

Obrada teksta

© Goodrich, Tamassia, Goldwasser

Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2019.

String

- **string** je niz karaktera
- primeri stringova:
 - Python program
 - HTML dokument
 - DNK sekvenca
 - digitalna slika
- **alfabet** Σ je skup mogućih karaktera za familiju stringova
- primeri alfabeta:
 - ASCII
 - Unicode
 - $\{0, 1\}$
 - $\{A, C, G, T\}$

String

- neka je P string dužine m
 - **podstring** $P[i..j]$ od P je podsekvenca od P koja sadrži karaktere sa rangom između i i j
 - **prefiks** od P je podstring tipa $P[0..i]$
 - **sufiks** od P je podstring tipa $P[i..m-1]$
- za date stringove T (tekst) i P (šablon, *pattern*) *pattern matching* problem je pronalaženje podstringa od T koji je jednak P
- primene:
 - editori teksta
 - mašine za pretragu (*search engines*)
 - bioinformatika

Nalaženje podstringa grubom silom

- nalaženje **grubom silom** (*brute force*) poredi šablon P sa tekstom T za svaki mogući položaj P u odnosu na T sve dok se
 - ne pronađe poklapanje
 - ne testiraju sve pozicije
- gruba sila radi u $O(nm)$ vremenu
- primer najgoreg slučaja:
 - $T = aaa \dots ah$
 - $P = aaah$
 - može da se pojavi u slikama i DNK sekvencama
 - retko u tekstovima

Nalaženje podstringa grubom silom

BruteForceMatch(T, P)

Input: tekst T dužine n i šablon P dužine m

Output: indeks početka podstringa u T jednakog P ili -1 ako nije pronađen

for $i \leftarrow 0$ **to** $n - m$ **do**

$j \leftarrow 0$

{testiramo položaj i }

while $j < m \wedge T[i + j] = P[j]$ **do**

$j \leftarrow j + 1$

if $j = m$ **then**

return i

{poklapanje na i }

else

break

return -1

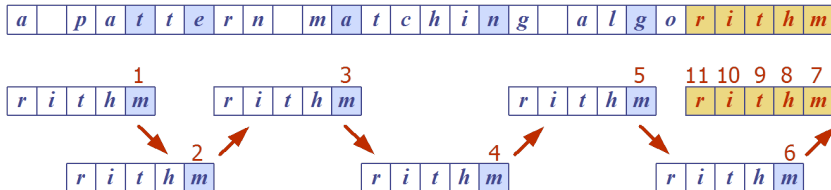
{nije pronađen}

Gruba sila u Pythonu

```
def find_brute_force(T, P):
    """Return the lowest index of T at which
       substring P begins (or else -1)."""
    n, m = len(T), len(P) # introduce convenient notations
    for i in range(n-m+1): # try every potential starting index within T
        k = 0               # an index into pattern P
        while k < m and T[i + k] == P[k]: # kth character of P matches
            k += 1
        if k == m:          # if we reached the end of pattern,
            return i        # substring T[i:i+m] matches P
    return -1              # failed to find a match starting with any i
```

Boyer-Moore

- Boyer-Moore algoritam se zasniva na dva principa:
 - ogledalo: poredi P sa podsekvencom u T idući unazad
 - skok: ako se razlika otkrije u $T[i] = c$



Boyer-Moore: funkcija poslednjeg pojavljivanja

- **Boyer-Moore** algoritam formira **last occurence** funkciju L koja mapira alfabet Σ na cele brojeve gde je $L(c)$ definisano kao
 - najveći indeks i takav da $P[i] = c$
 - -1 ako takvog indeksa nema
- primer: $\Sigma = \{a, b, c, d\}$ $P = abacab$

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

- može se predstaviti kao niz indeksiran numeričkim kodovima karaktera
- može se izračunati za $O(m + s)$ vreme gde je m dužina P a s je veličina Σ

Boyer-Moore

- **bad character shift**: preskoči sigurno neuspešna poređenja
- 0: ako P ne sadrži c : pomeri P tako da se poklope $P[0]$ i $T[i + 1]$
- 1: ako P sadrži c i poslednja pojava c je levo od pozicije i : pomeri P tako da se $T[i]$ poklopi sa poslednjom pojavom c u P
- 2: ako P sadrži c i poslednja pojava c je desno od pozicije i : pomeri P za jedno mesto

Boyer-Moore skok, slučaj 0

- slučaj 0 – ako P ne sadrži c :
- pomeri P tako da se poklope $P[0]$ i $T[i + 1]$

Text:

.	.	.	.	e	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern:

