Algoritmi i strukture podataka

- predavanja -

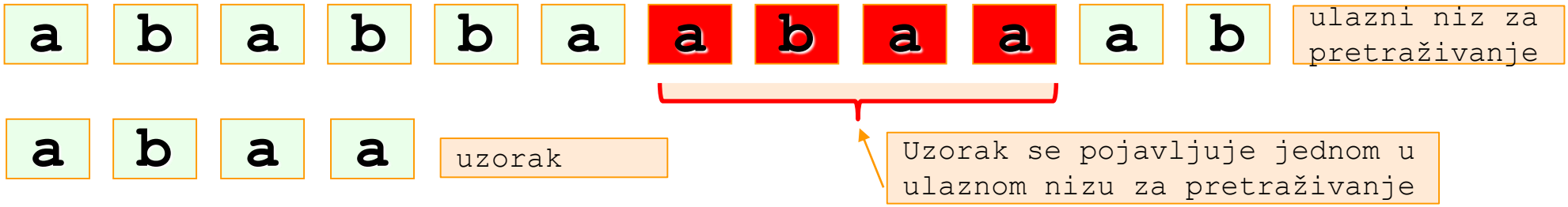
13. Pretraživanje znakovnih nizova

Pretraživanje znakovnih nizova (eng. string matching/searching)

- osnovni pojmovi
- osnovni algoritam pretraživanja znakovnih nizova (eng. naive/brute-force search)
- pretraživanje znakovnih nizova algoritmom Rabin-Karp
- pretraživanje znakovnih nizova algoritmom Knuth-Morris-Pratt

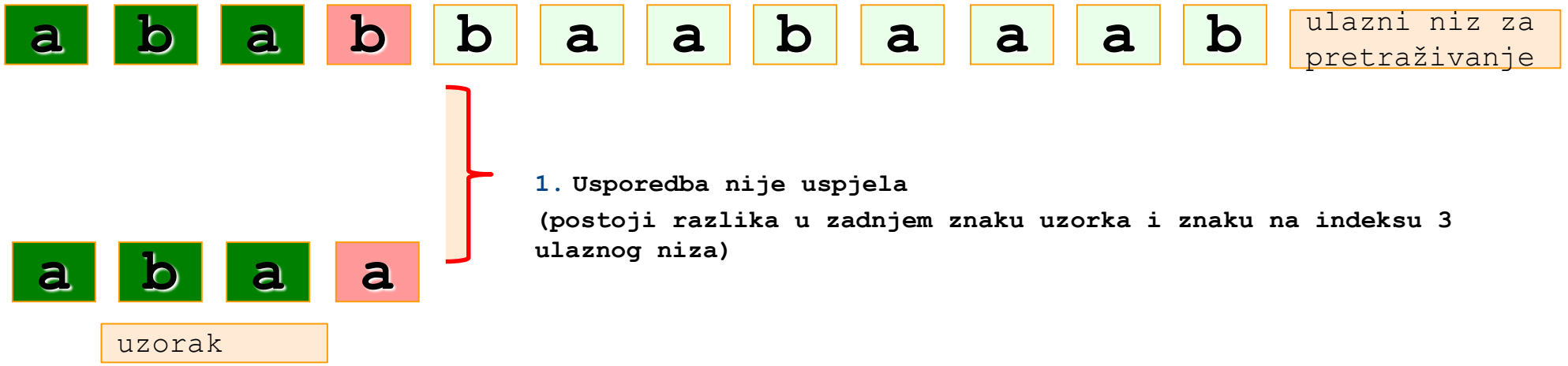
Pretraživanje znakovnih nizova (eng. string matching/searching)

- algoritmi pretraživanja znakovnih nizova pretražuju ulazni niz za pretraživanje (tipično velike duljine) u cilju pronalaska jednog ili više podnizova (tzv. uzoraka)
- i ulazni niz za pretraživanje i uzorak se sastoje od znakova neke abecede (konačnog skupa znakova)
  - npr. engleska abeceda (a-z), binarna abeceda (0-1), abeceda DNA (A,C,G,T)
- vrste algoritama pretraživanja znakovnih nizova:
  - algoritmi kod kojih ulazni niz za pretraživanje nije unaprijed poznat
    - samo ovi algoritmi se obrađuju na ovome kolegiju
  - algoritmi kod kojih uzorak koji se želi pronaći nije unaprijed poznat

