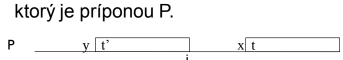

Boyerov-Moorov algoritmus

nech t= P[i,…,n], potom L’(i) je miesto pravého konca t’.

aby sa dalo používať pravidlo dobrej prípony, treba poznať L’(i) pre všetky i=1,…,n.

pre vzor P:

N_j je dĺžka najdlhšieho podreťazca, ktorý končí na mieste j a ktorý je príponou P.

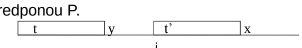


t=t’;
j=|t’|=|t|;
x≠y

Boyerov-Moorov algoritmus

N_j : dĺžka najdlhšieho podreťazca, ktorý končí na mieste j a ktorý je príponou P.

Z_i: dĺžka najdlhšieho podreťazca, ktorý začína na mieste i a ktorý je predponou P.



Boyerov-Moorov algoritmus

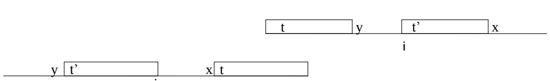
N je obrátené Z!

P^r je obrátený reťazec P

všimnime si, že $N_j(P)=Z_{n-j+1}(P^r)$

	1	2	3	4	5	6	7	8	9	0
P:	q	c	a	b	d	a	b	d	a	b
N_j:	0	0	0	2	0	0	5	0	0	0

	1	2	3	4	5	6	7	8	9	0
P^r:	b	a	d	b	a	d	b	a	c	q
Z_i:	0	0	0	5	0	0	2	0	0	0



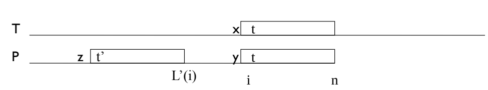
Boyerov-Moorov algoritmus

pre daný vzor P,

N_j (pre j=1,…,n) sa dá vypočítať v lineárnom čase O(n) algoritmom Z.

prečo treba určiť N_j ?

aby sa dalo použiť pravidlo dobrej prípony, treba poznať L’(i) pre všetky i=1,…,n.

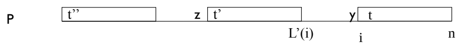


L’(i) sa dá získať z N_j!

Boyerov-Moorov algoritmus

Pre miesto i, nech t=P[i,…n].

L'(i) je najväčšie miesto j menšie než n také, že N_j=|t|



	1	2	3	4	5	6	7	8	9	0
P:	q	c	a	b	d	a	b	d	a	b
N_j:	0	0	0	2	0	0	5	0	0	0
L'(i):	0	0	0	0	0	7	0	0	4	0

	1	2	3	4	5	6	7	8	9	0
P^r:	b	a	d	b	a	d	b	a	c	q
Z_i:	0	0	0	5	0	0	2	0	0	0

Boyerov-Moorov algoritmus

Ako získať L'(i) z N_i v lineárnom čase?

vstup: vzor P

výstup: L'(i) pre i=1,…,n

Algoritmus

vypočítaj N_j pre j=1,…,n na základe algoritmu Z

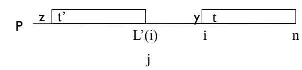
for i=1; i<=n; i++

L'(i)=0;

for j=1; j<n; j++

i=n-N_j+1

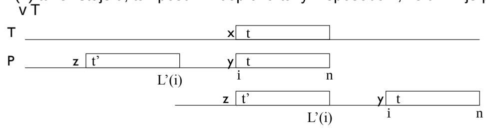
L'(i)=j;



Boyerov-Moorov algoritmus

Pravidlo dobrej prípony:

(1) ak existuje t', tak posuň P doprava takým spôsobom, že t' v P je pod t v T



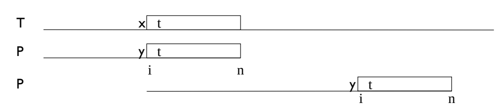

```
   123456789012345678
T: prstabstubabvqxrst
P:   qcabdabdab        i=9; L'(9)=4

P:         qcabdabdab
```


Boyerov-Moorov algoritmus

Pravidlo dobrej prípony:

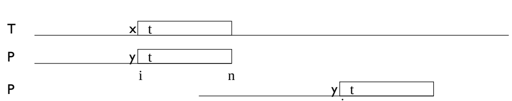
(1) Ak nastáva nezhoda na mieste i-1 v P a L'(i)>0 (t.j. existuje t'), tak podľa pravidla dobrej prípony možno posunúť P o n-L'(i) miest doprava.

(2) Čo ak nastane nezhoda na mieste i-1 v P a L'(i)=0 (t.j. t' neexistuje)? Možno P posunúť takto



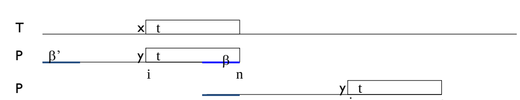
Boyerov-Moorov algoritmus

Dá sa však spraviť viac!