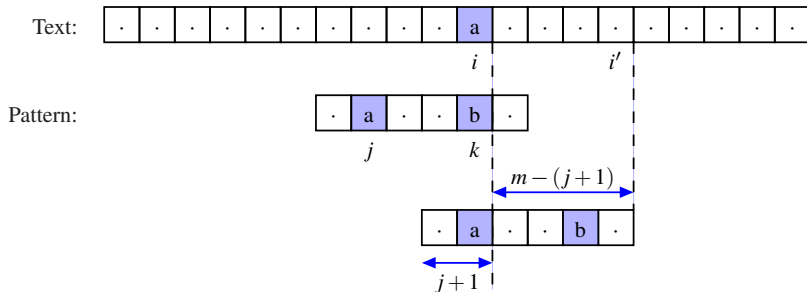
s	u	s	h	i
---	---	---	---	---

s	u	s	h	i
---	---	---	---	---

s	u	s	h	i
---	---	---	---	---

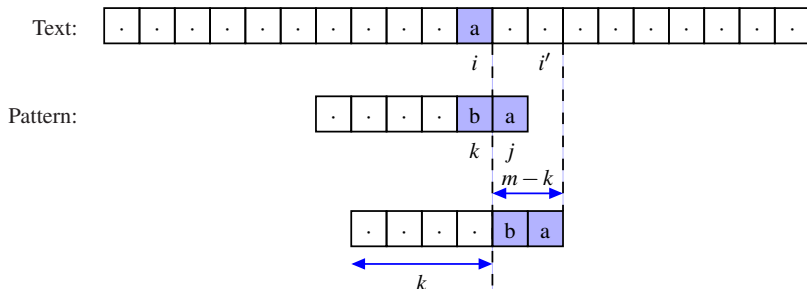
Boyer-Moore skok, slučaj 1

- slučaj 1 – ako P sadrži c i poslednja pojava c je levo od pozicije i :
- pomeri P tako da se $T[i]$ poklopi sa poslednjom pojavom c u P



Boyer-Moore skok, slučaj 2

- slučaj 2 – ako P sadrži c i poslednja pojava c je desno od pozicije i :
- rešenje A: pomeri P za jedno mesto
- rešenje B: pomeri P tako da se $T[i]$ poklopi sa sledećom pojavom c u P – last occurrence funkcija više nije dovoljna!



Boyer-Moore algoritam

BoyerMooreMatch(T, P, Σ)

$L \leftarrow \text{lastOccurence}(P, \Sigma)$

$i \leftarrow m - 1$

{indeks u T }

$j \leftarrow m - 1$

{indeks u P }

repeat

if $T[i] = P[j]$ **then**

if $j = 0$ **then**

return i

{poklapanje na i }

else

$i \leftarrow i - 1$

$j \leftarrow j - 1$

else

$l \leftarrow L[T[i]]$

{indeks poslednjeg pojavljivanja}

$i \leftarrow i + m - \min(j, 1 + l)$

{dva slučaja za skok}

$j \leftarrow m - 1$

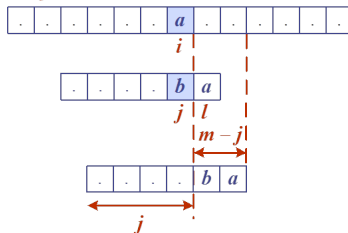
until $i > n - 1$

return -1

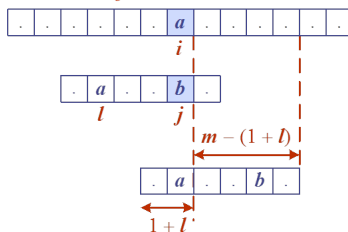
{nije pronađen}

Boyer-Moore

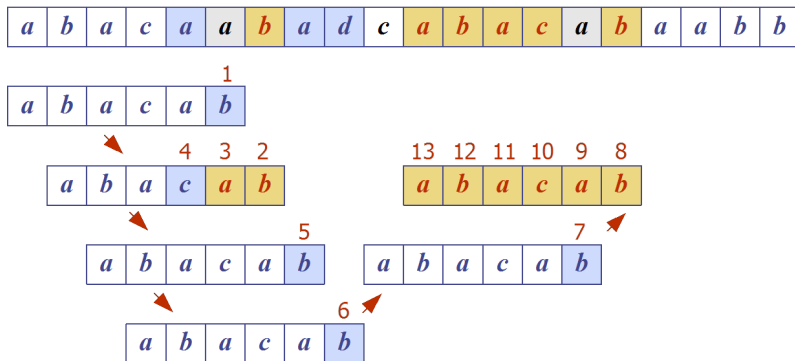
slučaj 1: $j \leq 1 + l$



slučaj 2: $1 + l \leq j$



Boyer-Moore: primer

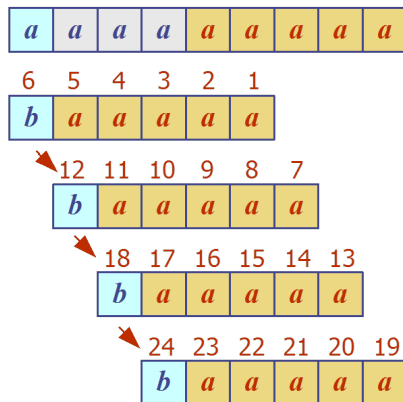


Boyer-Moore u Pythonu

```
def find_boyer_moore(T, P):
    """Return the lowest index of T at which substring P begins (or else -1)."""
    n, m = len(T), len(P)                # introduce convenient notations
    if m == 0: return 0                  # trivial search for empty string
    last = {}                             # build 'last' dictionary
    for k in range(m):
        last[ P[k] ] = k                  # later occurrence overwrites
    # align end of pattern at index m-1 of text
    i = m-1                               # an index into T
    k = m-1                               # an index into P
    while i < n:
        if T[i] == P[k]:                  # a matching character
            if k == 0:                    # pattern begins at index i of text
                return i
            else:
                i -= 1                     # examine previous character
                k -= 1                     # of both T and P
        else:
            j = last.get(T[i], -1)        # last(T[i]) is -1 if not found
            i += m - min(k, j + 1)         # case analysis for jump step
            k = m - 1                     # restart at end of pattern
    return -1
```