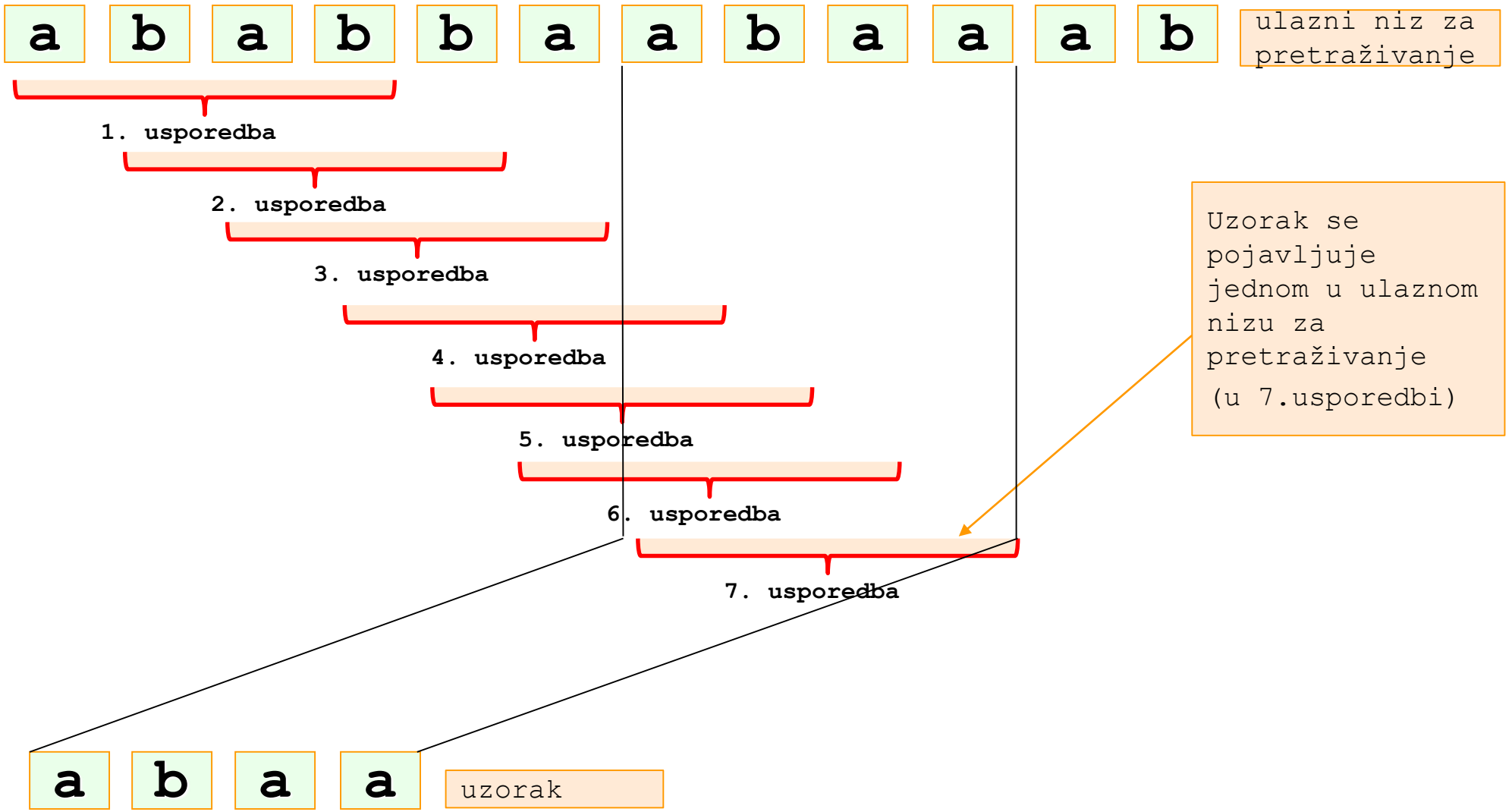


Osnovni algoritam pretraživanja znakovnih nizova (1)

- osnovni algoritam pretraživanja znakovnih nizova pretražuje iscrpno ulazni niz za pretraživanje
  - prolazi sve elemente ulaznog niza za pretraživanje
  - za svaki podniz ulaznog niza duljine uzorka za pretraživanje radi se usporedba s uzorkom za pretraživanje
  - algoritam ovisi o duljini ulaznog niza za pretraživanje: uzorak se uspoređuje onoliko puta koliko je ulazni niz za pretraživanje dugačak



Osnovni algoritam pretraživanja znakovnih nizova (2)



