Univerzitet u Nišu

Prirodno-matematički fakultet

Departman za računarske nauke


# Algoritmi za pretragu stringova

Master rad

**Student:**

Ivan Stošić

**Mentor:**

Marko Petković

Niš
Septembar, 2019.

# Sadržaj

# 1 Uvod

## 1.1 Uvodne definicije

Neformalno, string je niz simbola iz nekog alfabeta. U opštem slučaju, alfabet može biti bilo koji konačan skup. Međutim, za potrebe pojedinih algoritama, potrebno je da alfabet bude totalno uređen skup. Kod implementacije algoritama, dodatno možemo pretpostaviti da je alfabet jednak nekom konačnom skupu uzastopnih celih brojeva, najčešće $\{0, 1, \ldots, k-1\}$ ili $1, 2, \ldots$,k za neko $k \in \mathbb{N}$. Tradicionalno, ovaj alfabet se označava grčkim slovom $\Sigma$.

**Definicija 1.1** *Ako je $\Sigma$ alfabet a $n$ prirodan broj, $\Sigma^n$ označava skup svih uređenih $n$-torki $(s_0, s_1, \ldots, s_{n-1})$, gde je $s_i \in \Sigma$ za svako $i \in \{0, \ldots, n-1\}$.*

Ovakvu $n$-torku možemo kraće zapisati sa $s$, a njenu dužinu (broj elemenata) sa $|s|$. Koristi se i kraći zapis $n$-torke: $s_0 s_1 \ldots s_{n-1}$.

**Definicija 1.2** *Ako je $\Sigma$ alfabet, onda je*

$$\Sigma^+ = \bigcup_{n=1}^{\infty} \Sigma^n$$

*skup svih nepraznih reči nad alfabetom $\Sigma$.*

Ovom skupu možemo dodati i praznu reč, koju označavamo sa $e$ ili $\epsilon$.

**Definicija 1.3** *Skup svih reči nad alfabetom $\Sigma$ je skup $\Sigma^* = \Sigma^+ \cup \{e\}$.*

Za string možemo definisati i njegove podstringove na sledeći način.

**Definicija 1.4** *Podstring počev od pozicije $l$, do pozicije $r$ ne uključujući $r$, nekog stringa $s$, gde važi $0 \leq l \leq r \leq |s|$ je string $s_l s_{l+1} \ldots s_{r-1}$. Ovaj podstring kraće zapisujemo i $s_{[l,r)}$.*

Podstringove stringa $s$ kod kojih je $l = 0$ nazivamo prefiksima tog stringa, dok podstringove kod kojih je $r = |s|$ nazivamo sufiksima tog stringa. Ukoliko je $l = r$ radi se o praznom podstringu. Ukoliko je podstring različit od celog stringa, onda se radi o pravom podstringu, sufiksu odnosno prefiksu. Izbor notacije sa indeksiranjem od nule i poluotvorenim intervalom olakšava implementaciju većine algoritama sa stringovima.

**Definicija 1.5** *Ciklični podstring stringa $s$ dužine $n$ počev od pozicije $l$ do pozicije $r$ je string $s_{[l,r)} = s_{l \bmod n} s_{(l+1) \bmod n} \cdots s_{(r-1) \bmod n}$.*

Ciklični podstring uopštava pojam podstringa. Zaista, ako je $0 \leq l \leq r \leq n$, ciklični podstring jednak je običnom podstringu.

**Definicija 1.6** *Ciklični pomeraj stringa $s$ dužine $n$ počev od pozicije $i$ je string $s_{[i,i+n)}$.*

Stringovi se mogu i nadovezivati odnosno konkatenirati. Skup $\Sigma^+$, odnosno $\Sigma^*$ zajedno sa operacijom konkatenacije čini algebarsku strukturu polugrupe, odnosno monoida.

**Definicija 1.7** *Ako su $s, p$ stringovi, tada je njihova konkatenacija string $sp = s_0 s_1 \ldots s_{|s|-1} p_0 p_1 \ldots p_{|p|-1}$.*

**Definicija 1.8** *Ako je $s$ string dužine $n$, sa $\overline{s}$ označavamo string $s_{n-1} s_{n-2} \ldots s_0$.*

**Definicija 1.9** *String $s$ je palindrom ako važi $s = \overline{s}$.*

Ukoliko je skup $\Sigma$ totalno uređen, definišemo leksikografsko poređenje stringova kao uređenje skupa $\Sigma^*$, na sledeći način.

**Definicija 1.10** *Za string $s$ kažemo da je leksikografski manji od stringa $p$ ukoliko postoji ceo broj $k \geq 0$ takav da je $k < \min\{|s|, |p|\}, s_{[0,k)} = p_{[0,k)}$ i $s_k < p_k$ ili ako je $s$ pravi prefiks stringa $p$.*

**Teorema 1.1** *Leksikografsko poređenje je totalno uređenje skupa $\Sigma^*$.* □

**Definicija 1.11** *Za svako $x \in \Sigma$, $Ord(x)$ je broj elemenata skupa $\Sigma$ koji su strogo manji od $x$.*

## 1.2 Složenost algoritama

Vreme, odnosno broj koraka i količina utrošene memorije tokom izvršenja nekog algoritma zavisi od ulaznih parametara. *Veliko O* notacija nam olakšava opisivanje i izračunavanje ovih funkcionalnih zavisnosti. Neka je u narednim definicijama domen funkcija $f, g$ skup $\mathbb{N}_0$ a kodomen $\mathbb{R}^+ \cup \{0\}$.