Boyer-Moore: analiza

- Boyer-Moore je $O(nm + s)$
- primer najgoreg slučaja:
 - $T = aaa \dots a$
 - $P = baaa$
- najgori slučaj nije verovatan u tekstovima
- znatno brži od grube sile za tekstove na prirodnom jeziku

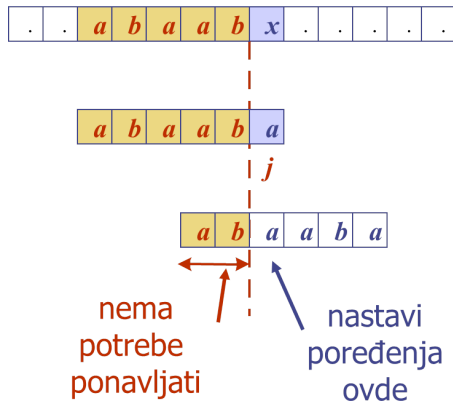


Boyer-Moore: analiza

- najgori slučaj za BM je $O(nm)$, isto kao i gruba sila
- za realne tekstove malo verovatan
- postoji i drugo pravilo za skok, **good suffix shift**, koje se zasniva na ideji koju koristi KMP algoritam (sledeći)

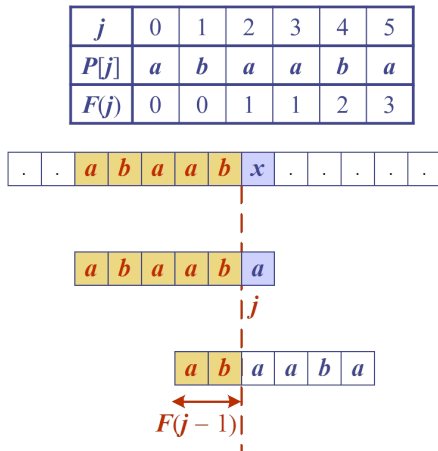
Knuth-Morris-Pratt

- **Knuth-Morris-Pratt** poredi tekst sa šablonom **slevo u desno** ali pomera šablon pametnije od grube sile
- kada se nađe razlika, koliko **najviše** možemo pomeriti šablon da izbegnemo suvišna poređenja?
- odgovor: najveći prefiks $P[0..j]$ koji je sufiks $P[1..j]$



KMP: funkcija neuspeha

- KMP analizira šablon da pronađe njegove prefikse unutar samog šablona
- funkcija neuspeha** $F(j)$ je veličina najvećeg prefiksa $P[0..j]$ takvog da je ujedno i sufiks $P[1..j]$
- ako nema poklapanja za $P[j] \neq T[i]$ pomeramo $j \leftarrow F(j-1)$



Knuth-Morris-Pratt algoritam

- funkcija neuspeha se može prikazati nizom koji se izračuna za $O(m)$
- u svakoj iteraciji petlje, ili
 - i se poveća za 1, ili
 - pomeraj $i - j$ se poveća za najmanje 1 (primeti da $F(j-1) < j$)
- \Rightarrow nema više od $2n$ iteracija u petlji
- \Rightarrow KMP je $O(m + n)$

KMPMatch(T, P)

```

 $F \leftarrow \text{failureFunction}(P)$ 
 $i \leftarrow 0$ 
 $j \leftarrow 0$ 
while  $i < n$  do
  if  $T[i] = P[j]$  then
    if  $j = m - 1$  then
      return  $i - j$  {poklapanje}
    else
       $i \leftarrow i + 1$ 
       $j \leftarrow j + 1$ 
    else
      if  $j > 0$  then
         $j \leftarrow F[j - 1]$ 
      else
         $i \leftarrow i + 1$ 
return  $-1$  {nije pronađen}
  
```

KMP: izračunavanje funkcije neuspeha

- funkcija neuspeha se može prikazati nizom koji se izračuna za $O(m)$
- slično kao i sam KMP algoritam
- u svakoj iteraciji petlje, ili
 - i se poveća za 1, ili
 - pomeraj $i - j$ se poveća za najmanje 1 (primeti da $F(j-1) < j$)
- \Rightarrow nema više od $2n$ iteracija u petlji

failureFunction(P)

```

 $F[0] \leftarrow 0$ 
 $i \leftarrow 1$ 
 $j \leftarrow 0$ 
while  $i < m$  do
  if  $P[i] = P[j]$  then
    {poklapa se  $j + 1$  znakova}
     $F[i] \leftarrow j + 1$ 
     $i \leftarrow i + 1$ 
     $j \leftarrow j + 1$ 
  else if  $j > 0$  then
    {koristi  $F$  da pomeriš  $P$ }
     $j \leftarrow F[j - 1]$ 
  else
     $F[i] \leftarrow 0$  {nema poklapanja}
     $i \leftarrow i + 1$ 

```

Knuth-Morris-Pratt: primer

a b a c a a b a c c a b a c a b a a b b

1 2 3 4 5 6
a b a c a b

7
a b a c a b

8 9 10 11 12
a b a c a b

13
a b a c a b

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

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

Knuth-Morris-Pratt u Pythonu ₁

```
def find_kmp(T, P):
    """Return the lowest index of T
       at which substring P begins (or else -1)."""
    n, m = len(T), len(P)          # introduce convenient notations
    if m == 0: return 0             # trivial search for empty string
    fail = compute_kmp_fail(P)      # rely on utility to precompute
    j = 0                            # index into text
    k = 0                            # index into pattern
    while j < n:
        if T[j] == P[k]:            # P[0:1+k] matched thus far
            if k == m - 1:          # match is complete
                return j - m + 1
            j += 1                  # try to extend match
            k += 1
        elif k > 0:                  # reuse suffix of P[0:k]
            k = fail[k-1]
        else:
            j += 1
    return -1                        # reached end without match
```