Osnovni algoritam pretraživanja znakovnih nizova - izvedba

- izvedba - 2 petlje:
  - vanjska služi za prolaz po elementima ulaznog niza za pretraživanje
    - indeksi idu od 0 do n-m (-m je zbog toga što se nakon indeksa n-m ima manje od m elemenata, pa usporedba s uzorkom duljine m nije moguća)
  - unutarnja služi za prolaz po m znakova uzorka i usporedbu

```
static size_t Search(string A, string pattern) {
      size_t i, j, n, m;
      n = A.length();
      m = pattern.length();
      for (j = 0; j <= n - m; j++) {
         for (i = 0; i < m && pattern[i] == A[i + j]; i++)
            ;
         if (i >= m)
            return j;
      }
      return -1;
   }
```

$$O(n * m)$$

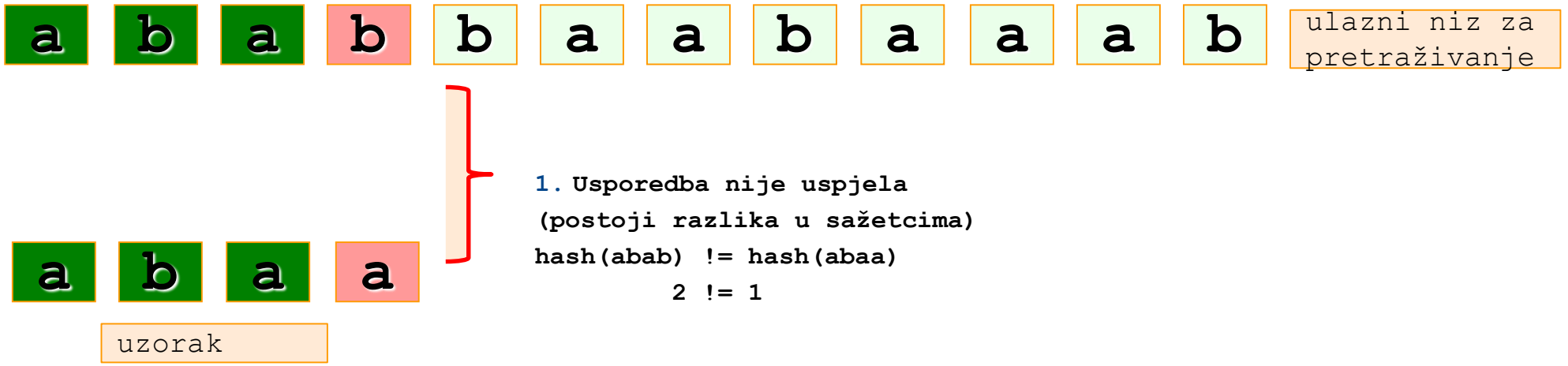
t_j

$$T(n) = \sum_{j=0}^{n-m} t_j$$

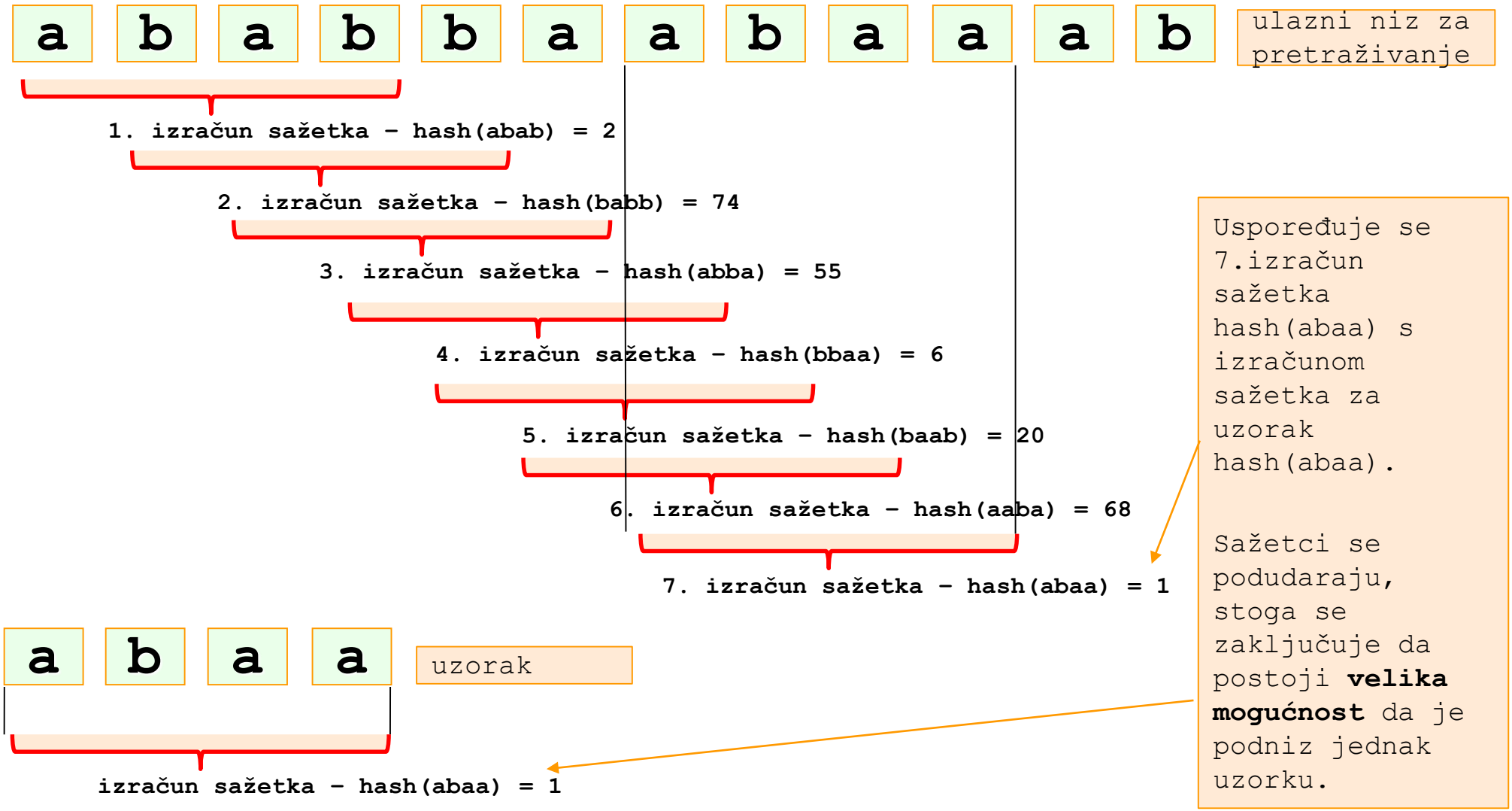
StringSearchNaive.cpp

Pretraživanje znakovnih nizova algoritmom Rabin-Karp (1)

- algoritam pretraživanja znakovnih nizova Rabin-Karp pretražuje ulazni niz za pretraživanje tako da računa sažetke podnizova
  - prolazi sve elemente ulaznog niza za pretraživanje
  - za svaki podniz ulaznog niza (veličine uzorka) za pretraživanje računa se **sažetak** i radi se usporedba sa sažetkom uzorka za pretraživanje
