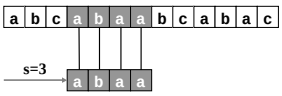
Porovnávanie reťazcov

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

- 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

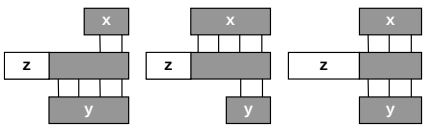


Základné pojmy

- Predpona (prefix): reťazec w je predponou reťazca x, ak x = wy, kde y je akýkoľvek reťazec z použitej abecedy Σ
  - **Napr: pre(*ab, abcca*)**
- Prípona (suffix): reťazec w je príponou reťazca x, ak x = yw, kde y je akýkoľvek reťazec z použitej abecedy 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*| ≤|*y*|, *tak suf(x,y)*
  - *ak* |*x*| ≥|*y*|, *tak suf(y,x)*
  - *ak* |*x*| = |*y*|, *tak x = y*



Najznámejšie algoritmy

Algoritmus	Predspracovanie	Vyhľadávanie
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 popisy, vizualiácie a zdrojové kódy môžete nájsť napr. 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 hľadanie výskytu reťazca v reťazci

- Samotný algoritmus pozostáva z porovnávania znakov na všetkých pozíciach medzi 0 a n-m. Pri každom kroku sa posúva iba o 1 pozíciu 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);
} }
```

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

- Hešovacia funkcia by mala mať tieto vlastnosti:
  - jednoducho vypočitateľná
  - 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. Keď použijeme konštantu d - maximálny počet možných znakov, tak platí:

- $x = t[i]d^m + t[i+1]d^{m-1}+...+t[i+m]$

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

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

Pokiaľ podrobne preskúmame x a x', tak zistíme, že:

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

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:
- $pathash = (d^{m-1} \text{ ord}(pat_0) + d^{m-2} \text{ ord}(pat_1) + ... + d \text{ ord}(pat_{m-2}) + \text{ord}(pat_{m-1})) \bmod q$
- $texthash_i = (d^{m-1} \text{ ord}(text_i) + d^{m-2} \text{ ord}(text_{i+1}) + ... + d \text{ ord}(text_{i+m-2}) + \text{ord}(text_{i+m-1})) \bmod q$
- $texthash_{i+1} = (d^{m-1} \text{ ord}(text_{i+1}) + d^{m-2} \text{ ord}(text_{i+2}) + ... + d \text{ ord}(text_{i+m-1}) + \text{ord}(text_{i+m})) \bmod q$

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

- Hešovaciu funkciu ovplyvňujú parametre d a q.

Rabinov-Karpov algoritmus

- Hešovacia funkcia:

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

- Voľba parametrov d a q:
  - d - maximálny počet možných znakov
  - q - prvočíslo

Rabinov-Karpov algoritmus

- Zhrnutie
  - Vypočítame základný heš reťazca aj vzoru a postupne budeme pokračovať vo vzore pripočítame hodnoty ďalšieho znaku a odpočítame špecifickú hodnotu prvého znaku v pôvodnom kuse reťazca. Teda dve konštantné funkcie, ktoré zaberú minimálne množstvo času. Stále však pretrváva problém kolízií, preto dva texty s rovnakým hešom porovnávame po znakoch

Rabinov-Karpov algoritmus

```
RABIN-KARP POROVNANIE(T, P, d, q)
n ← length(T)
m ← length(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  //parovanie
        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:
  - predspracovanie
    má časovú zložitosť $O(m)$
  - vyhľadávanie
    má časovú zložitosť $O(mn)$
  - Očakávaná doba výpočtu 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;
    }
}
```

Vyhľadávanie pomocou automatu

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

Vyhľadávanie pomocou automatu

Def.: Konečny automat (finite automaton) je usporiadana 5-tica $(Q, q_0, A, \Sigma, \delta)$, kde

- Q je konečna množina stavov
- q_0 začiatočny stav
- A množina koncovych stavov (akceptujuce), A je podmnožinou Q
- Σ je použita abeceda
- δ je tzv. prechodova funkcia z $Q \times \Sigma$ do Q.

Rozširenie prechodovej funkcie δ –

$\delta^* : Q \times \Sigma^* \rightarrow Q$ je definovane induktivne:

$\delta^*(q, \varrho) = q$

$\delta^*(q,wa) = \delta(\delta^*(q, w), a)$

Funkcia koncového stavu φ - vracia stav automatu po spracovani nejakeho slova

$\varphi(\varepsilon) = q_0$

$\varphi(wa) = \delta(\varphi(w), a)$

Algoritmus vyhľadávania pomocou automatu