Knuth-Morris-Pratt u Pythonu 2

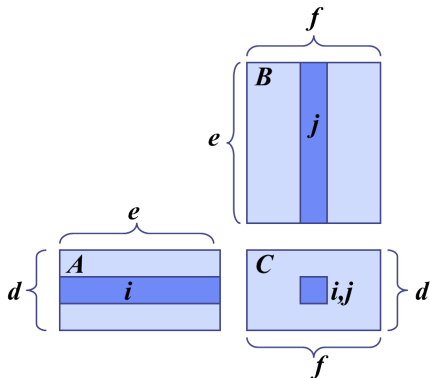
```
def compute_kmp_fail(P):
    """Utility that computes and returns KMP 'fail' list."""
    m = len(P)
    fail = [0] * m    # by default, presume overlap of 0 everywhere
    j = 1
    k = 0
    while j < m:      # compute f(j) during this pass, if nonzero
        if P[j] == P[k]: # k + 1 characters match thus far
            fail[j] = k + 1
            j += 1
            k += 1
        elif k > 0:    # k follows a matching prefix
            k = fail[k-1]
        else:          # no match found starting at j
            j += 1
    return fail
```

Dinamičko programiranje

- **dinamičko programiranje** je pristup dizajnu algoritama
- prvo primer: množenje matrica

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] \cdot B[k, j]$$

- vreme je $O(d \cdot e \cdot f)$



Množenje matrica

- računamo $A = A_0 \cdot A_1 \cdot \dots \cdot A_{n-1}$
- A_i ima dimenzije $d_i \times d_{i+1}$
- koji redosled množenja izabrati?
- primer:
 - B je 3×100
 - C je 100×5
 - D je 5×5
 - $(B \cdot C) \cdot D$ traži $1500 + 75 = 1575$ operacija
 - $B \cdot (C \cdot D)$ traži $1500 + 2500 = 4000$ operacija

Raspoređivanje zagrada / gruba sila

- traženje rešenja grubom silom: isprobati sve moguće kombinacije zagrada za $A = A_0 \cdot A_1 \cdot \dots \cdot A_{n-1}$
- izračunati broj operacija za svaku
- i izabrati najbolju
- vreme izvršavanja:
 - broj mogućih rasporeda zagrada je jednak broju različitih binarnih stabala sa n čvorova
 - **eksponencijalna** zavisnost!
 - tzv. Katalanov broj, iznosi skoro 4^n

Pohlepni pristup

- ideja #1: ponavljaj izbor onog proizvoda koji će imati **najviše** operacija
- protiv-primer:
 - A je 10×5
 - B je 5×10
 - C je 10×5
 - D je 5×10
 - ideja #1 daje $(A \cdot B) \cdot (C \cdot D)$,
tj. $500+1000+500 = 2000$ operacija
 - $A \cdot ((B \cdot C) \cdot D)$ je $500+250+250 = 1000$ operacija

Pohlepni pristup

- ideja #2: ponavlja izbor onog proizvoda koji će imati **najmanje** operacija
- protiv-primer:
 - A je 101×11
 - B je 11×9
 - C je 9×100
 - D je 100×99
 - ideja #2 daje $A \cdot ((B \cdot C) \cdot D)$,
tj. $109989 + 9900 + 108900 = 228789$ operacija
 - $(A \cdot B) \cdot (C \cdot D)$ je $9999 + 89991 + 89100 = 189090$ operacija
- pohlepni pristup ne donosi optimalan izbor

„Rekurzivni“ pristup

- definišemo **potprobleme**
 - nađi najbolji raspored zagrada za podniz $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$
 - neka je $N_{i,j}$ broj operacija za ovaj potproblem
 - optimalno rešenje za ceo problem je $N_{0,n-1}$
- **optimalnost potproblema**: optimalno rešenje se može dobiti pomoću optimalnih potproblema
 - mora postojati poslednje množenje (koren stabla izraza) za optimalno rešenje
 - neka je to na i -tom indeksu: $(A_0 \cdot \dots \cdot A_i) \cdot (A_{i+1} \cdot \dots \cdot A_{n-1})$
 - optimalno rešenje za ceo problem $N_{0,n-1}$ je suma dva optimalna potproblema plus poslednje množenje
 - ako bi optimalno rešenje imalo bolje potprobleme, ne bi bilo optimalno

Karakteristična jednačina

- globalni optimum se definiše pomoću optimalnih potproblema u zavisnosti od indeksa poslednjeg množenja
- razmotrimo sve moguće vrednosti tog indeksa
 - A_i je dimenzije $d_i \times d_{i+1}$
 - karakteristična jednačina za $N_{i,j}$ je:

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- potproblemi nisu nezavisni, nego se **preklapaju**

Algoritam dinamičkog programiranja

- pošto se problemi preklapaju, nećemo koristiti rekurziju
- konstruisaćemo optimalne potprobleme od dole na gore (*bottom-up*)
- $N_{i,i}$ je lako - nema množenja, tj. 0 operacija, počnemo od njih
- onda pređemo na potprobleme dužine 2, 3, ...
- vreme izvršavanja je $O(n^3)$

Algoritam dinamičkog programiranja

matrixChain(S)

Input: sekvenca S matrica koje treba pomnožiti

Output: broj operacija u optimalnom rasporedu zagrada

for $i \leftarrow 0$ **to** $n - 1$ **do**

$N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n - 1$ **do**

for $i \leftarrow 0$ **to** $n - b - 1$ **do**

$j \leftarrow i + b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$N_{i,j} \leftarrow \min_k \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

Python implementacija

```
def matrix_chain(d):
```

```
    """Return solution to the matrix chain problem.
```

d is a list of $n+1$ numbers describing the dimensions of a chain of n matrices such that k th matrix has dimensions $d[k]$ -by- $d[k+1]$.