**Definicija 1.12** *Skup $O(g)$ definišemo kao skup svih funkcija $f$ za koje važi da postoje konstante $c$ i $n_0$ takve da je $f(n) \leq cg(n)$ za svako $n \geq n_0$.*

Ovu notaciju koristimo kada želimo da opišemo gornju granicu neke funkcije, do na proizvod sa konstantom. Problem ove notacije je upravo u tome što samo daje gornju granicu ponašanja neke funkcije. Zato se uvodi $\Theta$-notacija.

**Definicija 1.13** *Skup $\Theta(g)$ definišemo kao skup svih funkcija $f$ za koje važi da postoje pozitivne konstante $c_1, c_2$ i $n_0$ takve da je $c_1 g(n) \leq f(n) \leq c_2 g(n)$ za svako $n \geq n_0$.*

Za algoritam čiji je ulazni parametar $n$, što može biti broj elemenata niza, broj vrsta matrice, broj čvorova grafa, itd. kažemo da ima vremensku složenost $\Theta(g(n))$ odnosno $O(g(n))$ ako je $f$, gde je $f(n)$ broj elementarnih koraka tokom izvršenja algoritma, u skupu $\Theta(g)$ odnosno $O(g)$. Slično definišemo memorijsku složenost preko broja iskorišćenih elementarnih memorijskih lokacija.

## 1.3 Implementacije algoritama

Za sve implementacije biće korišćen programski jezik C++, kompajler GCC, verzija 9.1.0. Radi jednostavnosti, kodovi će biti dati bez `main` funkcije, `#include` direktiva i naredbe `using namespace std;`. Pretpostavlja se da su uključene sve standardne biblioteke, što se kod GCC-a može postići sa `#include <bits/stdc++.h>`.

# 2 Osnovni algoritmi pretrage

Jedan od osnovnih problema pretrage stringova je problem nalaženja svih pojavljivanja jednog stringa u drugom. Formalno, za data dva stringa $s, p$ dužina $n$ i $m$, redom, naći sve indekse $i$ takve da je $0 \leq i \leq n-m$ i $s_{[i,i+m)} = p$. String $s$ unutar kojeg se traži se često naziva tekstom, dok se $p$ naziva rečju (iako ne mora biti jedna reč) ili *pattern*-om odnosno šablonom. U literaturi na engleskom jeziku se često sreću i pojmovi *haystack* (plast sena) i *needle* (igla).

## 2.1 Naivna pretraga

Naivna pretraga rešava prethodno opisani problem tako što za svako celo $i$ iz segmenta $[0, n-m]$ upoređuje karakter po karakter stringove $s_{[i,i+m)}$ i $p$, pri čemu se odmah zaustavlja ukoliko naiđe na dva različita karaktera. Ovo poređenje ima vremensku složenost $O(m)$ pa ceo algoritam ima vremensku složenost $O((n-m)m)$, ova složenost se dostiže npr. za stringove koji se sastoje samo od slova `a`.

Implementacija naivne pretrage

```
1  vector<int> naive_search(const string& s, const string& p) {
2    int n = s.size(), m = p.size();
3    vector<int> result;
4    for (int i=0; i<=n-m; i++) {
5      bool ok = true;
6      for (int j=0; j<m; j++) {
7        if (s[i+j] != p[j]) {
8          ok = false;
9          break;
10       }
11     }
12     if (ok)
13       result.push_back(i);
14   }
15   return result;
16 }
```

## 2.2 Knuth-Morris-Pratt algoritam

KMP je prvi otkriveni algoritam koji rešava problem pretrage stringa u linearnom vremenu, odnosno u složenosti $O(n+m)$.[1] Da bismo razumeli rad algoritma, definišimo sledeće pojmove.

**Definicija 2.1** *Sufiks-prefiks nepraznog stringa $s$ je svaki string $p$ različit od $s$, uključujući i prazan string, koji je istovremeno i sufiks i prefiks stringa $s$, tj. $s_{[0,|p|)} = s_{[|s|-|p|,|s|)} = p$.*

Označimo sa $g(s)$ dužinu najdužeg sufiks-prefiksa stringa $s$. Za prazan string $s$ definišemo $g(s) = -1$. Ova funkcija zadovoljava sledeću, važnu osobinu:

**Teorema 2.1** *Neka je $s$ neprazan string. Neka je $s' = s_{[0,|s|-1)}$, tada je $g(s) \leq g(s') + 1$.*

*Dokaz.* Pretpostavimo suprotno, da je $g(s) > g(s')+1$. Neka je $p$ najduži sufiks prefiks stringa $s$. Neka je $p' = p[0, |p| - 1)$. Tada je $p'$ sufiks-prefiks stringa $s'$, pa je $g(s') \geq |p'| = |p| - 1 = g(s) - 1$, kontradikcija.

