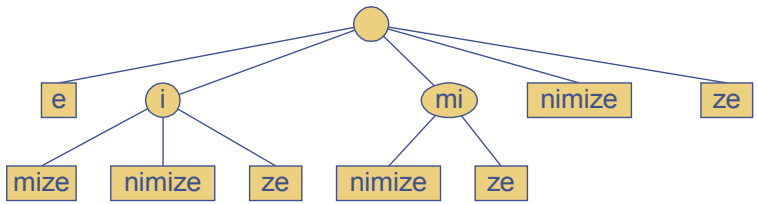
Algoritmy zpracování textů II

- datová struktura Trie
- nejdelší společná sekvence (LCS)
- nejkratší společná nad-sequence (SCS)
- vzdálenost mezi řetězci

Předzpracování řetězců

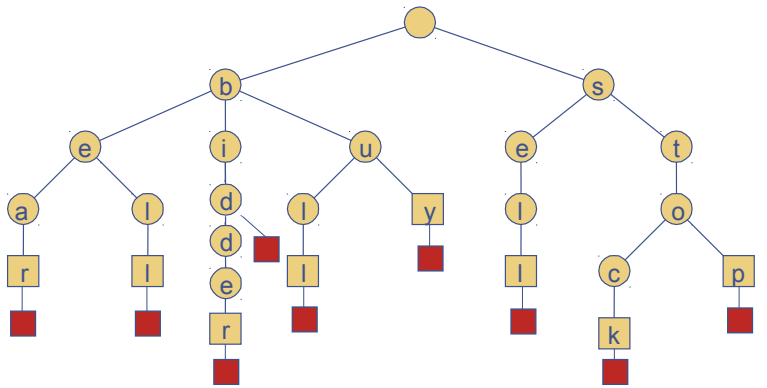
- U algoritmů vyhledávání řetězců se předzpracovává hledaný vzor, aby se urychlilo jeho vyhledávání
- Pro rozsáhlé neměnné texty ve kterých se často vyhledává je výhodnější předzpracovat celý text, než se zabývat předzpracováním vzoru (BM, KMP algoritmus)
- Trie je kompaktní datová struktura vhodná pro reprezentaci množiny retězců, kterými mohou být např. slova v textu
  - Trie umožňuje vyhledávat řetězce v čase úměrném velikosti hledaného vzoru

Datová struktura Trie



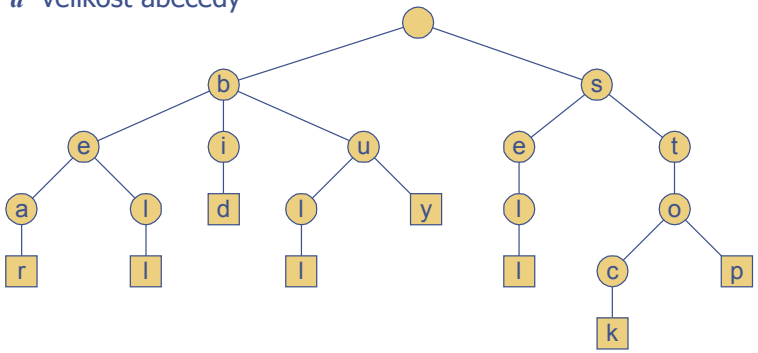
Standardní Trie

- 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 }



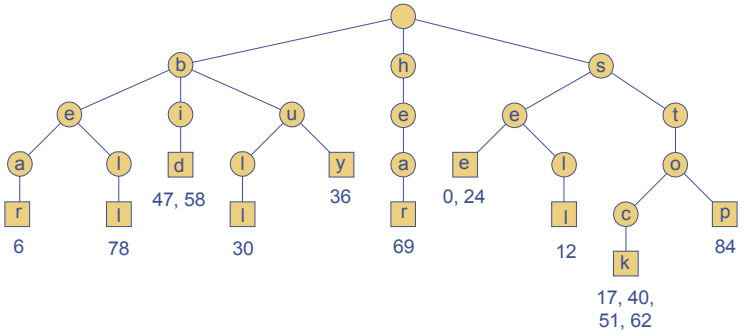
Analýza Standardní Trie

- Standardní trie vyžaduje $O(n)$ paměťového prostoru a umožňuje vyhledávání, vkládání a rušení v čase $O(dm)$, kde:
  - n celková velikost řetězců v S
  - m velikost zpracovávaného řetězce
  - d velikost abecedy



Vyhledávání slov pomocí Trie

- Slova z textu jsou uložena do trie
- V každém listu je zároveň uložena informace o pozici výskytu slova v textu

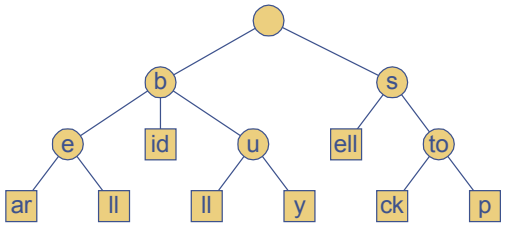


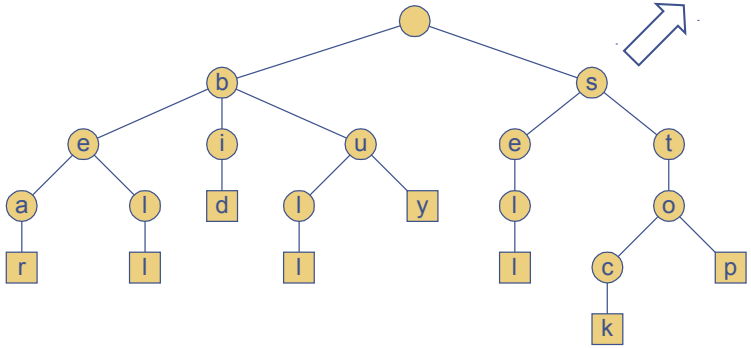
Typické použití datové struktury Trie

- Standardní trie umožňuje provádět následující operace nad předzpracovaným textem v čase O(m), kde m velikost slova X:
  - **Vyhledávání slov (Word Matching):** nalezení prvního výskytu slova X v textu.
  - **Vyhledávání prefixu (Prefix Matching):** Nalezení prvního výskytu nejdelšího prefixu slova X v textu.