  - algoritam prvo se uspoređuje sažetke. **Samo ako se sažetci podudaraju**, uspoređuje se i podniz ulaznog niza sa uzorkom za pretraživanje. Stoga, algoritam ovisi o duljini ulaznog niza za pretraživanje.



Pretraživanje znakovnih nizova algoritmom Rabin-Karp (2)



Algoritam pretraživanja znakovnih nizova Rabin-Karp - izvedba

- izvedba - 2 petlje:
  - vanjska služi za prolaz po elementima ulaznog niza za pretraživanje
  - unutarnja služi za prolaz po m znakova uzorka i usporedbu (samo ako se sažetci podudaraju)

```
static int Search(string A, string pattern) {
     int i, j, n, m, h, p, t;
     n = A.length();
     m = pattern.length();
     p = GetHash(pattern, 0, m); // get hash for pattern
     t = GetHash(A, 0, m);
     for (i = 0; i <= n - m; i++) {
        if (p == t) {
              for (j = 0; j < m; j++) {
                 if (A[i + j] != pattern[j])
                       break;
              }
              if (j == m)
                    return i;
        }
        if (i < n - m) {
           t = ReHash(t, A[i], A[i + m], m);
        }
     }
     return -1;
}
```

t_i

$$O(n * m)$$

$$T(n) = \sum_{i=0}^{n-m} t_i$$

StringSearchRabinKarp.cpp

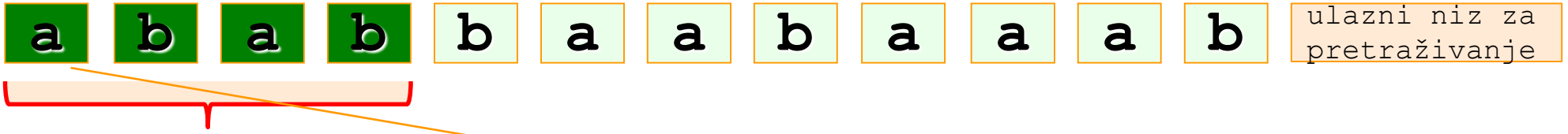
Izračun sažetka kod algoritma Rabin-Karp (1)