Ukoliko je $p$ najduži sufiks-prefiks stringa $s$, tada je i svaki drugi sufiks-prefiks stringa $s$ ujedno i sufiks-prefiks stringa $p$. Ova osobina nam omogućava da opišemo sve sufiks-prefikse nekog stringa $s$ kao lanac, gde je svaki naredni string najduži sufiks-prefiks prethodnog, sve dok se ne dođe do praznog stringa.

**Definicija 2.2** *Niz neuspeha stringa $s$ je niz $f_0, f_1, \ldots, f_{|s|}$, gde je $f_i = g(s[0, i))$.*

Prvi korak KMP algoritma je nalaženje niza neuspeha za string $p$. Prvo upisujemo $f_0 = -1$. Zatim, za svako $i > 0$, tražimo najduži sufiks-prefiks stringa $p_{[0,i-1)}$ koji se može proširiti slovom $p_{i-1}$, odnosno, nalazimo najveći broj $r$ takav da je $r$ dužina nekog sufiks-prefiksa stringa $p_{[0,i-1)}$ i važi $p_r = p_{i-1}$. Ukoliko ne nađemo takav broj $r$, onda je $f_i = 0$. U suprotnom, pošto je $p_{[0,r)} = p_{[i-1-r,i-1)}$ i $p_r = p_{i-1}$ imamo da je $p_{[0,r+1)} = p_{[i-r-1,i)}$, odnosno, $r+1$ je dužina jednog sufiks-prefiksa stringa $p_{[0,i)}$. Iz prethodnog razmatranja imamo da je ovo upravo najduži sufiks-prefiks stringa $p_{[0,i)}$, pa upisujemo $f_i = r + 1$. Niz svih sufiks-prefiksa možemo naći primenom opisane osobine lanca i činjenice da smo u $i$-tom koraku već izračunali dužine najdužih sufiks-prefiksa za sve prefikse stringa $p$ dužine manje od $i$.

**Primer.** Posmatrajmo string `atamatata`. Njegov niz neuspeha dat je u sledećoj tabeli. Indeksi u donjem redu su pomereni za jedno mesto da bi se bolje videlo o kom prefiksu je reč.

| | a | t | a | m | a | t | a | t | a |
|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |

Posmatrajmo šta se dešava pri računanju $f_i$ za pretposlednji prefiks, dužine 8. U prethodnom koraku smo imali string `atamata`, kome je najduži sufiks-prefiks string `ata`. Ako pokušamo da dodamo na njega slovo `t`, nećemo dobiti poklapanje jer je na poziciji 3 slovo `m`, zato pokušavamo sa narednim sufiks-prefiksom odnosno $r = f_3 = 1$. Sada uspešno dodajemo slovo `t` i upisujemo $r + 1 = 2$ u $f_8$.

Implementacija nalaženja niza neuspeha kod KMP algoritma

```
1  vector<int> kmp_ff(const string& p) {
2    int m = p.size();
3    vector<int> f(m+1);
4    f[0] = -1;
5    for (int i=1; i<=m; i++) {
6      int r = f[i-1];
7      while (r != -1 && p[r] != p[i-1])
8        r = f[r];
9      f[i] = r+1;
10   }
11   return f;
12 }
```

Ocenimo složenost ovog algoritma. Jasno je da složenost srazmerna zbiru broju iteracija ove dve petlje. Posmatrajmo vrednost $2i - r$. Nakon svake iteracije *while* petlje imamo da se $r$ smanjuje, jer je $f_r < r$ pa se $2i - r$ povećava. U jednoj iteraciji *for* petlje imamo da se i $i$ i $r$ povećavaju za tačno 1, pa se $2i - r$ ponovo povećava. Kako je početna vrednost $2i - 3$ jednaka 3, a važi $2i - r \leq 2m + 1$, ukupan broj iteracija, a samim tim i složenost algoritma je $O(m)$.

Opišimo sada glavni algoritam za traženje stringa. Algoritam za svaki prefiks $i$ stringa $s$ nalazi najduži sufiks koji je ujedno i prefiks stringa $p$. Ukoliko je dužina tog prefiksa jednaka $|p|$, tada dolazi do poklapanja na poziciji $i - |p|$. Neka je ta dužina jednaka $r_i$. Za $i = 0$ imamo $r_0 = 0$. Za svako $i > 0$, krećemo od prefiksa stringa $p$ dužine $r_{i-1}$ i proveravamo da li je naredno slovo jednako slovu $s_{i-1}$. Ako jeste, zaustavljamo se i upisujemo

dužinu nađenog prefiksa u $r_i$, u suprotnom, sledeći prefiks koji pokušavamo da produžimo je prefiks dužine $f_{r_{i-1}}$. Ovaj postupak ponavljamo sve dok ne dođemo do poklapanja ili do fiktivnog prefiksa $-1$, u tom slučaju dužina poklapanja je 0. Primetimo da je ovaj deo algoritma veoma sličan nalaženju niza neuspeha.