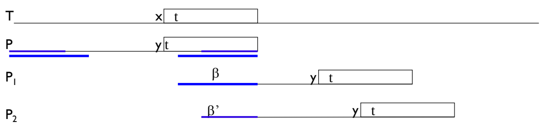
Boyerov-Moorov algoritmus

Pozorovanie 1: ak β je predponou P a je tiež príponou P, tak…



Boyerov-Moorov algoritmus

pozorovanie 2: ak je viac kandidátov β, tak posuň P o čo najmenšiu dĺžku

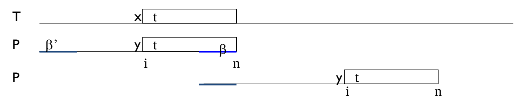


Boyerov-Moorov algoritmus

Pravidlo dobrej prípony: keď nastane nezhoda na mieste i-1 vzoru P

(1) ak L'(i)>0 (t.j. t' existuje), tak podľa pravidla dobrej prípony možno posunúť P o n-L'(i) miest doprava.

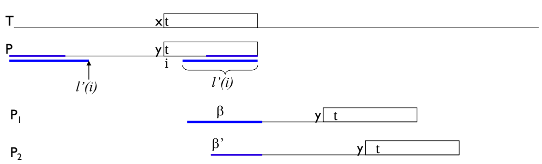
(2) inak ak L'(i)=0 (t.j. t' neexistuje)? Možno posunúť P poza ľavý koniec t o najmenší počet miest taký, že predpona posunutého vzoru sa zhoduje s príponou t.



Boyerov-Moorov algoritmus

l'(i) : dĺžka najväčšej prípony P[i,…,n] takej, že je aj predponou vzoru P. Ak taká neexistuje, tak *l'(i)=0.*

l'(i) je dĺžka prekrytia medzi neposunutým a posunutým vzorom.

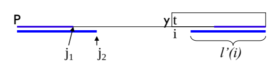


Boyerov-Moorov algoritmus

l'(i) sa rovná najväčšiemu *j*≤|*P[i,…n]*| takému, že $N_j=j$

1. $N_j=j$ tak β je predponou P a tiež príponou P

2. a chceme najväčšie j



Boyerov-Moorov algoritmus

l'(i) sa rovná najväčšiemu *j*≤|*P[i,…n]*| takému, že $N_j=j$

	1	2	3	4	5	6	7	8	9	0
P:	a	b	d	a	b	a	b	d	a	b
N_j:	0	2	0	0	5	0	2	0	0	0
***l'(i)*:**	5	5	5	5	5	5	2	2	2	0

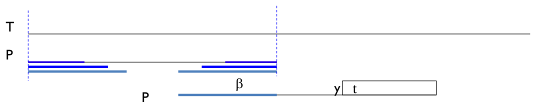
Boyerov-Moorov algoritmus

Ako vypočítať *l'(i)* z N_j v lineárnom čase?

Boyerov-Moorov algoritmus

Čo ak sa nájde zhoda?

posuň P o 1 miesto…ale…

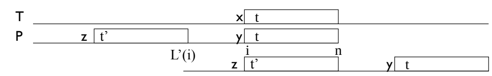


Posuň P o najmenší počet miest taký, že predpona posúvaného vzoru sa zhoduje s t, t.j. posuň P doprava o n-*l*'(2)

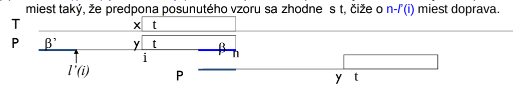
Boyerov-Moorov algoritmus

Pravidlo dobrej prípony: keď nastane nezhoda na mieste i-1 vzoru P

(1) ak L'(i)>0 (t.j. t' existuje), tak podľa pravidla dobrej prípony možno posunúť P o n-L'(i) miest doprava.



(2) Inak ak L'(i)=0 (t.j. t' neexistuje)? Možno posunúť P poza ľavý koniec t o najmenší počet miest taký, že predpona posunutého vzoru sa zhodne s t, čiže o n-*l*'(i) miest doprava.