Return an n -by- n table such that $N[i][j]$ represents the minimum number of multiplications needed to compute the product of A_i through A_j inclusive.

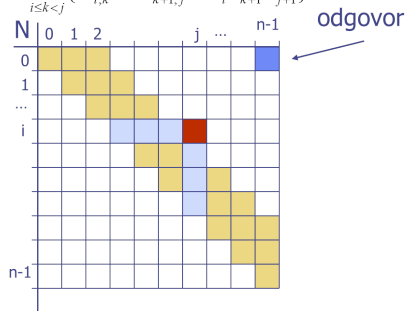
```
    """
```

```
    n = len(d) - 1                                # number of matrices
    N = [[0] * n for i in range(n)]               # initialize n-by-n result to zero
    for b in range(1, n):                          # number of products in subchain
        for i in range(n-b):                       # start of subchain
            j = i + b                              # end of subchain
            N[i][j] = min(N[i][k] + N[k+1][j] + d[i]*d[k+1]*d[j+1] \
                          for k in range(i,j))
    return N
```

Vizuelizacija algoritma

- bottom-up prvo popuni dijagonalu
- $N_{i,j}$ se dobija na osnovu vrednosti iz i -tog reda i j -te kolone
- popunjavanje svake ćelije u tabeli je $O(n)$
- ukupno vreme je $O(n^3)$
- raspored zagrada dobijamo pamćenjem k u ćelijama tabele

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



Opšti postupak dinamičkog programiranja

- primenljivo na probleme čije rešavanje traži puno vremena (moguće eksponencijalni) ukoliko postoje:
 - **jednostavni potproblemi**: potproblemi se mogu definisati pomoću promenljivih j, k, l, m itd.
 - **optimalni potproblemi**: globalni optimum se može definisati pomoću optimalnih potproblema
 - **preklapanje potproblema**: potproblemi nisu nezavisni i treba ih konstruisati bottom-up

Podsekvenca

- **podsekvenca** stringa $x_0x_1x_2 \dots x_{n-1}$ je string $x_{i_1}x_{i_2} \dots x_{i_k}$ gde je $i_j < i_{j+1}$
- nije isto što i podstring!
- primer stringa: ABCDEFGHIJK
 - jeste podsekvenca: ACEGIJK
 - jeste podsekvenca: DFGHK
 - nije podsekvenca: DAGH

Problem najduže zajedničke podsekvence

- longest common subsequence (LCS)
- za stringove X i Y , LCS je najduža podsekvence od X i Y
- primena: ispitivanje sličnosti DNK (alfabet je $\{A,C,G,T\}$)
- primer: ABCDEFG i XZACKDFWGH imaju LCS: ACDFG

LCS grubom silom

- primena grube sile na LCS:
 - pronađi sve podsekvence od X
 - izdvoj one koje su i podsekvence od Y
 - izaberi najdužu
- analiza:
 - ako je X dužine n , ima 2^n podsekvenci
 - ovo je eksponencijalno vreme!

LCS dinamičkim programiranjem

- neka je $L[i, j]$ LCS za $X[0..i]$ i $Y[0..j]$
- neka postoji indeks -1, tako da je $L[-1, k] = 0$ i $L[k, -1] = 0$; to znači da null deo X ili Y nema poklapanja sa drugim
- sada definišemo $L[i, j]$ u opštem slučaju:
 - ako je $x_i = y_j$ onda $L[i, j] = L[i - 1, j - 1] + 1$ (imamo poklapanje)
 - ako je $x_i \neq y_j$ onda $L[i, j] = \max\{L[i - 1, j], L[i, j - 1]\}$ (nemamo poklapanje)

LCS dinamičkim programiranjem

- slučaj $x_i = y_j$
- jednaki su poslednji znaci $X[0..i]$ i $Y[0..j]$
- taj poslednji znak mora biti deo najduže podsekvence (dokaz kontradikcijom)
- mora važiti $L[i, j] = L[i - 1, j - 1] + 1$
- primer: $L_{10,12} = 1 + L_{9,11}$



LCS dinamičkim programiranjem

- slučaj $x_i \neq y_j$
- LCS ne može da sadrži i x_i i y_j
- LCS može da sadrži jednog od njih ili nijednog
- sada važi $L[i, j] = \max\{L[i - 1, j], L[i, j - 1]\}$
- primer: $L_{9,11} = \max\{L_{9,10}, L_{8,11}\}$



LCS algoritam

LCS(X, Y)