Komprimovaná Trie

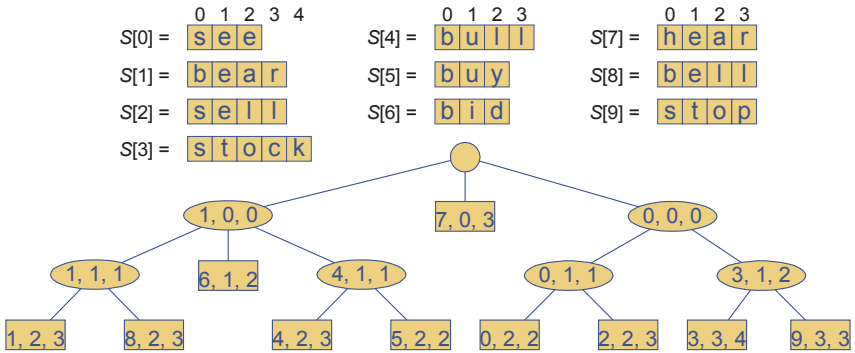
- Komprimovaná trie má vnitřní uzly stupně nejméně 2
- Získáva se ze standardní trie, kompresí řetězců tzv. „redundantních" uzlů tj. uzlů, které mají pouze jednoho následníka





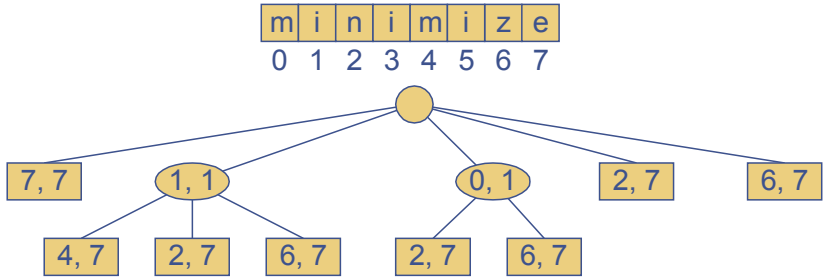
Kompaktní reprezentace komprimované Trie

- Kompaktní reprezentace komprimované trie pro pole řetězců:
  - Uchovává v uzlech trojici indexů (i,j,k) místo celých řetězců.
    i – index v poli (tabulce), kde je řetězec uložen
    j – počáteční index podřetězce uloženého v uzlu
    k – koncový index podřetězce uloženého v uzlu
  - Využívá $O(s)$ paměťového prostoru, kde s je počet řetězců v poli
  - Slouží jako pomocná indexová struktura



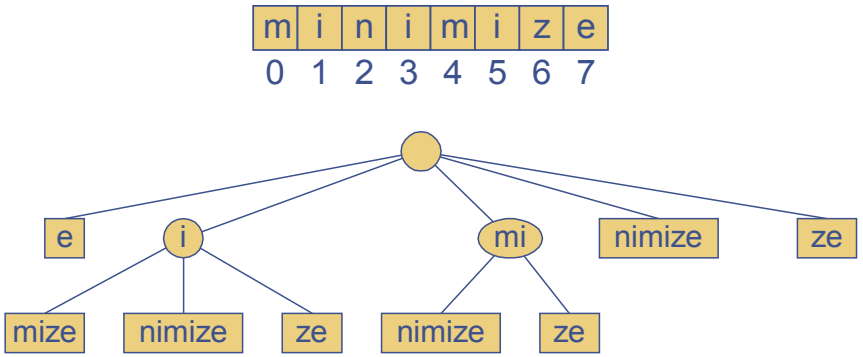
Analýza Suffixové Trie

- Kompaktní reprezentace suffixové trie řetězce X velikosti n vzniklého z abecedy mohutnosti d
  - Využívá $O(n)$ paměťového prostoru.
  - Umožňuje libovolné pokládání dotazů na přítomnost řetězce v textu X v čase $O(dm)$, kde m je velikost vzorového řetězce
  - Lze ji vytvořit v čase $O(n)$.



Suffixová Trie

- Suffixová trie řetězce X je komprimovaná trie všech suffixů X



Algoritmus vyhledávání řetězců suffixovou Trie

Algorithm suffixTrieMatch(T,P):
Input: Compact suffix trie T for a text X and pattern P
Output: Starting index of a substring of X matching P or an indication that P is not a substring of X

```
p ← P.length() { length of suffix of the pattern to be matched }
j ← 0 { start of suffix of the pattern to be matched }
v ← T.root()
repeat
  f ← true { flag indicating that no child was successfully processed }
  for each child w of v do
    i ← start(v)
    if P[j] = T[i] then
      { process child w }
      x ← end(w) − i + 1
      if p ≤ x then
        { suffix is shorter than or of the same length of the node label }
        if P[j..j + p − 1] = X[i..i + p − 1] then
          return i − j { match }
        else
          return "P is not a substring of X"
      else
        { suffix is longer than the node label }
        if P[j..j + x − 1] = X[i..i + x − 1] then
          p ← p − x { update suffix length }
          j ← j + x { update suffix start index }
          v ← w
          f ← false
          break out of the for loop
until f or T.isExternal(v)
return "P is not a substring of X"
```

Trie a Webové vyhledávání

- kolekce všech vyhledávaných slov (tzv. search engine index) je uchováván v komprimované trie.
- Každý uzel trie odpovídá hledanému slovu a je zároveň spojen se seznamem stránek (URLs) obsahující toto slovo - tzv. seznam výskytů (**occurrence list**).
- Trie se uchovává v interní paměti.
- Seznam výskytů se uchovává v externí paměti a jsou uspořádány podle důležitosti

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)

Platí :

- Nechť $X=<x_1,x_2,\ldots,x_m>$ a $Y=<y_1,y_2,\ldots y_n>$ jsou řetězce a $Z=<z_1,z_2,\ldots,z_k>$ je libovolná LCS X a Y
- Jestliže $x_m = y_n$ pak $z_k = x_m = y_n$ a Z_{k-1} je LCS X_{m-1} a Y_{n-1}
- Jestliže $x_m \neq y_n$ a $z_k \neq x_m$, pak z toho vyplývá, že Z je LCS X_{m-1} a Y
- Jestliže $x_m \neq y_n$ a $z_k \neq y_n$, pak Z je LCS X a Y_{n-1}

Postup:

- Nejprve nalezneme délku LCS a podél „cesty", kterou budeme procházet, si budeme nechávat značky, které nám pomohou nalézt výslednou nejdelší společnou sekvenci
- Nechť X_i, Y_j jsou prefixy X a Y délky i a j .
- Nechť $c[i,j]$ je délka LCS X_i and Y_j
- Pak délka kompletní LCS X a Y bude $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{ve zbývajících situacích} \end{cases}$$