Implementacija glavnog dela KMP algoritma

```
1  vector<int> kmp_main(const string& s, const string& p) {
2    vector<int> f = kmp_ff(p), result;
3    int n = s.size(), m = p.size(), r = 0;
4    for (int i=1; i<=n; i++) {
5      while (r != -1 && p[r] != s[i-1])
6        r = f[r];
7      r++;
8      if (r == m) {
9        result.push_back(i-m);
10       r = f[r];
11     }
12   }
13   return result;
14 }
```

Na potpuno isti način se ocenjuje složenost glavnog dela algoritma, posmatranjem vrednosti izraza $2i - r$. Složenost je $O(n+m)$, zajedno sa prvom fazom, ukupna složenost je takođe $O(n+m)$.

Cela implementacija algoritma se može značajno pojednostaviti na sledeći način: Posmatrajmo string $p\$s$, gde je \$ karakter koji se ne javlja u stringovima $p, s$. Na osnovu njegovog niza neuspeha možemo da zaključimo gde se sve pojavljuje $p$ u $s$, tačnije, $i$ je pojavljivanje $p$ u $s$ akko je $f_{i+2m+1} = m$.

Pojednostavljena implementacija celog algoritma

```
vector<int> kmp_simple(const string& s, const string& p) {
  string q = p + '\0' + s;
  int n = s.size(), m = p.size();
  vector<int> f(n+m+2), result;
  f[0] = -1;
  for (int i=1; i<=n+m+1; i++) {
    int r = f[i-1];
    while (r != -1 && q[r] != q[i-1])
      r = f[r];
    f[i] = ++r;
    if (r == m && i >= 2*m+1)
      result.push_back(i-2*m-1);
  }
  return result;
}
```

## 2.3 Z-algoritam

Osnovna ideja ovog algoritma je da se izračuna Z-niz, koji se definiše na sledeći način.

**Definicija 2.3** *Za string $s$ dužine $n$, Z-niz je niz $z_1, \ldots, z_{n-1}$ gde je $z_i$ dužina najdužeg zajedničkog prefiksa za stringove $s$ i $s_{[i,n)}$.*

Z-niz se može iskoristiti za pretragu stringova. Ukoliko nađemo Z-niz za string $ps$, $i$ je pojavljivanje $p$ u $s$ akko je $z_{i+m} \geq m$.

Z-algoritam je efikasan algoritam za nalaženje Z-niza.[2] Označimo sa $q$ string za koji nalazimo Z-niz. Algoritam za svako $i$ direktno izračuna $z_i$ poklapanjem slova na pozicijama $z_i, i + z_i$. Ključna ideja je da se prethodno izračunate vrednosti Z-niza mogu iskoristiti da se postavi bolja početna vrednost za $z_i$, umesto da se svaki put kreće od nule. Naime, neka je $r$ najveća nađena vrednost izraza $j + z_j$ za $j < i$, a neka je pritom $l = j$ za koje se dostiže taj maksimum. Drugim rečima, $[l, r)$ je prozor koji odgovara nekom do sada pronađenom podstringu koji je jednak nekom prefiksu stringa, i to onom kod kojeg je $r$ najveće. Ukoliko važi $l \leq i < r$, tada znamo da je $q_{[i,r)} = q_{[i-l,r-l)}$. Takođe, važi $q_{[i-l,i-l+z_{i-l})} = q_{[0,z_{i-l})}$ pa, ako označimo sa $t_i = \min(z_{i-l}, r - i)$, važi $q_{[i,i+t_i)} = q_{[0,t_i)}$ odnosno $z_i \geq t_i$.

Implementacija Z-algoritma za pretragu stringa

```
vector<int> z_algorithm(const string& s, const string& p) {
  int n = s.size(), m = p.size();
  string q = p + s;
  vector<int> z(n+m, 0), result;
  for (int i=1, l=0, r=0; i<n+m; i++) {
    if (i < r)
      z[i] = min(z[i-l], r-i);
    while (i+z[i] < n+m && q[i+z[i]] == q[z[i]])
      z[i]++;
    if (i+z[i] > r)
      l = i, r = i+z[i];
    if (z[i] >= m)
      result.push_back(i-m);
  }
  return result;
}
```

Dokažimo da ovaj algoritam ima složenost $O(n+m)$. Očigledno, kritična je unutrašnja *while* petlja. Dokazaćemo da svaka iteracija while petlje odgovara povećanju vrednosti promenljive $r$ za bar 1. Posmatrajmo sledeće slučajeve:

- Pre ulaska u *while* petlju važi $i \geq r$. U ovom slučaju $z_i$ ima početnu vrednost nula, na kraju *while* petlje će važiti $i + z_i \geq r$ pa će $r$ dobiti vrednost $i + z_i$, odnosno $r$ će se povećati za barem $z_i$, što je veće ili jednako od broja iteracija *while* petlje.