Input: stringovi X i Y dužine n odnosno m

Output: $L[i, j]$ za $0 \leq i < n$ i $0 \leq j < m$

for $i \leftarrow 0$ **to** $n - 1$ **do**

$N_{i,-1} \leftarrow 0$

for $j \leftarrow 0$ **to** $m - 1$ **do**

$N_{-1,j} \leftarrow 0$

for $i \leftarrow 0$ **to** $n - 1$ **do**

for $j \leftarrow 0$ **to** $m - 1$ **do**

if $x_i = y_j$ **then**

$L[i, j] \leftarrow L[i - 1, j - 1] + 1$

else

$L[i, j] \leftarrow \max\{L[i - 1, j], L[i, j - 1]\}$

return L

LCS algoritam: vizualizacija

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	3	3	3	3	3
6	0	1	1	1	2	2	2	3	4	4	4	4	4
7	0	1	1	2	2	3	3	3	4	4	5	5	5
8	0	1	1	2	2	3	4	4	4	4	5	5	6
9	0	1	1	2	3	3	4	5	5	5	5	5	6
10	0	1	1	2	3	4	4	5	5	5	6	6	6

0 1 2 3 4 5 6 7 8 9
 X = **G** **T** **T** C C **T** A A T **A**
 Y = C **G** A **T** A A **T** T G **A** G A
 0 1 2 3 4 5 6 7 8 9 10 11

LCS algoritam: analiza

- algoritam ima dve ugnježdene petlje
 - spoljna ima n ciklusa
 - unutrašnja ima m ciklusa
 - telo unutrašnje petlje ima konstantno vreme
 - \Rightarrow ukupno vreme je $O(nm)$
- odgovor je sačuvan u $L[n, m]$

Python implementacija 1

```
def LCS(X, Y):
    """Return table such that L[j][k] is
       length of LCS for X[0:j] and Y[0:k].
       """
    n, m = len(X), len(Y)
    L = [[0] * (m+1) for k in range(n+1)] # (n+1) x (m+1) table
    for j in range(n):
        for k in range(m):
            if X[j] == Y[k]:                # align this match
                L[j+1][k+1] = L[j][k] + 1
            else:                            # choose to ignore one char
                L[j+1][k+1] = max(L[j][k+1], L[j+1][k])
    return L
```

Python implementacija 2

```
def LCS_solution(X, Y, L):
    """Return the longest common substring
    of X and Y, given LCS table L.
    """

    solution = []
    j,k = len(X), len(Y)
    while L[j][k] > 0: # common characters remain
        if X[j-1] == Y[k-1]:
            solution.append(X[j-1])
            j -= 1
            k -= 1
        elif L[j-1][k] >= L[j][k-1]:
            j -=1
        else:
            k -= 1
    return ''.join(reversed(solution)) # return left-to-right
                                         # version
```


Pohlepna metoda

- **pohlepna metoda** je pristup dizajnu algoritama zasnovan na:
 - **konfiguracije**: različiti izbori, kolekcije ili vrednosti koje treba pronaći
 - **funkcija cilja**: vrednost (*score*) dodeljena konfiguracijama koju želimo da minimizujemo ili maksimizujemo
- najbolje funkcioniše za probleme koji imaju osobinu **pohlepnog izbora**:
 - globalno optimalno rešenje se može pronaći serijom lokalnih unapređenja polazeći od početne konfiguracije

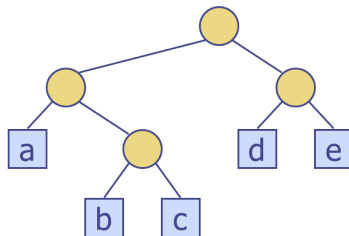
Kompresija teksta

- dati string X zapiši/kodiraj kao Y tako da Y zauzima manje memorije
 - štedi memoriju i/ili propusni opseg mreže
- odličan primer: **Huffman-ovo kodiranje**
 - izračunaj frekvenciju pojavljivanja svakog znaka
 - najčešće znakove kodiraj najkraćim kodovima
 - nijedan kôd nije prefiks nekog drugog
 - koristi optimalno stablo kodiranja za određivanje kodova

Stablo kodiranja

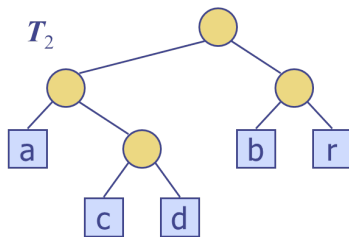
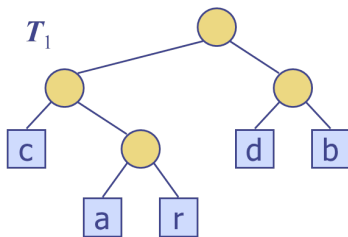
- **kôd** je preslikavanje karaktera iz alfabeta na binarni reprezent – kodnu reč
- **prefiksni kôd** je takav binarni kôd da nijedna kodna reč nije prefiks druge kodne reči
- **stablo kodiranja** predstavlja prefiksni kôd
 - listovi čuvaju karaktere iz alfabeta
 - kodna reč dobija se obilaskom putanje od korena do lista
 - 0 za levo dete i 1 za desno dete

00	010	011	10	11
a	b	c	d	e



Optimizacija stabla kodiranja

- za dati string X tražimo prefiksni kod takav da rezultat kompresije bude što kraći
 - česti karakteri treba da imaju kratke kodne reči
 - retki karakteri mogu da imaju duže kodne reči
- primer:
 - $X = \text{abracadabra}$
 - T_1 kodira X u 29 bita
 - T_2 kodira X u 24 bita