- izračun sažetka radi se za svaki podniz ulaznog niza koji je duljine uzorka
  - Takav izračun za sve podnizove nije optimalan
  - Rabin-Karpov algoritam uvodi poseban način izračuna sažetka tijekom izvođenja algoritma. Izračun sažetka **dva susjedna** niza duljine m:
    - `Sažetak(podniz(i,m)) = podniz[i+m-1]+BAZA*(podniz[i+m-2]+ ...+(BAZA*podniz[i])...)`
    - `Sažetak(podniz(i-1,m)) = podniz[i+m-2]+BAZA*(podniz[i+m-3]+ ...+(BAZA*podniz[i-1])...)`
  - Izrazi za dva susjedna niza se podudaraju u srednjim članovima sume, te je desni niz moguće izraziti putem lijevog:
    - `Sažetak(podniz(i,m)) = f(Sažetak(podniz(i-1,m))`
    - `Sažetak(podniz(i,m)) =`

    $(Sažetak(podniz(i-1,m)) - BAZA^{(m-1)} * podniz[i-1]) * BAZA + podniz[i+m-1]$
  - na navedeni način se umjesto m-1 operacije zbrajanja koristi 1 oduzimanje i **1 zbrajanje i 1 množenje s bazom potenciranom na m-1**
    - Izračun u konstantnom vremenu – baza na m-1 se izračuna jednom i zatim višestruko koristi

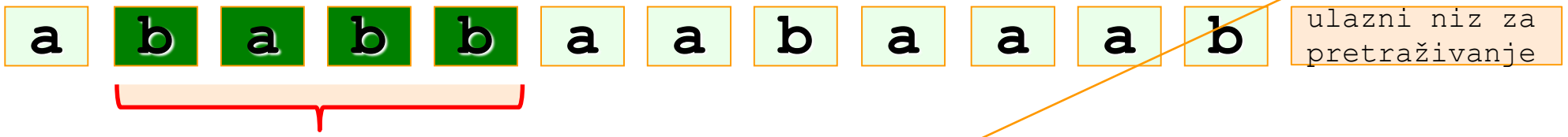
Izračun sažetka kod algoritma Rabin-Karp (2)

- s obzirom da se radi o pretraživanju znakovnih nizova, jedan element niza je znak duljine 8 bitova
  - BAZA = 2^8 = 256
  - vrijednost znaka je u rasponu [0, 255]



Sažetak(podniz(0,4))=Sažetak(abab)=ascii(a)*256^3+ascii(b)*256^2+ascii(a)*256^1+ascii(b)*256^0
=97*256^3+98*256^2+97*256^1+98*256^0=1633837410

Sažetak(podniz(1,4))=(Sažetak(podniz(0,4))-97*256^3)*256+98=(1633837410-97*256^3)*256+98=1650549346



Sažetak(podniz(1,4))=Sažetak(babb)=ascii(b)*256^3+ascii(a)*256^2+ascii(b)*256^1+ascii(b)*256^0
=98*256^3+97*256^2+98*256^1+98*256^0=1650549346

Izračun sažetka kod algoritma Rabin-Karp (3)

```
#define D 256
#define PRIME 251

static int GetHash(string s, int startIndex, int length) {
      int hash = 0;
      for (int i = 0; i < length; i++) {
         hash = (D * hash + s[i + startIndex]) % PRIME;
      }
      return hash;
}

static int ReHash(int oldHash, char leadingDigit, char lastDigit) {
      int oldHashWithoutLeadingDigit =
          oldHash - leadingDigit * mostSigDigitValue;
      int newHashwithoutLastDigit = oldHashWithoutLeadingDigit * D;
      int newHash = (newHashwithoutLastDigit + lastDigit) % PRIME;
      if (newHash < 0) // in case new hash is negative, covert to a positive num
         newHash += PRIME;
      return newHash;
}
```