```
POROVNANIE KONEČNÝM AUTOMATOM(T, δ ,m)
n ← length(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
```

O(*n*), ale treba zostrojiť automat, t.j. v podstate prechodovú funkciu

Ako zostrojiť automat

príponová funkcia pre P, |P| = m je

$\sigma : \Sigma^* \rightarrow \{0, 1, \ldots, m\}$

definovaná ako
$\sigma(x) = \max\{k : P_k \sqsupset x\}$,
kde u $\sqsupset$ v znamena, že u je príponou v a $P_k = P[1..k]$.

Po slovensky:
σ(x) je dĺžka najdlhšej predpony vzoru P, ktorá je súčasne príponou x

Ako zostrojiť automat

Definujme automat hľadajúci výskyt vzoru P dĺžky m vo vstupnom reťazci takto:

Množina stavov:

$Q = \{0, 1, \ldots, m\}$,

Začiatočný stav:

$q_0 = 0$,

Množina koncových stavov:

$A = \{m\}$,

Prechodová funkcia:

$\delta(q, a) = \sigma(P_q a)$.

Prechodová funkcia

definujme prechodovú funkciu takto:

$\delta(q, a) = \sigma(P_q a)$

prečo?

Invariant činnosti automatu:

$\varphi(T_i) = \sigma(T_i)$

t.j. po prečítaní prvých *i* znakov je automat v stave $q = \varphi(T_i)$, pričom $q = \sigma(T_i)$ je dĺžka najdlhšej prípony reťazca T_i ktorá je súčasne aj predponou vzoru *P*.

Ako zostrojiť automat

Vždy, keď sa počas simulacie vstupneho slova T na automate dostaneme do stavu m, našiel sa podvyraz P a jeho posun je rovny o m menej ako je aktualna pozicia v reťazci.

Platia nasledujuce vety:

- V (suffix-function inequality): Pre každy reťazec x a znak a plati: $\sigma(xa) \leq \sigma(x) + 1$.
- V (suffix-function recursion lemma): Pre každy reťazec x a znak a, ak $q = \sigma(x)$, tak $\sigma(xa) = \sigma(Pqa)$

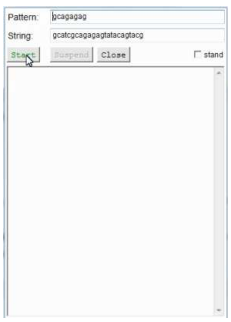
Vyhľadávanie pomocou automatu

- VYPOČET PRECHODOVEJ FUNKCIE (P,Σ)

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

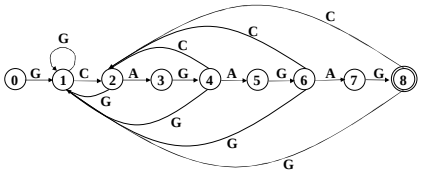
- Zložitosť algoritmu:
  - predspracovanie: $O(m^3|\Sigma|)$, ale dá sa urobiť šikovnejšie a zlepšiť na $O(m|\Sigma|)$
  - vyhľadávanie: $O(n)$

Vyhľadávanie pomocou automatu



Vyhľadávanie pomocou automatu

- Konečný automat z animovaného príkladu:

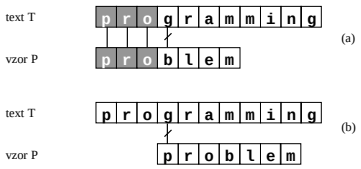


Knuthov-Morrisov-Prattov algoritmus

- 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.

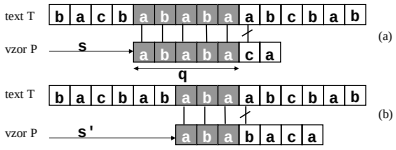
Knuthov-Morrisov-Prattov algoritmus

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:

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

$\pi(q) = \max\{k : k < q, P_k \sqsupset P_q\}$.

Po slovensky:

π(q) je dĺžka najdlhšej predpony vzoru P, ktorá je súčasne pravou príponou P_q

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

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 ďalší zľava doprava
        do while q > 0 and P[q + 1] ≠ T[i]
                do q ← π[q]          //nezhoduje sa ďalší znak
           if P[q + 1] = T[i]
                then q ← q + 1      //zhoduje sa ďalší znak
           if q = m                 //zhoduje sa celý vzor P?
                then print "Vzor sa v retazci vyskytuje s posunom " i-m
                     q ← π[q]                //hľadať ďalší možný výskyt
```

Knuthov-Morrisov-Prattov algoritmus

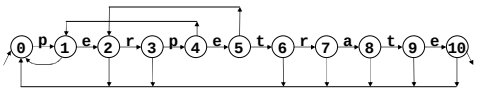
```
PREDPONOVA FUNKCIA (P)
m ← length(P)
π[1] ← 0
k ← 0
for q ← 2 to m
        do while k > 0 and P[k + 1] ≠ P[q]
                do k ← π[k]
           if P[k + 1] = P[q]
                then k ← k + 1
           π[q] ← k
return π
```

Knuthov-Morrisov-Prattov algoritmus

- 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 podla písmena p_1, prechod mezi prvným a druhým stavom podla 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.

Knuthov-Morrisov-Prattov algoritmus

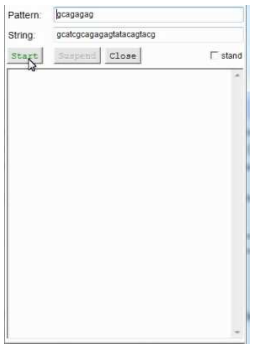
- Príklad konečného automatu pre vzor perpetrate



Knuthov-Morrisov-Prattov algoritmus

- Zložitosť algoritmu:
  - Predspracovanie: **O**(m)
  - Vyhľadávanie: **O**(n)
  - Celkovo KMP: **O**(m+n)

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;
    }
}
```

```
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];
        }
    }
}
```

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
  - Podpostupnot': 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 prefixy 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 l'
  - ***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 == Yj)
                        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 — X = ABCB; m = |X| = 4
BDCAB — Y = BDCAB; n = |X| = 5
alokácia 2D poľa c[0..4, 0..5]

Najdlhšia spoločná podpostupnosť – príklad (1)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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)

i		j	0	1	2	3	4	5
			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
			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					

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
			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				

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
		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	

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
		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

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])*

c[4,5] obsahuje dĺžku najdlhšej spoločnej podpostupnosti

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 spoločenj podpostupnosti.
- Ďalej je potrebné nájsť najdlhšiu spoločnú podpostupnosť.
- Musíme modifikovať algoritmus aby nám dával výstup najdlhšej spoločnej podpostupnosti postupnosti 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

i		j	0	1	2	3	4	5
		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

i		j	0	1	2	3	4	5
		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**