Rekurentní řešení

- Začneme s $i = j = 0$ (prázdné podřetězce x a y)
- Protože X_0 and Y_0 jsou prázdné řetězce je jejich LCS vždy prázdná (tj. $c[0,0] = 0$)
- LCS prázdného řetězce a libovolného jiného řetězce je také prázdná a tak pro každé i a j :

$$c[0, j] = c[i,0] = 0$$

- když určujeme hodnotu c[i,j], tak uvažujeme dva případy:
  - **První případ:** $x[i]=y[j]$: další symbol v řetězci X and Y se shoduje a délka LCS X_i a Y_j je rovna délce LCS kratších řetězců X_{i-1} a Y_{i-1} , zvětšená o 1.
  - **Druhý případ:** $x[i] != y[j]$ tj. symboly se neshodují a tudíž se délka LCS(X_i,Y_j) nezvětší a zůstává shodná jako předtím (tj. maximum z LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j))

LCS Algoritmus

```
LCS-Length(X, Y)
m = length(X),  n  = length(Y)
for i = 1 to m
     do c[i, 0] = 0
for j = 0 to n
      do c[0, j] = 0
for i = 1 to m
        do for j = 1 to n
              do if ( xi = = yj )
                      then c[i, j] = c[i - 1, j - 1] + 1
                           b[i, j] =" ←↑ "
              else if c[i - 1, j]>=c[i,  j - 1]
                              then  c[i, j] = c[i - 1, j]
                                    b[i, j] =" ↑ "
                              else c[i, j] = c[i, j - 1]
                                    b[i, j] =" ←"

return c and b
```

Příklad:

- Hledáme nejdelší společný podřetězec (LCS) řetězců
  - X = ABCB
  - Y = BDCAB

LCS(X, Y) = BCB

- X = A **B** **C** **B**
- Y = **B** D **C** A **B**

LCS příklad

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

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[6,5]

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

for i = 1 to m c[i,0] = 0

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					
2	B	0					
3	C	0					
4	B	0					

for j = 0 to n c[0,j] = 0

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 ↑				
2	B	0					
3	C	0					
4	B	0					

case i=1 and j=1
A != B
but, c[0,1]>=c[1,0]
so c[1,1] = c[0,1], and b[1,1] = ↑

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 ↑			
2	B	0					
3	C	0					
4	B	0					

case i=1 and j=2
A != D
but, c[0,2]>=c[1,1]
so c[1,2] = c[0,2], and b[1,2] = ↑

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 ↑		
2	B	0					
3	C	0					
4	B	0					

case i=1 and j=3
A != C
but, c[0,3]>=c[1,2]
so c[1,3] = c[0,3], and b[1,3] = ↑

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					
3	C	0					
4	B	0					

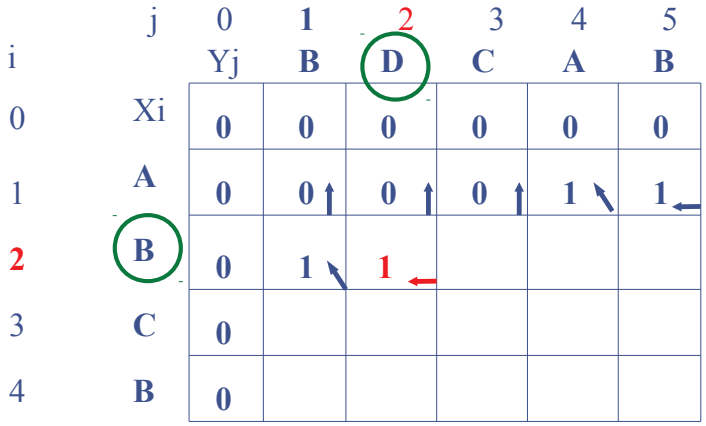
case i=1 and j=5
A != B
this time c[0,5]<c[1,4]
so c[1,5] = c[1, 4], and b[1,5] = ←

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					
3	C	0					
4	B	0					

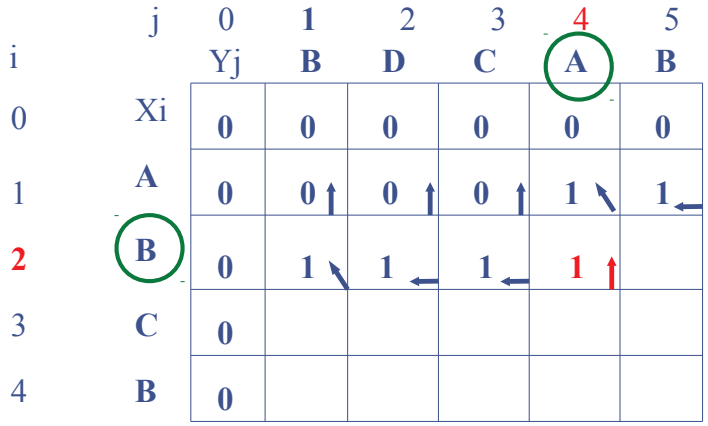
case i=1 and j=4
A = A
so c[1,4] = c[0,2]+1, and b[1,4] = ↖

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 ↖				
3	C	0					
4	B	0					

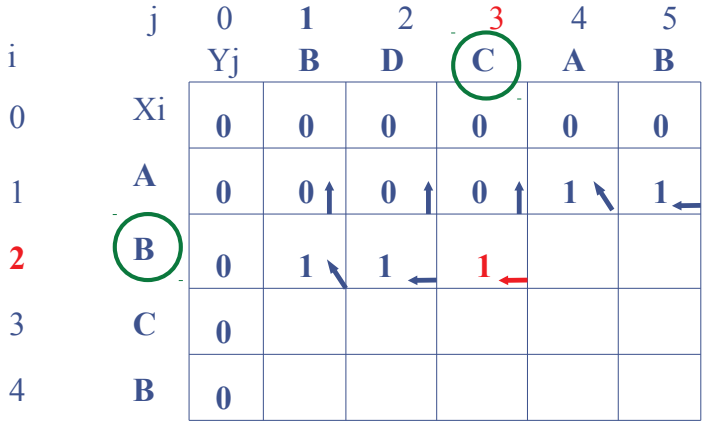
case i=2 and j=1
B = B
so c[2, 1] = c[1, 0]+1, and b[2, 1] = ↖



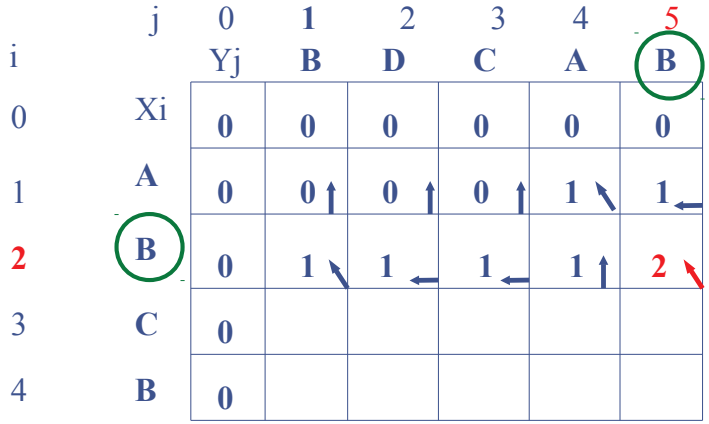
case i=2 and j=2
B != D
and c[1, 2] < c[2, 1]
so c[2, 2] = c[2, 1] and b[2, 2] = ←