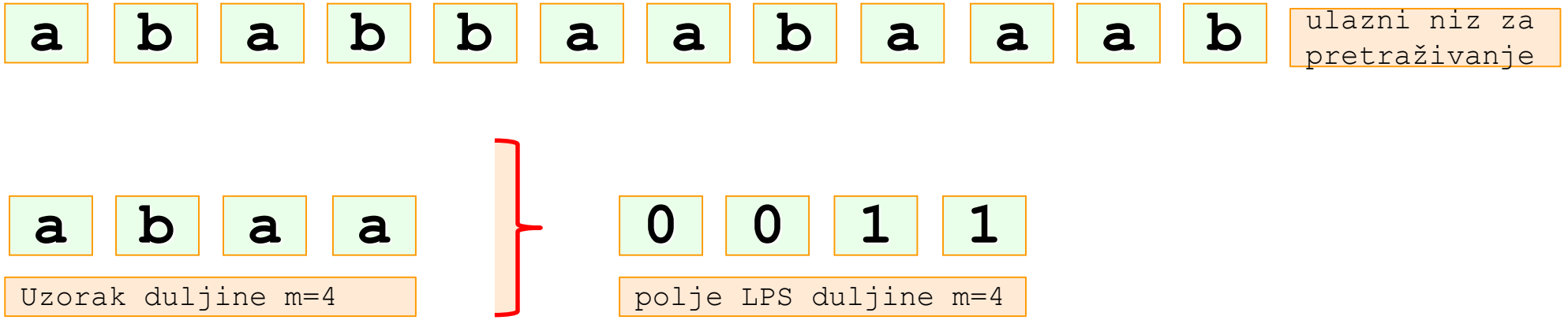
$\Theta(m)$

$\Theta(1)$

StringSearchRabinKarp.cpp

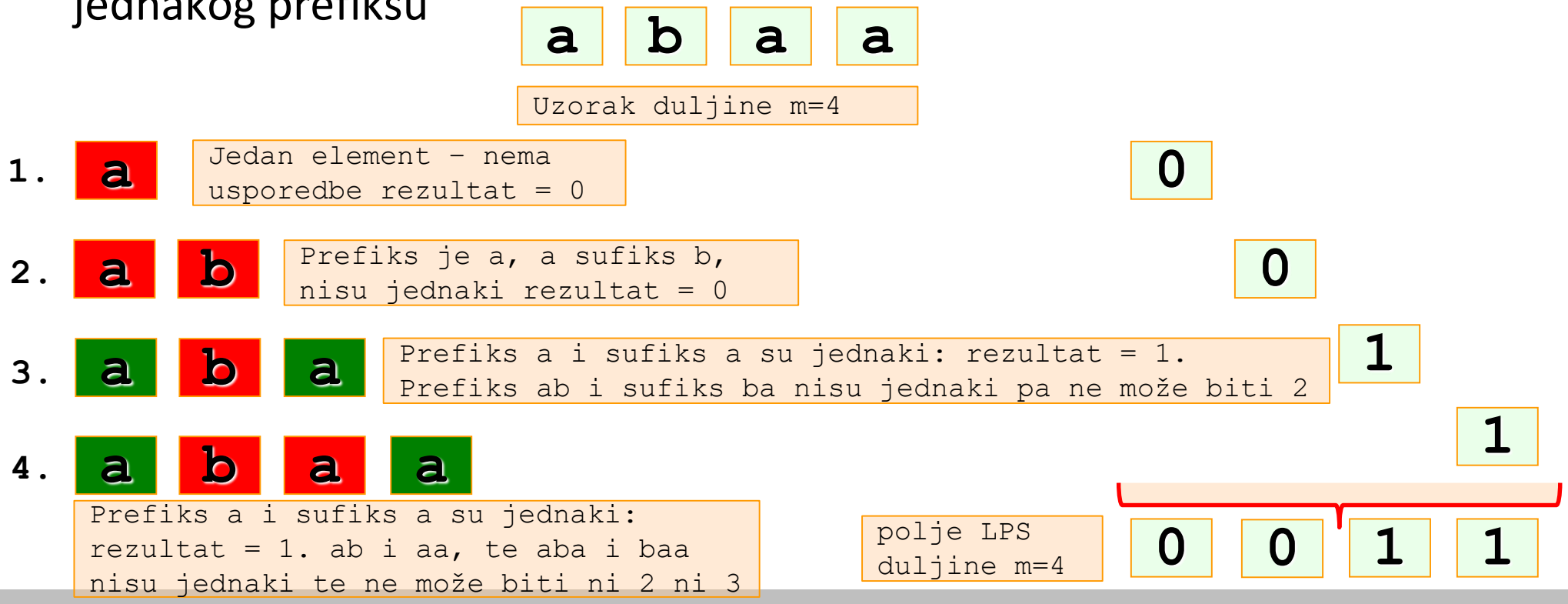
Pretraživanje znakovnih nizova algoritmom Knuth-Morris-Pratt (1)

- algoritam pretraživanja znakovnih nizova Knuth-Morris-Pratt pretražuje ulazni niz za pretraživanje slično osnovnom algoritmu, ali uz optimizaciju – tj. preskakanje usporedbi koje su viška
  - za postizanje navedenog preskakanja koristi te tzv. polje LPS (naziv dolazi od „*longest proper prefix which is also suffix*”)
  - polje LPS se gradi temeljem uzorka po kojem se radi pretraživanje
  - nakon što je polje LPS izgrađeno, kreće se s usporednom ulaznog niza, a uz pomoć polja LPS će se neke usporedbe moći preskočiti