- Pre ulaska u *while* petlju važi $i < r$ i $z_{i-l} < r - i$. Sada $z_i$ dobija početnu vrednost $z_{i-l}$. Dokažimo da će *while* petlja izvršiti tačno nula iteracija. Pretpostavimo suprotno, da je $q_{i+z_i} = q_{z_i}$, odnosno $q_{i+z_{i-l}} = q_{z_{i-l}}$. Iz definicije prozora $[l, r)$ imamo da je $q_{[0,r-l)} = q_{[l,r)}$, pošto je $l \leq i + z_{i-l} < r$, odnosno, ova pozicija je unutar prozora, važi $q_{i+z_{i-l}} = q_{i-l+z_{i-l}}$, odnosno, po pretpostavci, $q_{z_{i-l}} = q_{i-l+z_{i-l}}$, što znači da $z_{i-l}$ ima pogrešno izračunatu vrednost jer se poklapaju karakteri na kraju odgovarajućih podstringova, što dovodi do kontradikcije.

- Pre ulaska u *while* petlju važi $i < r$ i $z_{i-l} \geq r - i$. Sada $z_i$ dobija početnu vrednost $r - i$. Ako petlja izvrši $k$ iteracija, na kraju će važiti $z_i = k + r - i$ odnosno $i + z_i = k + r$, što znači da će nova vrednost $r$ biti za bar $k$ veća.

Kako je $r \leq n + m$ zaključujemo da je ukupna složenost $O(n + m)$.

# 3 Dinamičko programiranje nad stringovima

Dinamičko programiranje je tehnika rešavanja optimizacionih problema i problema prebrojavanja gde se glavni problem rešava tako što se identifikuju slični potproblemi manje veličine, koji se zatim rešavaju i čija se rešenja kombinuju u rešenje glavnog problema. Svaki ovako dobijeni potproblem se rešava najviše jedanput, nakon čega se njegovo rešenje pamti u memoriji.

Prvi korak u primeni dinamičkog programiranja na rešavanje nekog problema jeste da se identifikuju *potproblemi* tog problema. Najveća instanca među svim potproblemima jeste *glavni problem*. Najmanje instance su *trivijalni potproblemi*, koji se ne dele dalje na potprobleme i čija rešenja se dobijaju na neki drugi način. Zatim je neophodno, za svaki potproblem naći relaciju između njega i jednog ili više manjih potproblema. Ova relacija odnosno rešenje za potproblem je matematički izraz ili rezultat nekog jednostavnog algoritma u kojem figurišu rešenja manjih potproblema. Kažemo da potproblem $A$ zavisi od potproblema $B$ ako rešenje potproblema $B$ figuriše u izrazu koji je rešenje potproblema $A$. Da bismo našli rešenje nekog potproblema neophodno je da prethodno nađemo rešenja za sve potprobleme od kojih on zavisi.

## 3.1 LCP matrica

Jedna od jednostavnijih primena dinamičkog programiranja jeste izračunavanje LCP matrice stringa. $LCP(s, p)$ (od *longest common prefix*) je dužina najdužeg zajednički prefiksa stringova $s, p$.

**Definicija 3.1** *LCP matrica za string $s$ dužine $n$ je kvadratna matrica $A$ za koju važi $A_{i,j} = LCP(s_{[i,n)}, s_{[j,n)})$.*

Ukoliko važi $s_i \neq s_j$, onda je $A_{i,j} = 0$. U suprotnom, posmatrajmo stringove $s_{[i+1,n)}, s_{[j+1,n)}$. Važi da je $p$ njihov zajednički prefiks ako i samo ako je $s_i p = s_j p$ zajednički prefiks stringova $s_{[i,n)}, s_{[j,n)}$, odakle dobijamo da je $A_{i,j} = A_{i+1,j+1} + 1$, pod uslovom da su indeksi $i + 1, j + 1$ validni, inače se radi o praznim podstringovima pa možemo smatrati da je $A_{i+1,j+1} = 0$ odnosno $A_{i,j} = 1$. Ovo nas dovodi do sledećeg, jednostavnog algoritma za računanje LCP matrice.

Implementacija algoritma za nalaženje LCP matrice

```
1  vector<vector<int>> lcp_matrix(const string& s) {
2    int n = s.size();
3    vector<vector<int>> a(n, vector<int>(n));
4    for (int i=n-1; i>=0; i--)
5      for (int j=n-1; j>=0; j--)
6        if (s[i] != s[j])
7          a[i][j] = 0;
8        else if (i != n-1 && j != n-1)
9          a[i][j] = a[i+1][j+1] + 1;
10       else
11         a[i][j] = 1;
12   return a;
13 }
```

Vremenska i memorijska složenost ovog algoritma je $O(n^2)$.