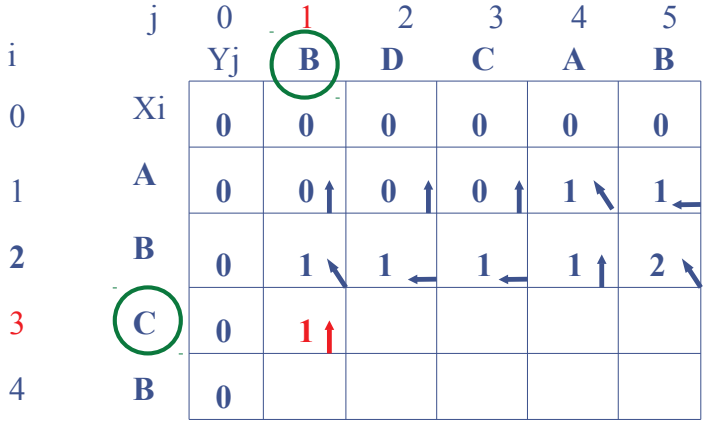
case i=2 and j=4
B != A
and c[1, 4] = c[2, 3]
so c[2, 4] = c[1, 4] and b[2, 2] = ↑



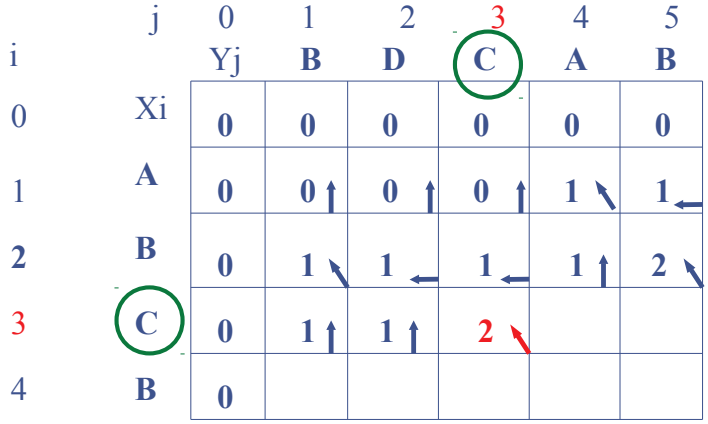
case i=2 and j=3
B != D
and c[1, 3] < c[2, 2]
so c[2, 3] = c[2, 2] and b[2, 3] = ←



case i=2 and j=5
B = B
so c[2, 5] = c[1, 4]+1 and b[2, 5] = ↖

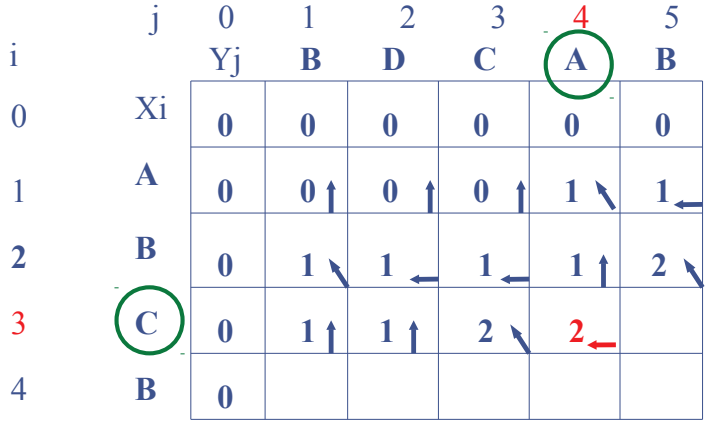


case i=3 and j=1
C != B
and c[2, 1] > c[3,0]
so c[3, 1] = c[2, 1] and b[3, 1] =



case i=3 and j= 3
C = C
so c[3, 3] = c[2, 2]+1 and b[3, 3] =

case i=3 and j= 2
C != D
and c[2, 2] = c[3, 1]
so c[3, 2] = c[2, 2] and b[3, 2] =



case i=3 and j= 4
C != A
c[2, 4] < c[3, 3]
so c[3, 4] = c[3, 3] and b[3, 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					

case i=3 and j= 5
C != B
c[2, 5] = c[3, 4]
so c[3, 5] = c[2, 5] and b[3, 5] = ↑

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 ↑			

case i=4 and j=2
B != D
c[3, 2] = c[4, 1]
so c[4, 2] = c[3, 2] and b[4, 2] = ↑

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 ↖				

case i=4 and j=1
B = B
so c[4, 1] = c[3, 0]+1 and b[4, 1] = ↖

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 ↑		

case i=4 and j= 3
B != C
c[3, 3] > c[4, 2]
so c[4, 3] = c[3, 3] and b[4, 3] = ↑

Nalezení LCS

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	

case i=4 and j=4
B != A
c[3, 4] = c[4, 3]
so c[4, 4] = c[3, 4] and b[3, 5] = ↑

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

case i=4 and j=5
B= B
so c[4, 5] = c[3, 4]+1 and b[4, 5] = ↖

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

LCS (obrácené pořadí): **B C B**

LCS (správné pořadí): **B C B**

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.

Rekurentní řešení

- Začneme s $i = j = 0$ (prázdné podřetězce x a y)
- Protože X_0 and Y_0 jsou prázdné řetězce je jejich SCS vždy prázdná (tj. $c[0,0] = 0$)
- SCS prázdného řetězce a libovolného jiného řetězce je rovná danému řetězci a tak pro každé i a j je délka :

  $c[0, j] = j$

  $c[i,0] = i$
- když určujeme hodnotu c[i,j], tak uvažujeme dva případy:
  - **První případ:** $x[i]=y[j]$: další symbol v řetězci X and Y se shoduje a délka SCS X_i a Y_j je rovna délce SCS kratších řetězců X_{i-1} a Y_{i-1} , zvětšená o 1.
  - **Druhý případ:** $x[i] != y[j]$ tj. symboly se neshodují a délka SCS(X_i,Y_j) je daná minimální hodnotou dvojice SCS(X_i, Y_{j-1}) and SCS(X_{i-1},Y_j)

Postup:

- Nejprve nalezneme délku SCS a podél „cesty", kterou budeme procházet, si budeme nechávat značky, které nám pomohou nalézt výslednou nejkratší společnou super-sekvenci
- Nechť X_i, Y_j jsou prefixy X a Y délky i a j .
- Nechť $c[i,j]$ je délka SCS X_i and Y_j
- Pak délka kompletní SCS X a Y bude v ***c[m,n]***