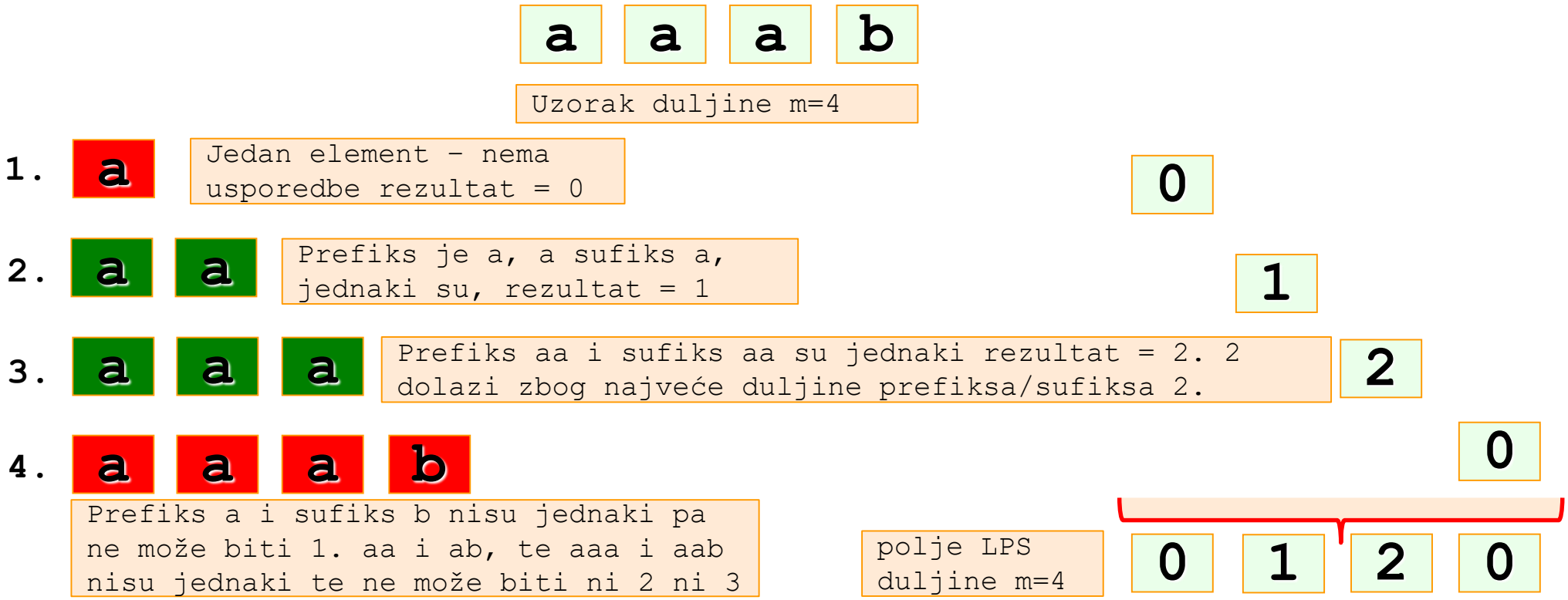


Gradnja polja LPS u algoritmu Knuth-Morris-Pratt – Primjer 1

- polje LPS se gradi temeljem ulaznog uzorka po kojem se pretražuje i veličine je iste kao i ulazni uzorak (m)
- polje LPS se gradi tako da se gledaju svi mogući prefiksi uzorka
- za svaki se takav prefiks računa kolika je duljina najduljenog sufiksa jednakog prefiksu



Gradnja polja LPS u algoritmu Knuth-Morris-Pratt – Primjer 2



Gradnja polja LPS u algoritmu Knuth-Morris-Pratt – kôd

```
static int *computeLPSArray(string pattern) {
     int *lps = new int[pattern.length()];
     int len = 0;
     lps[0] = 0;
     int i = 1;
     int m = pattern.length();
     while (i < m) {
        if (pattern[i] == pattern[len]) {
           len++;
           lps[i] = len;
           i++;
        } else {
           if (len != 0) {
              len = lps[len - 1];
           } else {
              lps[i] = 0;
              i++;
           }
        }
     }
     return lps;
  }
```

Ova grana kontrolne naredbe provjerava koliko je znakova prefiksa i sufiksa jednako. Na početku izvođenja su to znakovi jedni do drugih, ali kasnije pri izvođenju algoritma to mogu biti prefiks i sufiks koji su odvojeni – npr. AbaA (veliko A predstavlja odvojeni prefiks i sufiks)

$$\Theta(m)$$

Kada se identificira ne-podudaranje prefiksa i sufiksa postoje dvije mogućnosti: if: preskače se znak koji se ne podudara i nastavlja se usporedba prefiksa prije tog znaka i sufiksa nakon njega. else: na nekoj poziciji na indeksu i više nema prefiksa i sufiksa koji se podudaraju te se nastavlja analizirati uzorak.