## 3.2 Najduži zajednički podniz

**Definicija 3.2** *Podniz stringa $s$ dužine $n$ je string $s_{i_0}s_{i_1}\ldots s_{i_{k-1}}$ gde važi $0 \leq s_0 < s_1 < \ldots < i_{k-1} < n$.*

Dužina najdužeg zajedničkog podniza (LCS, od *longest common subsequence*) dva stringa $s, p$ se može koristiti kao mera njihove sličnosti. Preciznije, ukoliko su dužine $s, p$ redom $n, m$ a dužina njihovog LCS-a je $k$, tada je njihova udaljenost $n + m - 2k$, gde se udaljenost odnosi na minimalan broj izmena potrebnih da se od stringa $s$ dobije string $p$, gde su dozvoljene operacije brisanje jednog slova i umetanje jednog slova u string.

Neka je za fiksne stringove $s, p$ dužina $n, m$, redom, $d_{i,j}$ dužina LCS-a za stringove $s_{[0,i)}, p_{[0,j)}$.

- Ako je $i = 0$ ili $j = 0$, tada je $d_{i,j} = 0$.
- U suprotnom, posmatrajmo slova $s_{i-1}$ i $p_{j-1}$. Ukoliko su ona jednaka, tada se svaki zajednički podniz stringova $s_{[0,i-1)}, p_{[0,j-1)}$ može proširiti za karakter $s_{i-1} = p_{j-1}$ tako da se dobije zajednički podniz stringova $s_{[0,i)}, p_{[0,j)}$. Inače, svaki zajednički podniz stringova $s_{[0,i)}, p_{[0,j)}$ je ili zajednički podniz za $s_{[0,i-1)}, p_{[0,j)}$ ili za $s_{[0,i)}, p_{[0,j-1)}$.

Odavde dobijamo sledeću rekurentnu vezu: Ako su $i, j > 0$,

$$d_{i,j} = \begin{cases} \max(d_{i-1,j}, d_{i,j-1}) & s_{i-1} \neq p_{j-1} \\ \max(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1} + 1) & s_{i-1} = p_{j-1} \end{cases} \tag{3.2.1}$$

Jasno je da je $d_{n,m}$ dužina LCS-a za cele stringove $s, p$.

Implementacija nalaženja dužine LCS-a

```
1  int lcs_len(const string& s, const string& p) {
2    int n = s.size(), m = p.size();
3    vector<vector<int>> d(n+1, vector<int>(m+1));
4    for (int i=0; i<=n; i++)
5      for (int j=0; j<=m; j++)
6        if (i == 0 || j == 0)
7          d[i][j] = 0;
8        else if (s[i-1] != p[j-1])
9          d[i][j] = max(d[i-1][j], d[i][j-1]);
10       else
11         d[i][j] = max({d[i-1][j], d[i][j-1], d[i-1][j-1]+1});
12   return d[n][m];
13 }
```

Ako posmatramo $d$ kao matricu, primećujemo da se vrednosti u $i$-toj vrsti mogu izračunati samo na osnovu stringova $s, p$ i vrednosti u $i$-toj i vrsti $i-1$. Ovo nam omogućava da implementiramo algoritam tako da se u svakom trenutku pamte samo poslednje dve vrste.

Implementacija nalaženja dužine LCS-a sa $O(m)$ memorije

```
1  int lcs_len_mem(const string& s, const string& p) {
2    int n = s.size(), m = p.size();
3    vector<int> di(m+1, 0), dim1(m+1);
4    for (int i=1; i<=n; i++) {
5      swap(di, dim1);
6      for (int j=1; j<=m; j++)
7        if (s[i-1] != p[j-1])
8          di[j] = max(dim1[j], di[j-1]);
9        else
10         di[j] = max({dim1[j], di[j-1], dim1[j-1]+1});
11   }
12   return di[m];
13 }
```

Ukoliko je potrebno naći ceo podniz a ne samo njegovu dužinu, rekonstrukciju radimo kretanjem unazad kroz matricu $d$, uvek idući ka odgovarajućem polju koje ima najveću vrednost, odnosno, ako smo u polju $i, j$ idemo ka polju od kojeg je $d_{i,j}$ "uzelo" vrednost.

Implementacija nalaženja LCS-a

```
1  string lcs_string(const string& s, const string& p) {
2    int n = s.size(), m = p.size();
3    vector<vector<int>> d(n+1, vector<int>(m+1));
4    for (int i=0; i<=n; i++)
5      for (int j=0; j<=m; j++)
6        if (i == 0 || j == 0)
7          d[i][j] = 0;
8        else if (s[i-1] != p[j-1])
9          d[i][j] = max(d[i-1][j], d[i][j-1]);
10       else
11         d[i][j] = max({d[i-1][j], d[i][j-1], d[i-1][j-1]+1});
12
13   int i = n, j = m;
14   string q;
15   while (i > 0 && j > 0) {
16     if (s[i-1] == p[j-1]) {
17       if (d[i][j] == d[i-1][j-1] + 1)
18         q += s[i-1], i--, j--;
19       else if (d[i][j] == d[i-1][j])
20         i--;
21       else
22         j--;
23     } else {
24       if (d[i][j] == d[i-1][j])
25         i--;
26       else
27         j--;
28     }
29   }
30   reverse(q.begin(), q.end());
31   return q;
32 }
```