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \min(c[i,j-1]+1, c[i-1,j]+1) & \text{ve zbývajících situacích} \end{cases}$$

SCS Algoritmus

```
SCS-Length(X, Y)
m = length(X),  n  = length(Y)
for i = 1 to m
     do c[i, 0] = i
for j = 0 to n
      do c[0, j] = j
for i = 1 to m
        do for j = 1 to n
                do if ( xi = = yj )
                        then c[i, j] = c[i - 1, j - 1] + 1
                             b[i, j] =" ←↑ "
                else if c[i - 1, j]<=c[i,  j - 1]
                                then  c[i, j] = c[i - 1, j]+1
                                      b[i, j] =" ↑ "
                               else c[i, j] = c[i, j - 1]+1
                                      b[i, j] =" ←"

return c and b
```

Příklad:

- Hledáme nejkratší společný nadřetězec (SCS) řetězců
  - X = ABCB
  - Y = BDCAB

SCS(X, Y) = ABDCAB
- X = **A B C B**
- Y = **B D C A B**

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

for i = 1 to m c[i,0] = i

LCS příklad

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

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[6,5]

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	**B**	**2**					
3	**C**	**3**					
4	**B**	**4**					

for j = 0 to n c[0,j] = j

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				
2	B	2					
3	C	3					
4	B	4					

case i=1 and j=1
A != B
but, c[0,1]+1<=c[1,0]+1
so c[1,1] = c[0,1]+1, and b[1,1] =

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		
2	B	2					
3	C	3					
4	B	4					

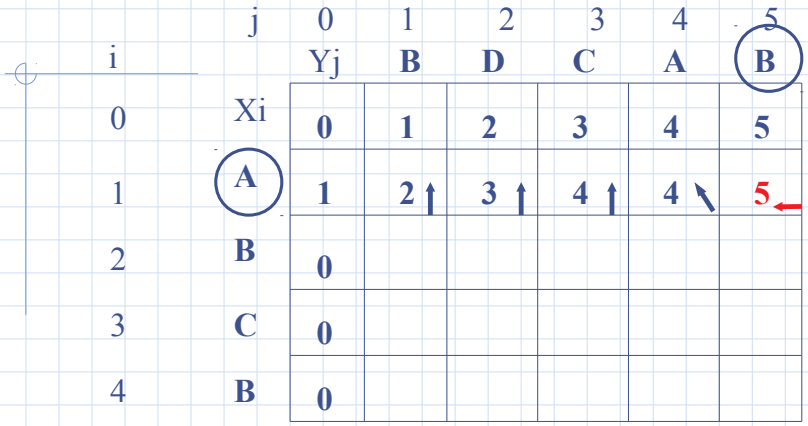
case i=1 and j=3
A != C
but, c[0,3]+1<=c[1,2]+1
so c[1,3] = c[0,3]+1, and b[1,3] =

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			
2	B	2					
3	C	3					
4	B	4					

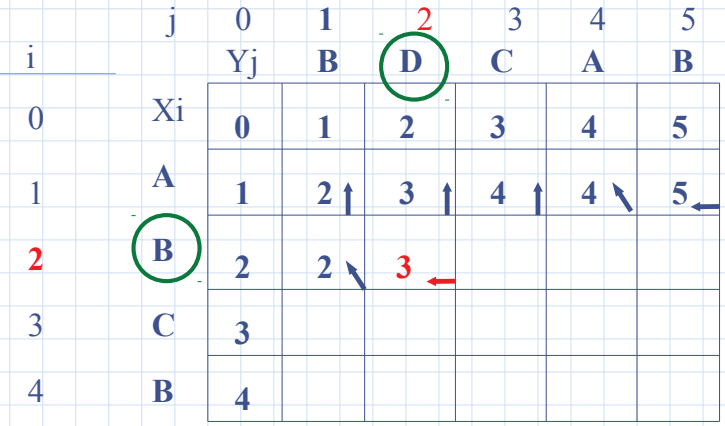
case i=1 and j=2
A != D
but, c[0,2]+1<=c[1,1]+1
so c[1,2] = c[0,2]+1, and b[1,2] =

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	
2	B	0					
3	C	0					
4	B	0					

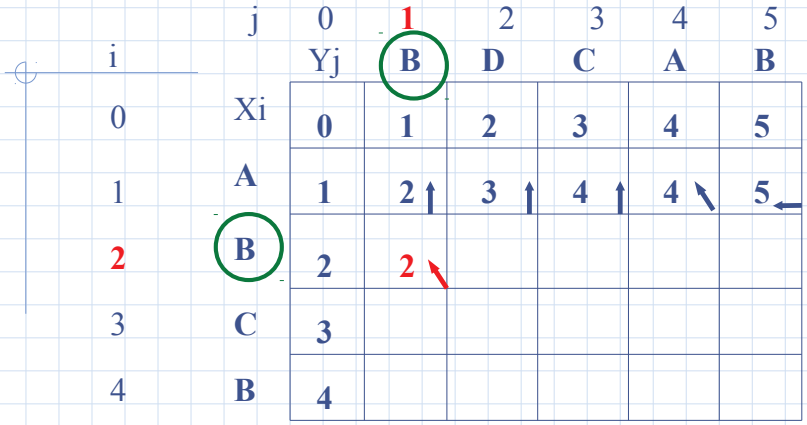
case i=1 and j=4
A = A
so c[1,4] = c[0,2]+1, and b[1,4] =



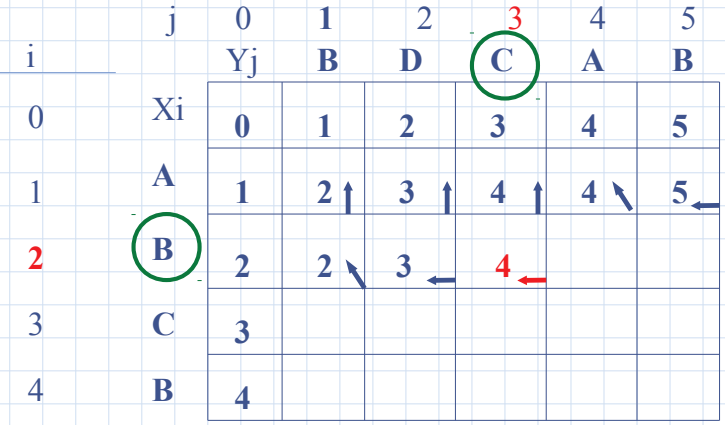
case i=1 and j=5
A != B
this time c[0,5]+1<c[1,4]+1
so c[1,5] = c[1, 4]+1, and b[1,5] = ←