Huffman-ovo kodiranje

- za dati string X Huffman-ovo kodiranje konstruiše prefiksni kod koji minimizuje dužinu kôda od X
- radi u $O(n + d \log d)$ vremenu
 - n je dužina X
 - d je veličina alfabeta
- pomoćna struktura podataka: red sa prioritetom implementiran pomoću heapa

Huffman-ov algoritam

Huffman(X)

Input: string X dužine n sa d različitih znakova

Output: stablo kodiranja za X

izračunaj frekvenciju $f(c)$ za svaki znak c iz X

Q je novi red sa prioritetom

for all $c \in X$ **do**

kreiraj koren stabla T koji čuva c

dodaj T u Q sa ključem $f(c)$

while $\text{len}(Q) > 1$ **do**

$(f_1, T_1) \leftarrow Q.\text{remove_min}()$

$(f_2, T_2) \leftarrow Q.\text{remove_min}()$

kreiraj novo stablo T sa levim podstablom T_1 i desnim T_2

dodaj T u Q sa ključem $f_1 + f_2$

$(f, T) \leftarrow Q.\text{remove_min}()$

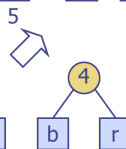
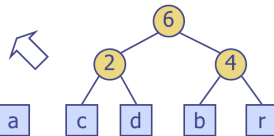
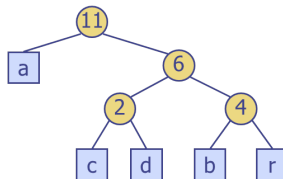
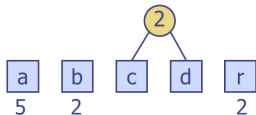
return T

Huffman: primer 1

$X = \text{abracadabra}$

Frekvencije

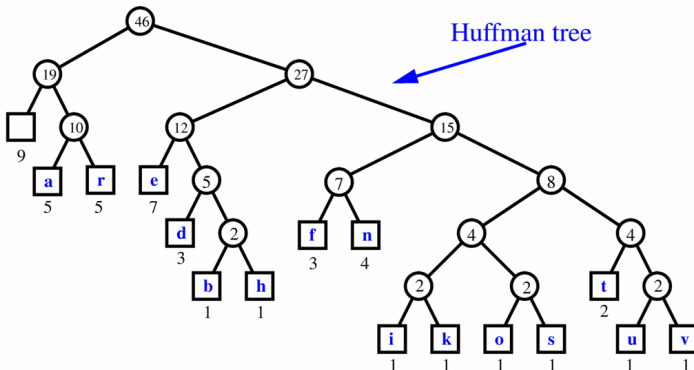
a	b	c	d	r
5	2	1	1	2



Huffman: primer 2

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

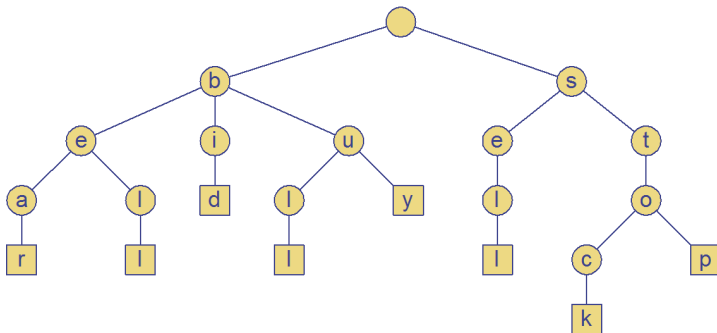


Predprocesiranje stringova

- predprocesiranje šablona ubrzava pattern matching
 - vreme za KMP je proporcionalno dužini teksta nakon predprocesiranja
- ako je tekst dugačak, ne menja se i često se pretražuje mogli bismo da predprocesiramo tekst umesto šablona
- **trie** (čita se kao „try“) je struktura podataka za čuvanje stringova, npr. svih reči u tekstu
 - vreme pretrage je proporcionalno dužini šablona

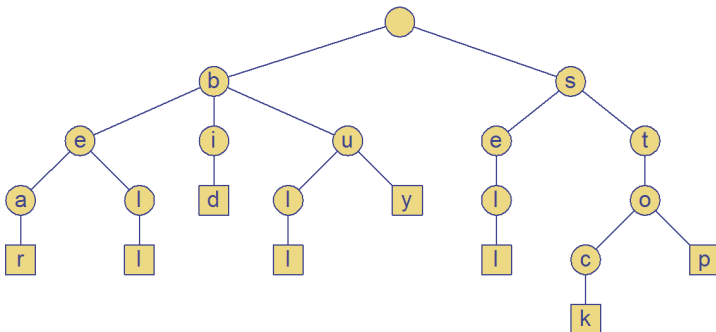
Standardni trie

- **standardni trie** za skup stringova S je stablo:
 - svaki čvor osim korena čuva jedan karakter
 - deca čvora su u alfabetskom redosledu
 - putanja od korena do lista daje čuvani string
- primer: $S = \{\text{bear, bell, bid, buy, sell, stock, stop}\}$