(3)Ak sa nájde nezhoda, tak posuň P doprava o n-*l*'(2)

Boyerov-Moorov algoritmus

rozšírené pravidlo zlého znaku vs. pravidlo dobrej prípony

```
     123456789012345678
T:   prstabstububabvqxrst
P:      qcabdabdab
           P:     qcabdabdab
  P:      qcabdabdab
```

```
     123456789012345678
T:   prstabstuqabvqxrst
P:      qcabdabdab
        P:     qcabdabdab
          P:     qcabdabdab
```

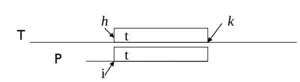

Boyerov-Moorov algoritmus

Posuň P o najväčší počet miest určený niektorým z oboch pravidiel. To je podstata Boyerovho-Moorovho algoritmu!

vstup: text T, vzor P; výstup: nájdi výskyty vzoru P v T

Algoritmus **Boyer-Moore**

```
    vypočítaj L'(i), L`(i) a R(x)
    k=n;
    while (k≤m) do
        i=n
        h=k
        while i>0 and P[i]=T[h] do
                i--;
                h--;
        if i=0
                oznám výskyt vzoru P v T končiaci na mieste k;
                k=k+n-l`(2)
        else posuň P (zvýš k) o väčší počet miest z počtu určeného rozšíreným
    pravidlom zlého znaku a počtu určeného pravidlom  dobrej prípony.
```



Boyerov-Moorov algoritmus

- výkonnosť závisí od dĺžky vzoru
- O(n/m)
- Dlhšie vzory = lepšia výkonnosť
- Najmenší vzor = m = 1
  O(n) – lineárne hľadanie

porovnávanie pomocou automatu

- Na základe vzoru sa vytvorí minimálny deterministický konečný automat, pomocou ktorého sa rozpoznáva vzor v zadanom reťazci.

porovnávanie pomocou automatu

Def.: Konečný automat (finite automaton) je usporiadaná 5-tica
(Q, q_0, A, Σ, δ), kde
- Q je konečná množina stavov
- q_0 počiatočný stav
- A množina koncových stavov (akceptujúce)
- Σ je vstupná abeceda
- δ je tzv. prechodová funkcia z $Q \times \Sigma$ do Q.

Rozšírenie δ funkcie - $\delta_* : Q \times \Sigma_* \rightarrow Q$ je definované induktívne:
$\delta_*(q, \varrho) = q$
$\delta_*(q, wa) = \delta(\delta_*(q, w), a)$

79

porovnávanie pomocou automatu

funkcia koncového stavu - vracia stav automatu po spracovaní nejakého slova

príponová funkcia pre P, $|P| = m$ je
$\sigma : \Sigma_* \rightarrow \{0, 1, . . . ,m\}$
definovaná ako
$\sigma(x) = \max\{k : P_k \sqsupset x\}$,
kde $u \sqsupset v$ znamená, že u je príponou v a $P_k = P[1..k]$.

80

porovnávanie pomocou automatu

Definícia automatu: pre P, $|P| = m$
$Q = \{0, 1, . . . ,m\}$, $q_0 = 0$, $A = \{m\}$, $\delta(q, a) = \sigma(P_q a)$.

Vždy, keď sa počas simulácie vstupného slova T na automate dostane automat do stavu m, našiel sa podvýraz P a jeho posun je daný aktuálnym miestom v reťazci zmenšeným o m.

Platí:
- Pre každý reťazec x a znak a platí: $\sigma(xa) \leq \sigma(x) + 1$.
- Pre každý reťazec x a znak a, ak $q = \sigma(x)$, tak $\sigma(xa) = \sigma(P_q a)$

81

porovnávanie pomocou automatu

```
POROVNANIE KONEČNÝM AUTOMATOM(T, δ, m)
n ← dĺžka(T)
q ← 0
for i ← 1 to n
        do q ← δ(q, T[i])
                if q = m
                then
                   print "Vzor sa v retazci vyskytuje s posunom " i-m
```

82

porovnávanie pomocou automatu

- VÝPOČET PRECHODOVEJ FUNKCIE (P, Σ)

```
m := |P|
     for q := 0 to m do
            for každý symbol a ∈ Σ do
                   k := min(m+1, q+2)
                   repeat k := k − 1
                   until P_k ⊐ P_q a
                   δ(q, a) := k
return δ
```

- Zložitosť algoritmu:
  - Preprocesná fáza: $O(m\ |\Sigma|)$
  - Fáza vyhľadávania: $O(n)$

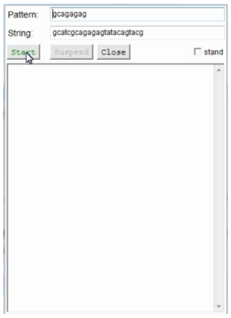
83

porovnávanie pomocou automatu

- Zložitosť algoritmu:
  - fáza predspracovania: $O(m.|\Sigma|)$
  - fáza vyhľadávania: $O(n)$

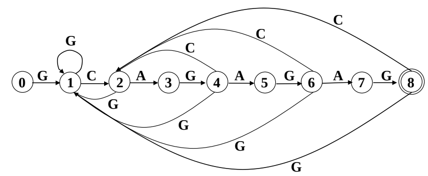
84

porovnávanie pomocou automatu



porovnávanie pomocou automatu

- DFA z príkladu:



Rabinov-Karpov algoritmus

- Namiesto priameho porovnávania reťazca so vzorom sa porovnavajú výstupy hešovacích funkcií. Porovnáva sa heš vzoru s hešom vybraného podreťazca (vyberá sa podreťazec taký dlhý ako je dĺžka vzoru). Pokiaľ sa heše zhodujú, uskutočnuje sa porovnávanie jednotlivých znakov.