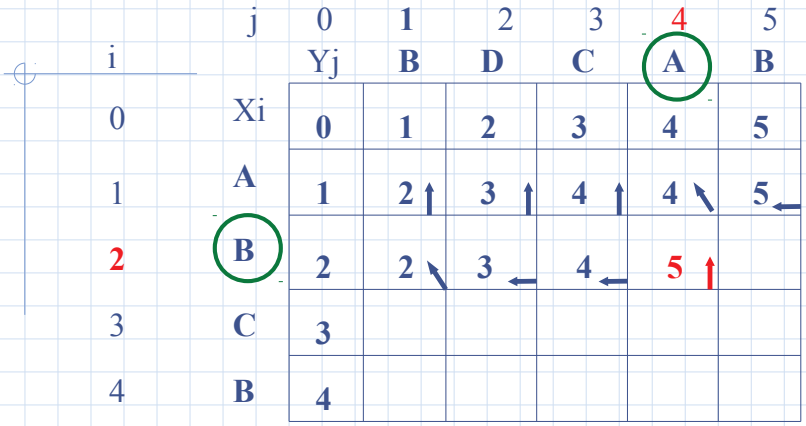
case i=2 and j=2
B != D
and c[1, 2]+1 > c[2, 1]+1
so c[2, 2] = c[2, 1]+1 and b[2, 2] = ←



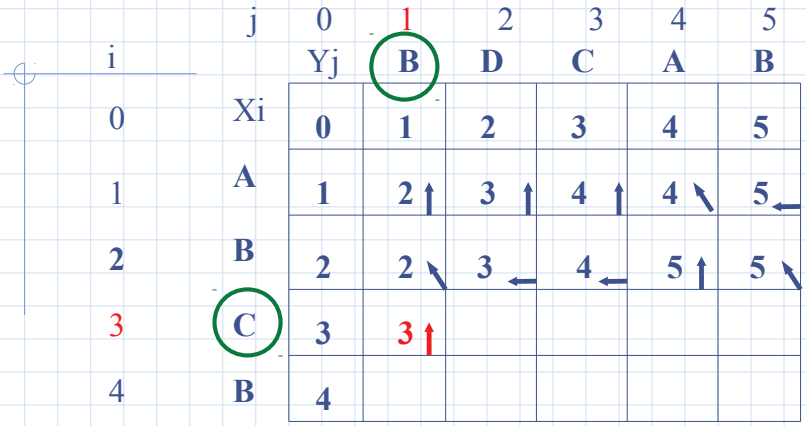
case i=2 and j=1
B = B
so c[2, 1] = c[1, 0]+1, and b[2, 1] = ↖



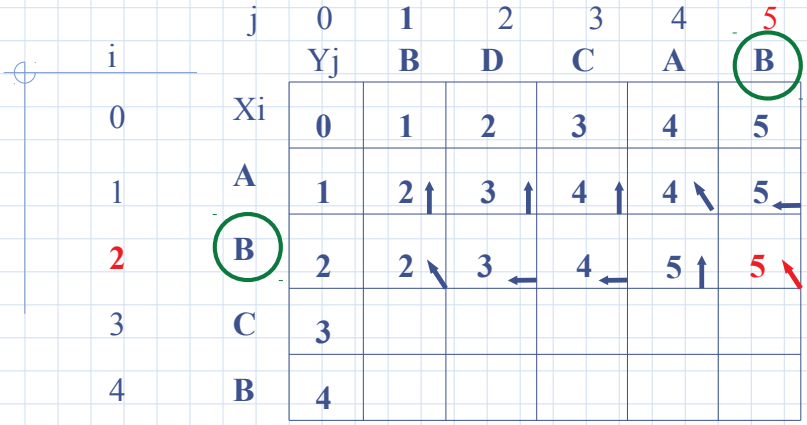
case i=2 and j=3
B != D
and c[1, 3]+1 > c[2, 2]+1
so c[2, 3] = c[2, 2]+1 and b[2, 3] = ←



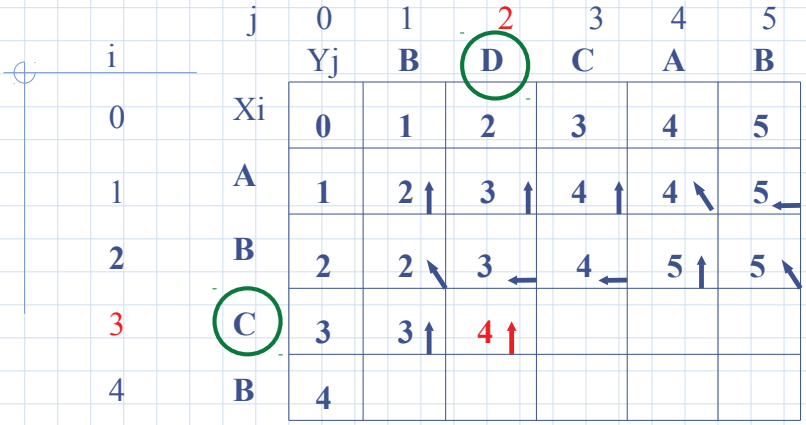
case i=2 and j=4
B != A
and c[1, 4]+1 = c[2, 3]+1
so c[2, 4] = c[1, 4]+1 and b[2, 2] =



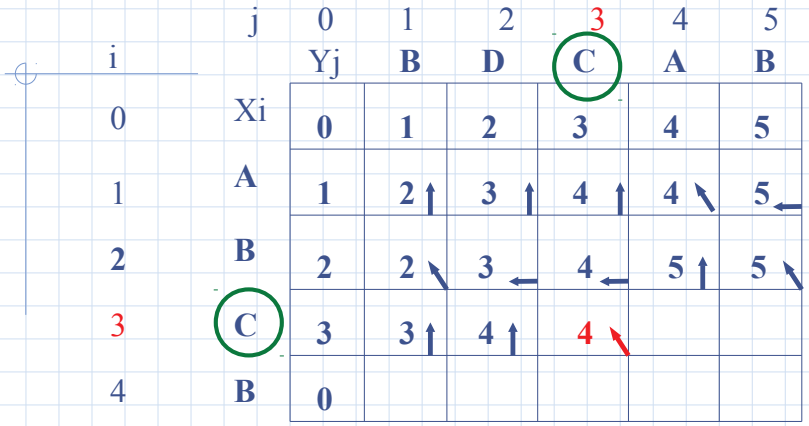
case i=3 and j=1
C != B
and c[2, 1] +1 < c[3,0]+1
so c[3, 1] = c[2, 1]+1 and b[3, 1] =