Standardni trie: analiza

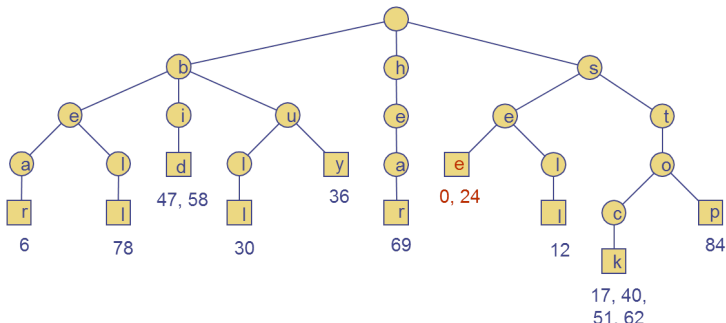
- standardni trie troši $O(n)$ prostora
- dodavanje, uklanjanje i pretraga su $O(dm)$
 - n ukupna dužina stringova u S
 - m dužina stringa u operaciji
 - d veličina alfabeta



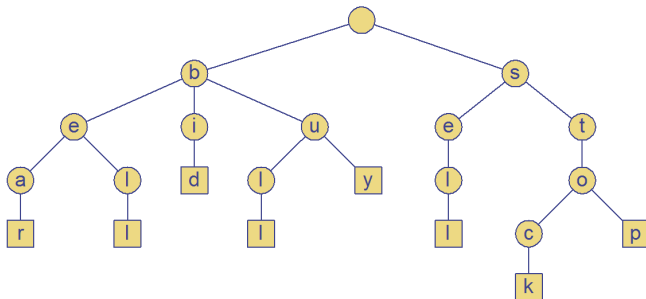
Traženje reči u trie

- dodaj reči iz teksta u trie
- svaki list je jedna reč
- list čuva indekse gde počinje reč

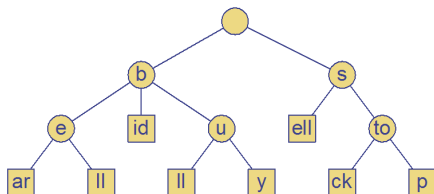
s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!		
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!				
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				



Kompresovani trie

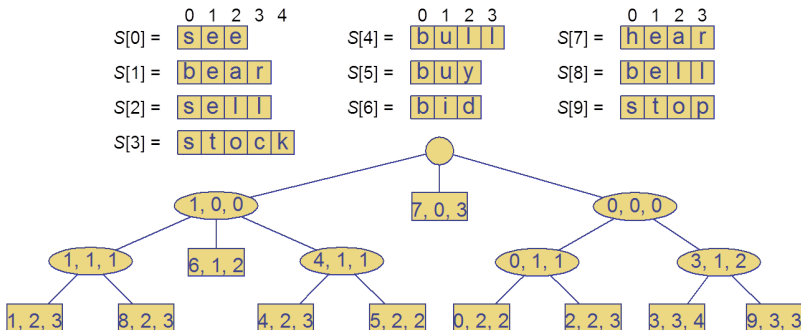


- **kompresovani trie** ima unutrašnje čvorove sa bar 2 deteta
- dobija se od standardnog kompresovanjem lanaca „redundantnih“ čvorova



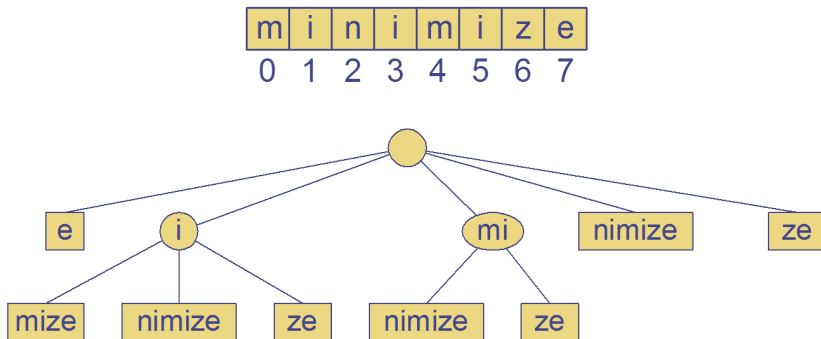
Kompaktna reprezentacija

- kompaktna reprezentacija kompresovanog trie-a za niz stringova
 - čvorovi čuvaju opsege indeksa umesto podstringove
 - troši $O(s)$ prostora gde je s broj stringova u nizu
 - služi kao pomoćna indeksna struktura



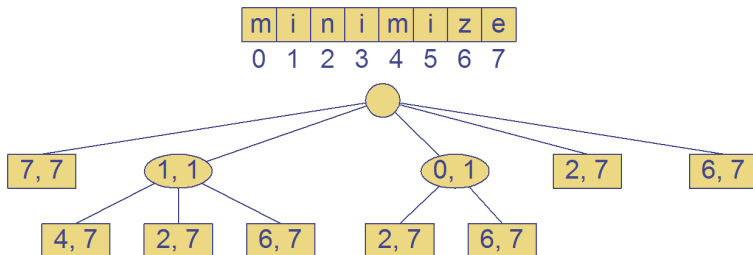
Sufiksni trie

- **sufiksni trie** stringa X je kompresovani trie svih sufiksa od X



Sufiksni trie: analiza

- string X dužine n , alfabet veličine d
 - troši $O(n)$ prostora
 - pretraga za $O(dm)$ vreme; m dužina traženog šablona
 - konstruiše se za $O(n)$ vreme



Kodni trie

- **kodni trie** predstavlja prefiksni kod
 - svaki list čuva karakter
 - kodna reč predstavlja putanju od korena do lista
 - 0 za levo dete, 1 za desno dete

00	010	011	10	11
a	b	c	d	e