Rabinov-Karpov algoritmus – príklad 1/1

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	C	D	E	A	C	A	C	C	D	E

← Hash(ACDE) = 10 →

p[0]	p[1]	p[2]	p[3]
A	C	D	B

← Hash(ACDB) = 5 →

Rabinov-Karpov algoritmus – príklad 1/2

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	C	D	E	A	C	A	C	C	D	E

← Hash(CDEA) = 6 →

p[0]	p[1]	p[2]	p[3]
A	C	D	B

← Hash(ACDB) = 5 →

Rabinov-Karpov algoritmus

```
algorithm RabinKarp:
Input    pole znakov T, dĺžka n
         pole znakov P, dĺžka m

    hP := hash(P[1..m])
    hT := hash(T[1..m])
    for i from 1 to n-m+1
        if hT = hP
            if T[i..i+m-1] = P
                return i
        hT := hash(T[i+1..i+m])
    return nenašiel sa výskyt
```

Rabinov-Karpov algoritmus – príklad 2/1

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	C	D	E	A	C	A	C	C	D	E

← Hash(ACDE) = 10 →

p[0]	p[1]	p[2]	p[3]
A	C	D	B

← Hash(ACDB) = 5 →

j[0]	j[1]	j[2]	j[3]
F	M	D	T

← Hash(FMDT) = 62 →

Rabinov-Karpov algoritmus – príklad 2/2

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
A	C	D	E	A	C	A	C	C	D	E

← Hash(CDEA) = 6 →

p[0]	p[1]	p[2]	p[3]
A	C	D	B

← Hash(ACDB) = 5 →

j[0]	j[1]	j[2]	j[3]
F	M	D	T

← Hash(FMDT) = 62 →

Rabinov-Karpov algoritmus

- Hešovacia funkcia by mala mať tieto vlastnosti:
  - jednoducho vypočítateľná
  - s malou pravdepodobnosťou kolízií
  - Heš posunutého podreťazca by mal byť jednoducho odvoditeľný z predchádzajúceho hešu (táto vlastnosť výrazne uľahčí výpočet a algoritmus sa tým stáva omnoho efektívnejší ako naivný)

Rabinov-Karpov algoritmus

- Hešovacia funkcia:

Predpokladáme, že nahradíme reťazec *M* znakov určitým celým číslom. Ak použijeme konštantu *b* - maximálny počet možných znakov v abecede, tak definujeme:

- $x = t[i]b^M + t[i+1]b^{M-1} + ... + t[i+M]$

Pokročme v texte o jeden znak dopredu a hodnota x' bude:

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

ak preskúmame x a x' , tak zistíme, že:

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

Rabinov-Karpov algoritmus

Hešovacia funkcia:

- Tretej požiadavke vyhovuje napríklad hešovacia funkcia definovaná ako polynóm (m-1). stupňa, kde hodnoty znakov vystupujú ako koeficienty. Aby sme sa vyhli problémom s príliš veľkými číslami pri výpočtoch, používa sa modulo aritmetika:
- $pathash = (f^{m-1} \text{ord}(pat_0) + f^{m-2} \text{ord}(pat_1) + ... + f \text{ord}(pat_{m-2}) + \text{ord}(pat_{m-1})) \bmod p$
- $texthash_i = (f^{m-1} \text{ord}(text_i) + f^{m-2} \text{ord}(text_{i+1}) + ... + f \text{ord}(text_{i+m-2}) + \text{ord}(text_{i+m-1})) \bmod p$
- $texthash_{i+1} = (f^{m-1} \text{ord}(text_{i+1}) + f^{m-2} \text{ord}(text_{i+2}) + ... + f \text{ord}(text_{i+m-1}) + \text{ord}(text_{i+m})) \bmod p$

$= \boldsymbol{(f (texthash_i - f^{m-1} \text{ord}(text_i)) + \text{ord}(text_{i+m})) \bmod p}$

- Hešovaciu funkciu ovplyvňujú parametre f a p.