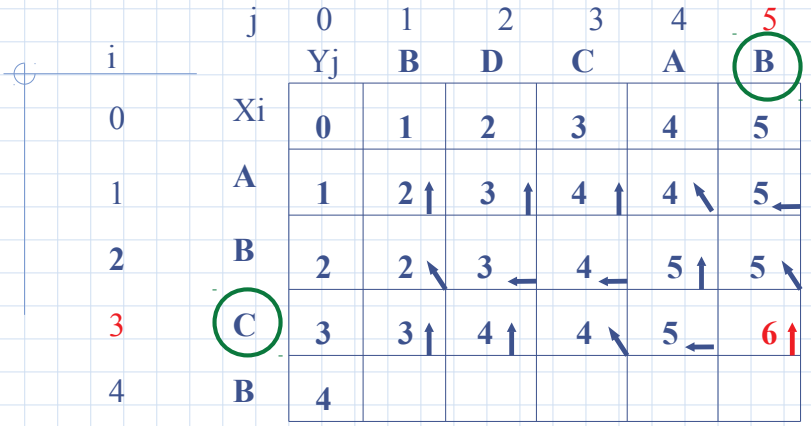
case i=2 and j=5
B = B
so c[2, 5] = c[1, 4]+1 and b[2, 5] =



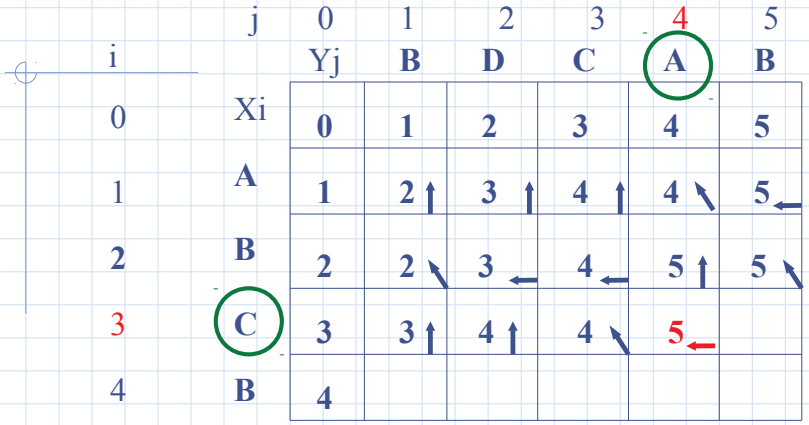
case i=3 and j= 2
C != D
and c[2, 2]+1 = c[3, 1]+1
so c[3, 2] = c[2, 2]+1 a b[3, 2] =



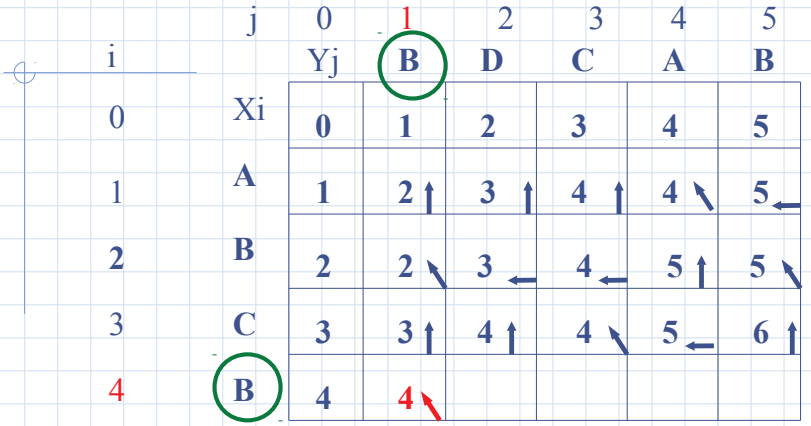
case i=3 and j= 3
C = C
so c[3, 3] = c[2, 2]+1 and b[3, 3] =



case i=3 and j= 5
C != B
c[2, 5]+1= c[3, 4]+1
tak c[3, 5] = c[2, 5]+1 a b[3, 5] =



case i=3 and j= 4
C != A
c[2, 4] +1 > c[3, 3]+1
so c[3, 4] = c[3, 3]+1 and b[3, 3] =



case i=4 and j=1
B = B
so c[4, 1] = c[3, 0]+1 and b[4, 1] =

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 ↑			

case i=4 and j=2
B != D
c[3, 2]+1 = c[4, 1]+1
so c[4, 2] = c[3, 2]+1 and b[4, 2] = ↑

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 ↑	

case i=4 and j=4
B != A
c[3, 4] +1= c[4, 3]+1
so c[4, 4] = c[3, 4] +1 and b[3, 5] = ↑

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 ↑		

case i=4 and j= 3
B != C
c[3, 3]+1 > c[4, 2]+1
so c[4, 3] = c[3, 3]+1 and b[4, 3] = ↑

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 ↖

case i=4 and j=5
B= B
so c[4, 5] = c[3, 4]+1 and b[4, 5] = ↖

Nalezení 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

SCS (obrácené pořadí): **B A C D B A**

SCS (správné pořadí): **A B D C A B**

Porovnávání řetězců (edit distance)

- přesné porovnávání dvou řetězců (vzájemná shoda) není použitelné v některých oblastech, které využívají symbolický popis (strukturní metody rozpoznávání)
- k testování podobnosti dvou řetězců

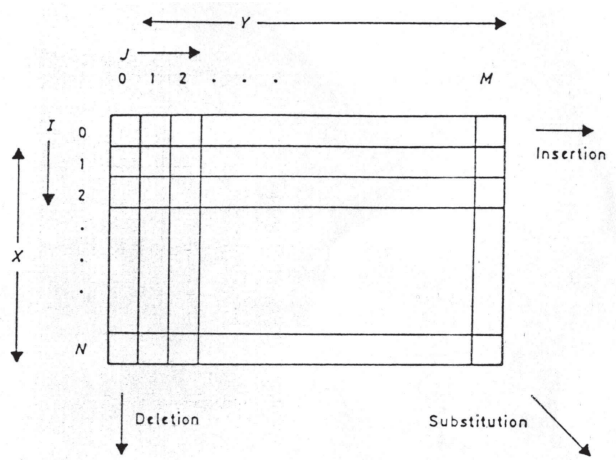
  $x=x_1x_2...x_n \in T^*$ a $y=y_1y_2...y_n \in T^*$

