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 Σ.

Definicija 1.1 *Ako je Σ alfabet a n prirodan broj, Σ^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 Σ alfabet, onda je*

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

skup svih nepraznih reči nad alfabetom Σ.

Ovom skupu možemo dodati i praznu reč, koju označavamo sa e ili ϵ.

Definicija 1.3 *Skup svih reči nad alfabetom Σ 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 Σ^+, odnosno Σ^* 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 Σ totalno uređen, definišemo leksikografsko poređenje stringova kao uređenje skupa Σ^*, 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 Σ^*.* □

Definicija 1.11 *Za svako $x \in \Sigma$, $Ord(x)$ je broj elemenata skupa Σ 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 Θ-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. Verzija standarda jezika biće C++17. 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 broja 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 $2i - r$ 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

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

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

```
1  vector<int> z_algorithm(const string& s, const string& p) {
2    int n = s.size(), m = p.size();
3    string q = p + s;
4    vector<int> z(n+m, 0), result;
5    for (int i=1, l=0, r=0; i<n+m; i++) {
6      if (i < r)
7        z[i] = min(z[i-l], r-i);
8      while (i+z[i] < n+m && q[i+z[i]] == q[z[i]])
9        z[i]++;
10     if (i+z[i] > r)
11       l = i, r = i+z[i];
12     if (z[i] >= m)
13       result.push_back(i-m);
14   }
15   return result;
16 }
```

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

```
string hirschberg(const string& s, const string& p) {
  if (s.size() > p.size())
    return hirschberg(p, s);
  int n = s.size(), m = p.size();
  if (n == 0)
    return string();
  if (n == 1) {
    if (p.find(s[0]) != string::npos)
      return string(1, s[0]);
    else
      return string();
  }
  string s1 = s.substr(0, n/2), s2 = s.substr(n/2);
  string p_rev = p, s2_rev = s2;
  reverse(p_rev.begin(), p_rev.end());
  reverse(s2_rev.begin(), s2_rev.end());
  vector<int> v1 = lcs_vector(s1, p);
  vector<int> v2 = lcs_vector(s2_rev, p_rev);
  int i_best = 0;
  for (int i=1; i<=m; i++)
    if (v1[i] + v2[m-i] > v1[i_best] + v2[m-i_best])
      i_best = i;
  return hirschberg(s1, p.substr(0, i_best))
       + hirschberg(s2, p.substr(i_best));
}
```

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

Definišemo pojam udaljenosti između stringova na drugačiji način, tačnije, dodavanjem nove operacije – izmene slova.

Definicija 3.4 *Levenshtein udaljenost između stringova s, p je minimalan broj operacija potreban da se string s prevede u string p, gde su operacije brisanje slova, umetanje slova i menjanje jednog slova u drugo.*

Ovaj problem rešavamo algoritmom dinamičkog programiranja koji je veoma sličan prethodno opisanim algoritmu za nalaženje LCS-a. Kako je skup operacija na stringovima invertibilan, odnosno, za svaku operaciju postoji inverzna operacija iste težine, svejedno je da li operaciju radimo na stringu s ili na stringu p. Ovo znači da je dovoljno da nađemo string q takav da se minimizuje ukupan broj operacija da se oba stringa dovedu do stringa q. Takođe primetimo da u tom slučaju možemo u potpunosti da zanemarimo operaciju umetanja slova, jer svako umetnuto slovo u, na primer, string s, odgovara nekom slovu u stringu q. Ako je to slovo bilo umetnuto i u p, onda je ono suvišno i može se eliminisati, pri čemu se smanjuje broj operacija. U suprotnom, ono je ili nastalo izmenom nekog slova u p ili od nekog slova koje je na početku bilo u p. U oba slučaja možemo jednostavno obrisati to slovo u p i ne dodavati ga u s, pri čemu se ne povećava broj operacija. Posmatrajmo dva stringa s, p dužina n, m. Tražimo string q takav da se minimizuje ukupan broj poteza da se i s i p prevedu u q, gde su operacije brisanje slova i izmena slova. Neka je $d_{i,j}$ minimalan broj operacija potreban da se stringovi $s_{[0,i)}$ i $p_{[0,j)}$ dovedu do istog stringa, odnosno, to je njihova Levenshtein udaljenost. Nakon svih operacija njima moraju da se poklapaju poslednja dva slova. Ukoliko se u početku njima poklapaju poslednja dva slova, možemo samo da rešimo problem za $s_{[0,i-1)}$ i $p_{[0,j-1)}$. U suprotnom, možemo da izjednačimo ta dva slova u jednoj operaciji, pri čemu ponovo problem svodimo na stringove $s_{[0,i-1)}$ i $p_{[0,j-1)}$ ili možemo jedno od tih slova obrisati, pri čemu svodimo problem na $s_{[0,i-1)}$ i $p_{[0,j)}$ ako brišemo iz stringa s, ili na $s_{[0,i)}$ i $p_{[0,j-1)}$ ako brišemo iz stringa p. Jasno je da, ako je $i = 0$ ili $j = 0$, onda je Levenshtein udaljemost j odnosno i. Dolazimo do sledeće rekurentne veze:

$$d_{i,j} = \begin{cases} i + j & i = 0 \vee j = 0 \\ \min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + \delta(s_{i-1} \neq p_{j-1})) & i, j > 0 \end{cases} \tag{3.4.1}$$

Implementacija algoritma za nalaženje Levenshtein udaljenosti

```
1  int levenshtein(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] = i+j;
8        else
9          d[i][j] = min({d[i-1][j]+1, d[i][j-1]+1,
10           d[i-1][j-1] + (s[i-1] != p[j-1])});
11   return d[n][m];
12 }
```

Kao i kod algoritma za nalaženje LCS-a, mogu se primeniti ideje za smanjenje memorijske složenosti, uključujući i Hirschberg-ov algoritam. U svakom slučaju, vremenska složenost je $O(nm)$.

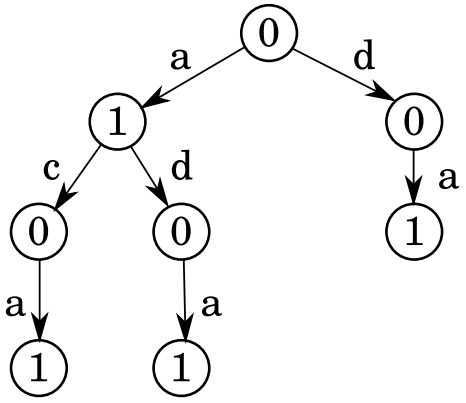
4 Prefiksne strukture podataka

4.1 Prefiksno stablo

Prefiksno stablo, odnosno *trie* je struktura podataka za čuvanje i pretragu kolekcije stringova, i ima oblik n-arnog korenskog stabla, odnosno stabla gde svaki čvor ima stepen najviše $|\Sigma|$. Kod prefiksnog stabla grane imaju labele koje odgovaraju nekom slovu alfabeta Σ, dok čvorovi čuvaju nenegativan ceo broj c_i koji ćemo zvati *višestrukost*.

Definicija 4.1 *Prefiksno stablo za kolekciju stringova $S, s \in \Sigma^*$ je korensko stablo sa minimalnim brojem čvorova, kod kojeg za svaki string $s \in S$ postoji put od korena do čvora koji označavamo sa $\delta(s)$ tako da labele obiđenih grana obrazuju string s, a u čvoru $\delta(s)$ je višestrukost jednaka broju pojavljivanja stringa s u kolekciji S, i kod kojeg izlazne grane svakog čvora imaju međusobno različite labele.*

Na osnovu definicije sledi da različiti putevi u stablu koji počinju od istog čvora odgovaraju različitim stringovima. Posmatrajmo skup svih prefiksa svih stringova u S, neka je to skup P. Kako postojanje puta od korena sa labelom $s \in S$ implicira postojanje takvog puta za sve prefikse stringa s, važi da će za svaki string $p \in P$ postojati jedinstven čvor u stablu. Pokazuje se da je moguće konstruisati jedinstveno stablo sa tačno $|P|$ čvorova koje zadovoljava definiciju 4.1.



Prefiksno stablo za kolekciju $\{aca, ada, a, da\}$.