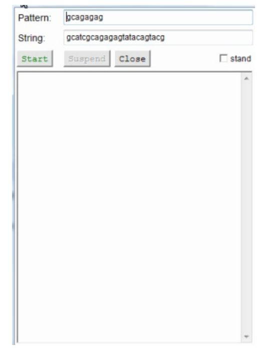
Rabinov-Karpov algoritmus

```
RABIN-KARP POROVNANIE(T, P, d, q)
n ← dĺžka(T)
m ← dĺžka(P)
h ← d^(m-1) mod q
p ← 0
t_0 ← 0
for i ← 1 to n        //spracovanie
          do p ← (dp + P[i]) mod q
            t_0 ← (dt_0 + T[i]) mod q
for s ← 0 to n - m             //párovanie
          do if p = t_s
                    then if P[1..m]=T[s+1..s+m]
                       then print "Vzor sa v retazci vyskytuje s posunom“ s
            if s < n-m
                    then t_(s+1) ← (d(t_s-T[s+1]h)+T[s+m+1]) mod q
```

Rabinov-Karpov algoritmus

- Zložitosť algoritmu:
  - fáza predspracovania má časovú zložitosť ***O***(*m*)
  - vyhľadavacia fáza má časovú zložitosť ***O***(*mn*)
  - Očakávaná doba behu algoritmu je ***O***(*n*+*m*)

Rabinov-Karpov algoritmus



Rabinov-Karpov algoritmus

- C++:

```
#define REHASH(a, b, h) ((((h) - (a)*d) << 1) + (b))
void KR(char *x, int m, char *y, int n)
{
        int d, hx, hy, i, j;
        /* Preprocessing */
        /* computes d = 2^(m-1) with the left-shift operator */
        for (d = i = 1; i < m; ++i)
                d = (d<<1);
        for (hy = hx = i = 0; i < m; ++i)
        {
                hx = ((hx<<1) + x[i]);
                hy = ((hy<<1) + y[i]);
        }
        /* Searching */
        j = 0;
        while (j <= n-m)
        {
                if (hx == hy && memcmp(x, y + j, m) == 0)
                        OUTPUT(j);
                hy = REHASH(y[j], y[j + m], hy);
                ++j;
        }
}
```


časová efektívnosť

- Jeden vzor
  - BM O(n/m)
  - KMP O(n)
  - RK O(mn)
- Viac vzorov
  - BM, KMP O(n.k)
  - RK O(n + k)

aplikácie

- BM - textové editory – search/replace
- Karp-Rabin – vyhľadávanie plagiátov

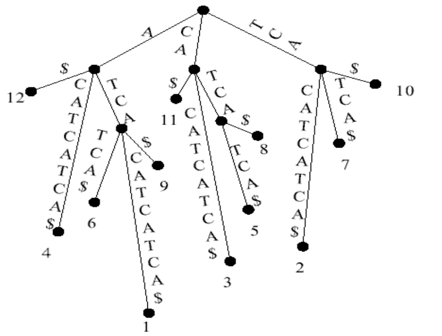
prípony

- Prípony reťazca *T*=“`ATCACATCATCA`”

`ATCACATCATCA`	$T_{(1)}$
`TCACATCATCA`	$T_{(2)}$
`CACATCATCA`	$T_{(3)}$
`ACATCATCA`	$T_{(4)}$
`CATCATCA`	$T_{(5)}$
`ATCATCA`	$T_{(6)}$
`TCATCA`	$T_{(7)}$
`CATCA`	$T_{(8)}$
`ATCA`	$T_{(9)}$
`TCA`	$T_{(10)}$
`CA`	$T_{(11)}$
`A`	$T_{(12)}$

príponový strom

- príponový strom reťazca T="ATCACATCATCA"



vlastnosti príponového stromu

- každá hrana je ohodnotená nejakým podreťazcom reťazca *T*.
- každý vnútorný vrchol na najmenej dvoch potomkov.
- každá prípona $T_{(i)}$ má svoju ohodnotenú cestu z koreňa do listu pre $1 \leq i \leq n$.
- príponový strom má n listov.
- hrany vychádzajúce z toho istého vrchola majú ohodnotenia, ktoré sa určite nezačínajú tým istým znakom.

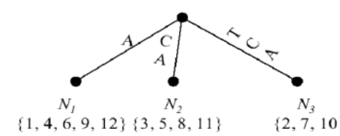
algoritmus vytvorenia príponového stromu

krok 1: rozdeľ všetky prípony do skupín podľa prvého znaku a vytvor vrchol.

krok 2: pre každú skupinu: ak obsahuje len jednu príponu, tak vytvor listový vrchol a hranu ohodnotenú touto príponou, inak nájdi najdlhšiu spoločnú predponu medzi príponami v tejto skupine a vytvor hranu vychádzajúcu z vrchola ohodnotenú najdlhšou spoločnou predponou. Vymaž túto predponu zo všetkých prípon v tejto skupine.

krok 3: opakuj predchádzajúci postup pre každý vrchol, ktorý nie je ukončený.