### 3.2.1 Hirschberg-ov algoritam

Iako se algoritam koji samo nalazi dužinu LCS-a dva stringa suštinski ne razlikuje od onog koji nalazi ceo taj podniz, prvi se može jednostavno realizovati tako da mu je memorijska složenost $O(m)$. Hirschberg-ov algoritam nalazi ceo

LCS u memorijskoj složenosti $O(n+m)$ bez žrtvovanja vremenske složenosti.[3] Ideja algoritma je da string $s$ predstavimo kao $s = s_1 s_2$ gde $s_1, s_2$ imaju približno jednake dužine, a da zatim nađemo predstavljanje stringa $p = p_1 p_2$ takvo da je $|LCS(s_1, p_1)| + |LCS(s_2, p_2)| = |LCS(s, p)|$, odnosno, ako posmatramo LCS za $s, p$ slova iz $s_1$ su uparena tačno sa slovima iz $p_1$ i slova iz $s_2$ su uparena tačno sa slovima iz $p_2$. Znajući particije ovih stringova, rekurzivno nalazimo $LCS(s_1, p_1)$ i $LCS(s_2, p_2)$ a zatim konkateniramo rezultate.

Opišimo prvo pomoćni algoritam koji za stringove $s, p$ dužina $n, m$ nalazi, za svako $j \in \{0, 1, \ldots, m\}$ vrednost $|LCS(s, p_{[0,j)})|$. Primetimo da je ovaj algoritam identičan algoritmu koji nalazi dužinu LCS-a u $O(m)$ memorije, osim što vraća ceo vektor a ne samo njegov poslednji element.

Pomoćna funkcija Hirschberg-ovog algoritma

```
vector<int> lcs_vector(const string& s, const string& p) {
  int n = s.size(), m = p.size();
  vector<int> di(m+1, 0), dim1(m+1);
  for (int i=1; i<=n; i++) {
    swap(di, dim1);
    for (int j=1; j<=m; j++)
      if (s[i-1] != p[j-1])
        di[j] = max(dim1[j], di[j-1]);
      else
        di[j] = max({dim1[j], di[j-1], dim1[j-1]+1});
  }
  return di;
}
```

Nađimo ovaj vektor za parove stringova $s_1, p$ i $\overline{s_2}, \overline{p}$, neka su to vektori $v_1, v_2$. Ako je $i \in \{0, 1, \ldots, m\}$, tada je $v_1(i) + v_2(m - i)$ dužina LCS-a koji odgovara predstavljanju $p = p_1 p_2$ sa $p_1 = p_{[0,i)}, p_2 = p_{[i,m)}$, pa maksimiziranjem prethodnog izraza po $i$ nalazimo traženu particiju za $p$.

Implementacija Hirschberg-ovog algoritma

```
1  string hirschberg(const string& s, const string& p) {
2    if (s.size() > p.size())
3      return hirschberg(p, s);
4    int n = s.size(), m = p.size();
5    if (n == 0)
6      return string();
7    if (n == 1) {
8      if (p.find(s[0]) != string::npos)
9        return string(1, s[0]);
10     else
11       return string();
12   }
13   string s1 = s.substr(0, n/2), s2 = s.substr(n/2);
14   string p_rev = p, s2_rev = s2;
15   reverse(p_rev.begin(), p_rev.end());
16   reverse(s2_rev.begin(), s2_rev.end());
17   vector<int> v1 = lcs_vector(s1, p);
18   vector<int> v2 = lcs_vector(s2_rev, p_rev);
19   int i_best = 0;
20   for (int i=1; i<=m; i++)
21     if (v1[i] + v2[m-i] > v1[i_best] + v2[m-i_best])
22       i_best = i;
23   return hirschberg(s1, p.substr(0, i_best))
24        + hirschberg(s2, p.substr(i_best));
25 }
```

Uzećemo da je $s$ duži string. Ukoliko nije, rekurzivno zovemo istu funkciju gde parametri menjaju mesto. Koristimo matematičku indukciju da dokažemo da je memorijska složenost algoritma $O(n+m)$. Indukciju radimo po zbiru $n+m$. Dokazaćemo da postoji realan broj $c_2$ takav da algoritam koristi ne više od $c_2(n+m)$ bajtova memorije. Za $n+m \leq 1$ tvrđenje očigledno važi jer nema rekurzivnih poziva. U suprotnom, telo funkcije koristi ne više od $c_1(n+m)$ bajtova dodatne memorije, dok je kod rekurzivnih poziva zbir dužina stringova ne više od $\frac{n}{2}+m \leq \frac{3}{4}(n+m)$ (ovo važi jer je $m \leq n$), odnosno, ako uzmemo da je $c_2 = 4c_1$, važi da algoritam za dužine $n, m$ koristi ne više od $c_2 \cdot \frac{3}{4}(n+m) + c_1(n+m) = 4c_1(n+m) = c_2(n+m)$ memorije, čime završavamo indukcijski korak.