  (T je abeceda symbolů) je nutné definovat vhodnou metriku
- Hammingova metrika $d_H(x,y)$ – pouze pro řetězce stejné délky. Je definovaná jako počet odlišných symbolů ***x*** a ***y*** v odpovídajících si pozicích (např. řetězce abcab , bbdab mají $d_H=2$)
- Levensteinova metrika $d(x,y)$ (někdy označovaná jako edit distance), která je definovaná jako nejmenší počet transformací, které převedou řetězec **x** na řetězec **y**
- Transformace:
  - náhrada (substituce) symbolu **a** ∈ **T** v **x** symbolem **b** ∈ **T** v **y** **a≠b** (a→b)
  - vložení symbolu **a** ∈ **T** (ε→a) ε označuje prázdný symbol
  - zrušení symbolu **a** ∈ **T** (a→ ε)

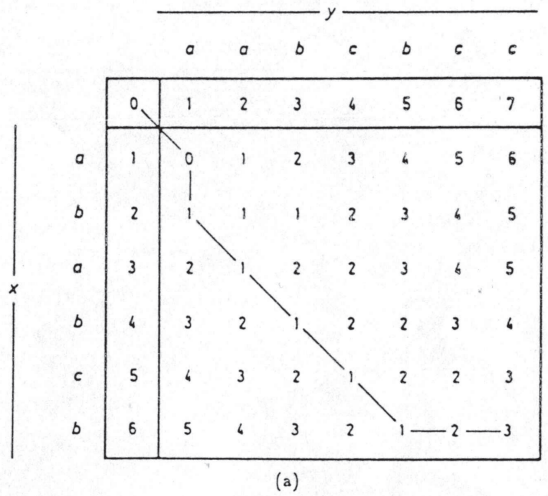
Algoritmus výpočtu vzdálenosti

```
input:
x = x_1 ... x_n ∈ T*, y = y_1 ... y_m ∈ T*, costs(a → b); a, b ∈ T ∪ {ε}
output:
d(x, y)
method:
0  begin
1  D(0,0) := 0;
2  for i = 1 to n do D(i,0) := D(i-1,0) + c(x(i) → ε);
3  for j = 1 to m do D(0,j) := D(0,j-1) + c(ε → y(j));
4  for i = 1 to n do
5      for j = 1 to m do
6          begin
7          m_1 := D(i-1,j-1) + c(x(i) → y(j));
8          m_2 := D(i-1,j) + c(x(i) → ε);
9          m_3 := D(i,j-1) + c(ε → y(j));
10         D(i,j) := min(m_1, m_2, m_3);
11         if m_1 = D(i,j) then set pointer from (i,j) to (i-1,j-1);
12         if m_2 = D(i,j) then set pointer from (i,j) to (i-1,j);
13         if m_3 = D(i,j) then set pointer from (i,j) to (i,j-1);
14         end;
15 d(x,y) := D(n,m);
16 end
```

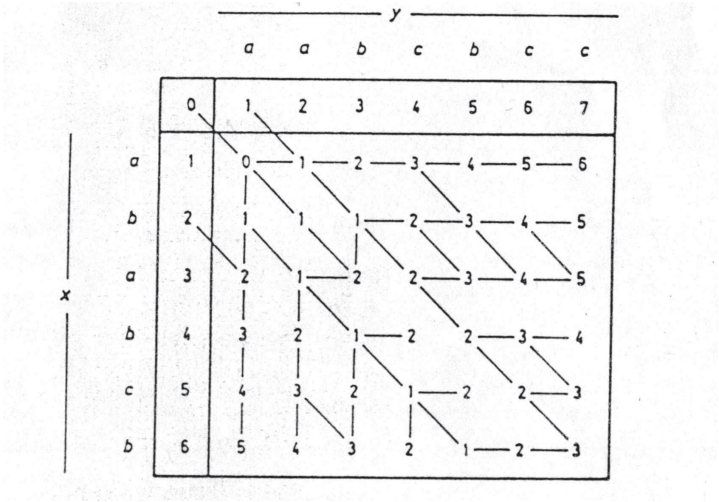
Matice pro výpočet vzdáleností

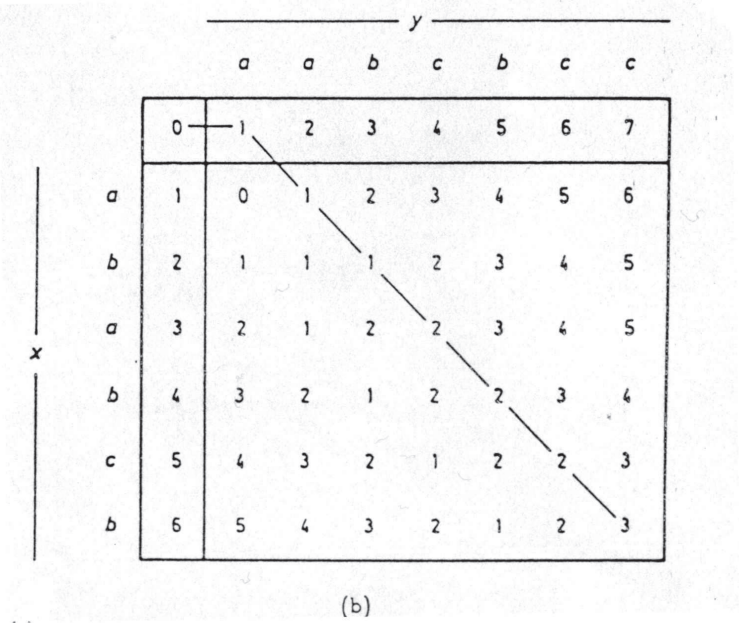


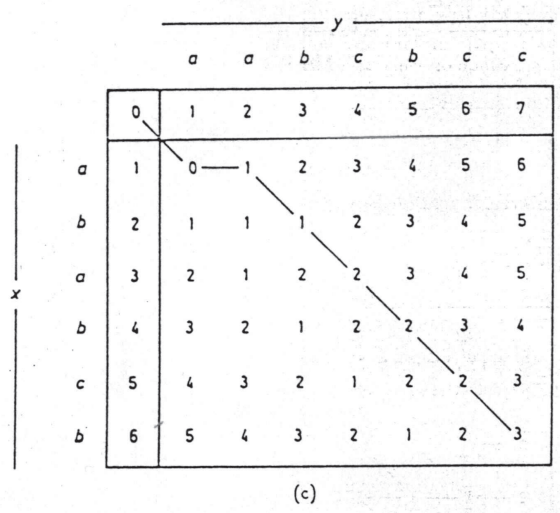
Rozdílné cesty, které vedou k úpravě řetězců



Příklad výpočtu vzdálenosti






y
a a b c b c c
0 1 2 3 4 5 6 7
x
a 1 0 1 2 3 4 5 6
b 2 1 1 1 2 3 4 5
a 3 2 1 2 2 3 4 5
b 4 3 2 1 2 2 3 4
c 5 4 3 2 1 2 2 3
b 6 5 4 3 2 1 2 3

(c)