príklad vytvorenia príponového stromu



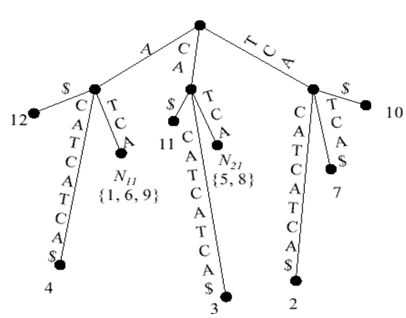
- T="ATCACATCATCA".
- začiatočné znaky: "A", "C", "T"
- v N_3,

  $T(2)$ ="TCACATCATCA"

  $T(7)$ ="TCATCA"

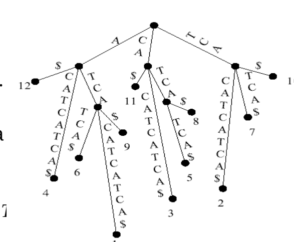
  $T(10)$ ="TCA"
- najdlhšia spoločná predpona N_3 je "TCA"

- T="ATCACATCATCA".
- druhá rekurzia:



nájdenie podreťazca pomocou príponového stromu

- *T* = "ATCACATCATCA"
- *P* ="TCAT"
  - *P* je na mieste 7 v *T*.
- *P* ="TCA"
  - *P* je na mieste 2, 7 a v *T*.
- *P* ="TCATT"
  - *P* sa nenachádza v



príponový strom – časová zložitosť

- Príponový strom pre textový reťazec T dĺžky *n* sa dá zostrojiť v čase O(*n*) time (pomocou zložitého algoritmu).
- Vyhľadanie vzoru *P* dĺžky *m* v príponovom strome vyžaduje O(*m*) porovnaní.
- Hľadanie presného výskytu reťazca: O(*n*+*m*) time

109

Príponové pole

- V príponovom poli sú všetky prípony reťazca T v neklesajúcom lexikálnom poradí.
- napr. T="ATCACATCATCA"

i	1	2	3	4	5	6	7	8	9	10	11	12
A	12	4	9	1	6	11	3	8	5	10	2	7

4	ATCACATCATCA	$T_{(1)}$
11	TCACATCATCA	$T_{(2)}$
7	CACATCATCA	$T_{(3)}$
2	ACATCATCA	$T_{(4)}$
9	CATCATCA	$T_{(5)}$
5	ATCATCA	$T_{(6)}$
12	TCATCA	$T_{(7)}$
8	CATCA	$T_{(8)}$
3	ATCA	$T_{(9)}$
10	TCA	$T_{(10)}$
6	CA	$T_{(11)}$
1	A	$T_{(12)}$

1	A	$T_{(12)}$
2	ACATCATCA	$T_{(4)}$
3	ATCA	$T_{(9)}$
4	ATCACATCATCA	$T_{(1)}$
5	ATCATCA	$T_{(6)}$
6	CA	$T_{(11)}$
7	CACATCATCA	$T_{(3)}$
8	CATCA	$T_{(8)}$
9	CATCATCA	$T_{(5)}$
10	TCA	$T_{(10)}$
11	TCACATCATCA	$T_{(2)}$
12	TCATCA	$T_{(7)}$

110

Hľadanie v príponovom poli

- ak *T* sa reprezentuje príponovým poľom, vzor *P* sa dá nájsť v *T* v čase O(*m*.log*n*) binárnym vyhľadávaním.
- príponové pole sa dá určiť v čase O(*n*) lexikálnym hľadaním do hĺbky v príponovom poli.
- celkový čas: O(*n*+*m*.log*n*)

111

hľadanie približného výskytu reťazca

- Textový reťazec *T*, |*T*|=*n*

  vzor (reťazec) *P*, |*P*|=*m*

  k chýb, kde chybami môžu byť náhrada, vymazanie alebo zloženie znaku.
- napr:

  T ="pttapa", *P* ="patt", *k* =2,

  $T_{1,2}$, $T_{1,3}$, $T_{1,4}$ a $T_{5,6}$ sú všetko reťazce vzdialené nie viac než 2 chyby od vzoru *P*.

-112

vzdialenosť editovania prípony

- Nech sú dané 2 reťazce S_1 a S_2, vzdialenosť editovania prípony je najmenší počet náhrad, vložení a výmazov, ktoré prepíšu nejakú príponu S_1 do S_2.
- napr:
  - S_1="ptt" a S_2="pt". Vzdialenosť editovania prípony medzi S_1 a S_2 je 1.
  - S_1="ptt" a S_2="p". Vzdialenosť editovania prípony medzi S_1 a S_2 je 1.
  - S_1="pt" a S_2="patt". Vzdialenosť editovania prípony medzi S_1 a S_2 je 2.