Procenimo sada vremensku složenost algoritma. Označimo sa $H = nm$. Algoritam za računanje nizova $v_1, v_2$ koristi $O(H)$ vremena, dok rekurzivni pozivi zajedno imaju veličinu $\frac{n}{2} \cdot m = \frac{H}{2}$, pa vremenska složenost izražena preko $H$ zadovoljava relaciju $T(H) = O(H) + T(\frac{H}{2})$, pa je na osnovu Master

teoreme vremenska složenost upravo $O(H)$ odnosno $O(nm)$.

## 3.3 Najduži palindromski podniz

**Definicija 3.3** *Za dati string $s$ dužine $n$, najduži palindromski podniz je palindrom najveće dužine koji se javlja kao podniz stringa $s$.*

Problem nalaženja najdužeg palindromskog podniza se efikasno može rešiti pomoću dinamičkog programiranja. Nađimo dužinu najdužeg palindromskog podniza za svaki podstring stringa $s$, preciznije, neka je $d_{l,r}$ za $0 \leq l < r \leq n$ dužina najdužeg palindromskog podniza za string $s_{[l,r)}$. Posmatrajmo sledeće slučajeve:

- $r - l = 1$. Tada string $s_{[l,r)}$ sadrži jedno slovo pa je $d_{l,r} = 1$.
- $r - l = 2$. Tada string $s_{[l,r)}$ sadrži dva slova, ukoliko su ona jednaka $d_{l,r} = 2$, inače je $d_{l,r} = 1$.
- $r - l > 2, s_l \neq s_{r-1}$. Svaki palindromski podniz stringa $s_{[l,r)}$ je sigurno sadržan u celosti ili u $s_{[l,r-1)}$ ili u $s_{[l+1,r)}$, pa je $d_{l,r} = \max(d_{l+1,r}, d_{l,r-1})$.
- $r - l > 2, s_l = s_{r-1}$. Svaki palindromski podniz stringa $s_{[l,r)}$ je sigurno sadržan u celosti ili u $s_{[l,r-1)}$ ili u $s_{[l+1,r)}$ ili počinje na poziciji $l$, završava se na poziciji $r - 1$, a ostatak, koji je takođe palindrom, je u celosti sadržan u $s_{[l+1,r-1)}$ pa je $d_{l,r} = \max(d_{l+1,r}, d_{l,r-1}, d_{l+1,r-1} + 2)$.

Rešenje glavnog problema, odnosno dužina najdužeg palindromskog podniza je po definiciji $d_{0,n}$. Sâmo rešenje se može rekonstruisati sličnim postupkom kao kod nalaženja LCS-a.

Implementacija algoritma za nalaženje najdužeg palindromskog podniza

```
string lpalsubseq(const string& s) {
  int n = s.size();
  vector<vector<int>> d(n, vector<int>(n+1));
  for (int l=n-1; l>=0; l--) {
    d[l][l+1] = 1;
    if (l+2 <= n)
      d[l][l+2] = 1 + (s[l] == s[l+1]);
    for (int r=l+3; r<=n; r++) {
      d[l][r] = max(d[l+1][r], d[l][r-1]);
      if (s[l] == s[r-1])
        d[l][r] = max(d[l][r], d[l+1][r-1]+2);
    }
  }
  int l = 0, r = n;
  string prefix, middle;
  while (r-l > 2) {
    if (s[l] == s[r-1] && d[l+1][r-1]+2 == d[l][r])
      prefix += s[l], l++, r--;
    else if (d[l+1][r] == d[l][r])
      l++;
    else
      r--;
  }
  if (r-l == 2 && s[l] == s[l+1])
    middle = s.substr(l, 2);
  else
    middle = string(1, s[l]);
  string suffix = prefix;
  reverse(suffix.begin(), suffix.end());
  return prefix + middle + suffix;
}
```

Vremenska i memorijska složenost algoritma je $O(n^2)$. Ukoliko je potrebna samo dužina, dovoljno je vratiti vrednost $d_{0,n}$ nakon kraja prve spoljne *for* petlje.

## 3.4 Levenshtein udaljenost

# 4 Prefiksne strukture podataka

## 4.1 Trie

## 4.2 Aho-Corasick algoritam

# 5 Sufiksne strukture podataka

## 5.1 Sufiks niz

## 5.2 Sufiks stablo

## 5.3 Sufiks automat

# 6 Heširanje

## 6.1 Rabin-Karp algoritam

## 6.2 Primene

## 6.3 Konstrukcija anti-heš primera

# 7 Palindromi

## 7.1 Manacher-ov algoritam

## 7.2 Palindromsko stablo

# Literatura

[1] Knuth D.E, Morris J.H, Pratt V.R. *Fast Pattern Matching in Strings*. SIAM Journal on Computing, 1977, Vol. 6, No. 2 : pp. 323-350

[2] Gusfield D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997

[3] Hirschberg D.S. *A linear space algorithm for computing maximal common subsequences*. Comm. A.C.M. 18(6) p341-343, 1975