4.1.1 Konstrukcija

Kao što je prethodno opisano, svaki čvor u sebi čuva višestrukost, i takođe će čuvati izlazne grane. Grane će biti čuvane kao kolekcija parova (k, v), gde će k biti slovo, a v pokazivač na čvor do kojeg se dolazi tom granom. Ovo je najjednostavnije uraditi pomoću rečnika, što nam omogućа va brzo umetanje nove grane a takođe i brzu pretragu grane sa određenim slovom.

Struktura čvora prefiksnog stabla

```
1 struct trie_node {
2   map<char, trie_node*> next;
3   int c;
4   trie_node() : c(0) {}
5 };
```

Algoritam za konstrukciju prefiksnog stabla dodaje redom jedan po jedan string iz kolekcije. Pri dodavanju jednog stringa, krećemo se po postojećim granama dokle god je to moguće, a ukoliko nije kreiramo novi čvor. Na kraju se u čvoru u kom se završi kretanje dodaje 1 na vrednost c.

Algoritam za dodavanje jednog stringa u prefiksno stablo

```
1 void trie_insert(trie_node* root, const string& s) {
2   trie_node* t = root;
3   for (char x : s)
4     if (t->next.count(x))
5       t = t->next[x];
6     else
7       t = t->next[x] = new trie_node;
8   t->c++;
9 }
```

Ukoliko pretpostavimo da operacije na mapi imaju konstantnu složenost, što je opravdano konstantnom veličinom alfabeta, dolazimo do toga da je vremenska složenost algoritma linearna po veličini stringa koji se dodaje.

4.1.2 Primene

Prefiksno stablo ima veliki broj primena. Svaka od primena može zahtevati drugačiji izgled strukture jednog čvora, ali svima je zajedničko to da čvor

čuva svoje izlazne grane i skelet algoritma za dodavanje stringa je isti kao što je prikazano u kodu gore.

Najosnovnija primena prefiksnog stabla je za implementaciju rečnika, odnosno, ono nam omogućava da brzo proverimo da li se neki string javlja u kolekciji ili ne.

Algoritam za brojanje pojavljivanja stringa u prefiksnom stablu

```
1 int trie_count(trie_node* root, const string& s) {
2   trie_node* t = root;
3   for (char x : s)
4     if (t->next.count(x))
5       t = t->next[x];
6     else
7       return 0;
8   return t->c;
9 }
```

Ukoliko pri dodavanju jednog stringa svim usput obiđenim čvorovima inkrementiramo višestrukost c, prethodni algoritam će vratiti broj stringova u kolekciji koji imaju s kao svoj prefiks, odnosno, moguće je dodati sve prefikse jednog stringa odjednom, bez povećanja vremenske ili memorijske složenosti.

Moguće je i pronaći leksikografski najmanji string u kolekciji koji je veći ili jednak zadatom, odnosno, naći *lower bound* za dati string. Za to nam je potrebna pomoćna funkcija koja za dati čvor nalazi leksikografski najmanji put od tog čvora do nekog čvora sa pozitivnom višestrukošću.

Algoritam za nalaženje najmanjeg stringa počev od zadatog čvora

```
1 string trie_smallest(trie_node* t) {
2   string s;
3   while (!t->c) {
4     s += t->next.begin()->first;
5     t = t->next.begin()->second;
6   }
7   return s;
8 }
```

Primetimo da $c = 0$ implicira da čvor ima bar jednu izlaznu granu.

Algoritam za nalaženje lower bound-a za zadati string

```
string trie_lb(trie_node* root, const string& s) {
  int n = s.size();
  trie_node* t = root;
  vector<trie_node*> path = {t};
  for (int i=0; i<n; i++) {
    char x = s[i];
    if (t->next.count(x)) {
      t = t->next[x];
      path.push_back(t);
    } else {
      for (int j=i; j>=0; j--) {
        if (auto it = path[j]->next.upper_bound(s[j]);
          it != path[j]->next.end())
        {
          return s.substr(0, j) + it->first +
            trie_smallest(it->second);
        }
      }
      return "";
    }
  }
  return s + trie_smallest(t);
}
```