113

vzdialenosť editovania prípony

- Nech *T* a *P* sú reťazce, ak aspoň jedna zo vzdialeností editovania prípony medzi $T_{1,1}$, $T_{1,2}$, …, $T_{1,n}$ a *P* nie je väčšia než *k*, tak P sa približne vyskytuje v T s chybou nie väčšou než *k*.
- napr: T ="pttapa", P ="patt", k=2
  - pre $T_{1,1}$="p" a P ="patt", vzdialenosť editovania prípony je 3.
  - pre $T_{1,2}$ ="pt" a P ="patt", vzdialenosť editovania prípony je 2.
  - pre $T_{1,5}$ ="pttap" a P ="patt", vzdialenosť editovania prípony je 3.
  - pre $T_{1,6}$ ="pttapa" a P ="patt", vzdialenosť editovania prípony je 2.

114

hľadanie približného výskytu reťazca

- riešenie metódami dynamického programovania
- nech $E(i,j)$ označuje vzdialenosť editovania prípony medzi $T_{1,j}$ a $P_{1,i}$.

$E(i, j) = E(i–1, j–1)$	if $P_i = T_j$
$E(i, j) = \min\{E(i, j–1), E(i–1, j), E(i–1, j–1)\}+1$	if $P_i \neq T_j$

hľadanie približného výskytu reťazca

- napr: T =“`pttapa`”, P =“`patt`”, k=2

T

			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	0	0	0	0	0	0
	1	p	1	0	1	1	1	0	1
P	2	a	2	1	1	2	1	1	0
	3	t	3	2	1	1	2	2	1
	4	t	4	3	2	1	2	3	2

Hľadanie najdlhšej spoločnej podpostupnosti

- Longest common subsequence (LCS) problém
  - Dané sú dve postupnosti x[1..m] a y[1..n]; máme nájsť najdlhšiu podpostupnosť, ktorá sa vyskytuje v oboch postupnostiach
  - Podpostupnosť: prvky v pôvodnej postupnosti nemusia byť nevyhnutne vedľa seba, ale ich poradie ostáva nezmenené
- x = {A B C B D A B}, y = {B D C A B A}
  - {B B A} je podpostupnosť oboch postupností **x** a **y**
- Algoritmus hrubej sily
  - Pre každú podpostupnosť v **x**, zisti či nie je podpostupnosťou **y**. Vráť najdlhšiu.
  - Koľko podpostupností je v **x**?
    - 2^m
  - Aká by bola časová náročnosť?
    - 2^m podpostupností **x** porovnať s ***n*** prvkami postupnosti **y**
    - $O(n\,2^m)$

Hľadanie najdlhšej spoločnej podpostupnosti

- Úloha: Porovnanie dvoch DNA reťazcov
- X = {A B C B D A B}, Y = {B D C A B A}
- Algoritmom hrubej sily porovnáme každú podpostupnosť X so znakmi v Y
  - X = A B C B D A B
  - Y = B D C A B A
- LCS problém má optimálnu subštruktúru: riešenie čiastkových problémov je časť konečného riešenia
- Čiastkový problém
  - Nájsť najdlhšiu spoločnú podpostupnosť párov prefixov X a Y
- Na vyriešenie tohto problému môžeme použiť dynamické programovanie!

Hľadanie najdlhšej spoločnej podpostupnosti

- V prvom rade nájdeme dĺžku LCS. Neskôr zmodifikujeme algoritmus pre nájdenie LCS samotnej.
- Definujeme X_i a Y_j ako predpony X a Y dĺžky *i* respektíve *j*
- *Definujeme c[i,j] ako dĺžku LCS X_i a Y_j*
- *Potom dĺžka LCS X a Y bude c[m,n]*
- *Rekurzívna definícia c[i,j]*

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & ak\ x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & inak \end{cases}$$

Definícia dĺžky najdlhšej spoločnej podpostupnosti

- Začneme s *i=j=0* (prázdna podpostupnosť x a y)
  - Pretože X_0 a Y_0 sú prázdne reťazce, ich LCS je vždy prázdna (c[0,0]=0)
  - LCS prázdnej postupnosti a nejakej inej postupnosti je pre každé i a j: c[0,j]=c[i,0]=0
- Pre výpočet c[i,j] sa rozhodujeme medzi dvoma prípadmi:
  - ***x[i] = y[j]***
    - Pri zhode symbolu v postupnostiach X a Y je dĺžka LCS X_i a Y_j rovnaká ako dĺžka LCS menšej postupnosti X_{i-1} a Y_{j-1}, plus 1
  - ***x[i] != y[j]***
    - Ak sa symboly nezhodujú dĺžka ostáva nezmenená *(max(c[i-1,j], c[i, j-1]))*