StringSearchKnuthMorrisPratt.cpp

Izvođenje algoritma Knuth-Morris-Pratt – Primjer (1)

```
LPS array:0 1 2 0

New algorithm iteration started! i=0, j=0
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

A[i] and pattern[j] do not match and j > 0, change only j
j = lps[j-1] = lps[0] = 0

New algorithm iteration started! i=1, j=0
A = ababbaabaaab
pattern = aaab
A[i] and pattern[j] do not match and j = 0, do i++

New algorithm iteration started! i=2, j=0
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

A[i] and pattern[j] do not match and j > 0, change only j
j = lps[j-1] = lps[0] = 0

New algorithm iteration started! i=3, j=0
A = ababbaabaaab
pattern = aaab
A[i] and pattern[j] do not match and j = 0, do i++
```

Indeks i prati ulazni niz za pretraživanje, a indeks j prati uzorak za pretraživanje

Nađeno je nepodudaranje. Za nastavak se uzima prethodna vrijednost LPS (na indeksu 0). Ta vrijednost govori koliko znakova uzorka možemo preskočiti u usporedbi (koliko znakova se već smatra uspoređenim i jednakim).

StringSearchKnuthMorrisPratt.cpp

Izvođenje algoritma Knuth-Morris-Pratt – Primjer (2)

```
LPS array:0 1 2 0

New algorithm iteration started! i=4, j=0
A = ababbaabaaab
pattern = aaab
A[i] and pattern[j] do not match and j = 0, do i++

New algorithm iteration started! i=5, j=0
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

New algorithm iteration started! i=6, j=1
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

A[i] and pattern[j] do not match and j > 0, change only j
j = lps[j-1] = lps[1] = 1

New algorithm iteration started! i=7, j=1
A = ababbaabaaab
pattern = aaab
A[i] and pattern[j] do not match and j > 0, change only j
j = lps[j-1] = lps[0] = 0
```

U novoj iteraciji algoritma j postaje 1. Zbog toga se ne radi usporedba s početnim znakom uzorka (a). Ta usporedba nije ni potrebna jer nam LPS polje kaže da je već podudaranje postoji (zapisano u lps[1]).

Izvođenje algoritma Knuth-Morris-Pratt – Primjer (3)

```
New algorithm iteration started! i=7, j=0
A = ababbaabaaab
pattern = aaab
A[i] and pattern[j] do not match and j = 0, do i++

New algorithm iteration started! i=8, j=0
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

New algorithm iteration started! i=9, j=1
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

New algorithm iteration started! i=10, j=2
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++

New algorithm iteration started! i=11, j=3
A = ababbaabaaab
pattern = aaab
pattern[j] == A[i] - do j++ and i++
j == M. Pattern is found

Pattern found at position 8 (i-j = 12-4 = 8)
```

Algoritam Knuth-Morris-Pratt

```
static int Search(string A, string pattern) {
     int M = pattern.length();
     int N = A.length();
     int *lps = computeLPSArray(pattern);
     int i = 0; // index for A
     int j = 0; // index for pattern
     while (i < N) {
        if (pattern[j] == A[i]) {
           j++;
           i++;
        }
        if (j == M) {
           return i - j;
        } else if (i < N && pattern[j] != A[i]) {
           if (j != 0) {
              j = lps[j - 1];
           } else {
              i = i + 1;
           }
        }
     }
     return -1;
  }
```

$\Theta(n)$

Najviše 2n – 1 usporedbi znakova tijekom pretraživanja ulaznog niza

Maksimalni broj usporedbi za jedan znak <=$\log_x m$, gdje je x=(1+√ 5)/ 2 (zlatni rez)

StringSearchKnuthMorrisPratt.cpp

Usporedba

naziv	pretprocesiranje	usporedba
Osnovni algoritam	0 (nema)	O(n*m)
Rabin-Karp	Θ(m)	O(n*m) Prosječno Θ(n*m)
Knuth-Morris-Pratt	Θ(m)	Θ(n)