Algoritmus na nájdenie dĺžky najdlhšej spoločnej podpostupnosti

```
LCS DĹŽKA(X,Y)
m = length(X)
n = length(Y)
for i = 1 to m c[i,0] = 0        //X0
for i = 1 to m c[i,0] = 0        //Y0
for i = 1 to m        //pre všetky Xi
        for j = 1 to n        //pre všetky Yj
                if (Xi == Yi)
                        c[i,j] = c[i-1,j-1]+1
                else
                        c[i,j] = max(c[i-1,j], c[i,j-1])
return c
```

- Príklad: X = ABCB; Y = BDCAB
  - LCS(X, Y) = BCB
  - X = A B C B
  - Y = B D C A B

Najdlhšia spoločná podpostupnosť – príklad (inicializácia)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

ABCB
BDCAB

X = ABCB; m = |X| = 4
Y = BDCAB; n = |X| = 5
alokácia 2rozmerného poľa c[0..4, 0..5]

Najdlhšia spoločná podpostupnosť – príklad (1)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0					
2	**B**	0					
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

for *i* = 1 **to** m *c[i,0]* = 0
for *j* = 1 **to** n *c[0,j]* = 0

Najdlhšia spoločná podpostupnosť – príklad (2)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0	0				
2	**B**	0					
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (3)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0	0	0	0		
2	**B**	0					
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (4)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0	0	0	0	1	
2	**B**	0					
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]*+1
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (5)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1
2	**B**	0					
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

if (*Xi == Yi*)
c[i,j] = c[i-1,j-1]+1
else *c[i,j] = max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (6)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1
2	**B**	0	1				
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

if (*Xi == Yi*)
c[i,j] = c[i-1,j-1]+1
else *c[i,j] = max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (7)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1
2	**B**	0	1	1	1	1	
3	**C**	0					
4	**B**	0					

ABCB
BDCAB

if (*Xi == Yi*)
c[i,j] = c[i-1,j-1]+1
else *c[i,j] = max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (8)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	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					
4	**B**	0					

ABCB
BDCAB

if (*Xi == Yi*)
c[i,j] = c[i-1,j-1]+1
else *c[i,j] = max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (9)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	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			
4	**B**	0					

ABCB
BDCAB

if (*Xi == Yi*)
c[i,j] = c[i-1,j-1]+1
else *c[i,j] = max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (10)

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	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		
4	**B**	0					

ABCB
BDCAB

if (*Xi == Yi*)
c[i,j] = c[i-1,j-1]+1
else *c[i,j] = max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (11)

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					

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (12)

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				

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (13)

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	

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Najdlhšia spoločná podpostupnosť – príklad (13)

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

c[4,5] obsahuje dĺžku najdlhšej spoločnej podpostupnosti

ABCB
BDCAB

if (*Xi* == *Yi*)
c[i,j] = *c[i-1,j-1]+1*
else *c[i,j]* = *max(c[i-1,j], c[i,j-1])*

Analýza algoritmu pre nájdenie najdlhšej spoločnej podpostupnosti

- LCS algoritmus vypočíta hodnoty každého vstupu poľa *c[m,n]*. *Aký je teda výpočtový čas?*
- *O(m*n)*
  - Každá hodnota *c[i,j]* je spočítaná v konštantnom čase, a v poli máme *m*n* prvkov
- Zatiaľ sme našli len dĺžku najdlhšej spoločnej podpostupnosti.
- Ďalej je potrebné nájsť najdlhšiu spoločnú podpostupnosť.
- Musíme modifikovať algoritmus aby nám dával výstup najdlhšej spoločnej podpostupnosti postupností X a Y
  - V poli *c[i,j]* máme všetko zaznamenané
  - Každá hodnota *c[i,j]* závisí na *c[i-1,j]* alebo *c[I,j-1]*
  - Pre každú hodnotu *c[i,j]* vieme určiť ako sme ju dosiahli

Hľadanie najdlhšej spoločnej podpostupnosti

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & ak\ x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & inak \end{cases}$$

2	2
2	3

V tomto prípade
c[i,j] = c[i-1,j-1] + 1 = 2 + 1 = 3

1	1
2	2

V tomto prípade
c[i,j] = max(c[i,j-1], c[i-1,j]) = max(2, 1) = 2

- Môžeme začať z c[m,n] a ísť späť
- Ak *c[i,j] = c[i-1,j-1]* + 1, zapamätáme si *x[i]*
  - *x[i]* je časť z najdlhšej spoločnej podpostupnosti
- Ak *i* = 0 alebo *j* = 0 (dosiahneme začiatok), výstupom sú písmená odpamätané v X, usporiadané v opačnom poradí

Hľadanie najdlhšej spoločnej podpostupnosti

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	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**

Hľadanie najdlhšej spoločnej podpostupnosti

	j	0	1	2	3	4	5
i		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i	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**

Najdlhšia spoločná postupnosť (odzadu): **B C B**

Najdlhšia spoločná postupnosť: **B C B**