Algoritam je grabljive prirode. Posmatrajmo najduži zajednički prefiks stringa s za koji tražimo *lower bound* i nekog stringa iz kolekcije. Ukoliko je taj prefiks jednak celom stringu s, potrebno je da nađemo najmanji string q dostižan iz čvora $\delta(s)$, i da vratimo sq. U suprotnom, ako je dužina tog zajedničkog prefiksa i, biramo najveću poziciju $j \leq i$ takvu da u čvoru $\delta(s_{[0,j)})$ postoji izlazna grana čija je labela slovo strogo veće od s_j, zatim vraćamo string $s_{[0,j)}yq$, gde je y to slovo a q najmanji string dostižan iz čvora $\delta(s_{[0,j)}y)$. Ukoliko takav indeks ne postoji, ne postoji ni leksikografski veći ili jednak string, a algoritam vraća prazan string. Ukupna memorijska i vremenska složenost celog algoritma je $O(n+m)$. gde je $n = |s|$, a m je dužina rezulutujućeg stringa.

4.2 Aho-Corasick algoritam

Aho-Corasick algoritam omogućava pretragu više stringova odjednom, odnosno, za dati skup nepraznih stringova P i string s, on pronalazi sva pojavljivanja

svih stringova iz P u s. Kako je ukupan broj pojavljivanja svih stringova u najgorem slučaju srazmeran $|s| \cdot |P|$, tako je i ovo donja granica na složenost bilo kog algoritma koji pronalazi sva pojavljivanja.

Aho-Corasick algoritam generalizuje KMP algoritam. Osnova za ovaj algoritam je upravo prefiksno stablo i prvi korak jeste njegova konstrukcija, uz malu modifikaciju, koja se odnosi na postojanje dodatnih polja, koje se zovu *sufiks veza* i *rečnička veza*. Takođe, umesto višestrukosti, za čvor $\delta(p)$ pamtimo id, odnosno redni broj stringa $p \in P$, ukoliko se ne radi o stringu iz P već o nekom prefiksu, upisujemo $id = -1$.

Struktura čvora kod Aho-Corasick algoritma

```
1  struct aho_node {
2    map<char, aho_node*> next;
3    int id;
4    aho_node* link;
5    aho_node* dict;
6    aho_node() : id(-1), link(nullptr), dict(nullptr) {}
7  };
```

Dodavanje stringa u prefiksno stablo kod Aho-Corasick algoritma

```
1  void aho_insert(aho_node* root, const string& s, int id) {
2    aho_node* t = root;
3    for (char x : s)
4      if (t->next.count(x))
5        t = t->next[x];
6      else
7        t = t->next[x] = new aho_node;
8    t->id = id;
9  }
```

Definicija 4.2 *Za prefiksno stablo izgrađeno od skupa stringova P, sufiks veza čvora $\delta(s)$ pokazuje na čvor $\delta(q)$ takav da je q najduži string koji je prefiks nekog stringa u P, a koji je ujedno pravi sufiks stringa s. Za koren se sufiks veza ne definiše.*

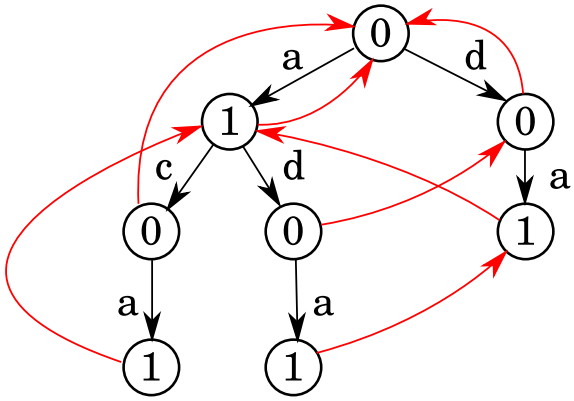
Primetimo da za jednoelementni skup P ove sufiks veze tačno odgovaraju nizu neuspeha kod KMP algoritma. Sufiks veza nekog čvora uvek pokazuje na čvor strogo manje dubine. Zbog ovoga, čvorove je neophodno obraditi u redosledu sortiranom po dubini, za šta se koristi pretraga u širinu. Prirodno,

algoritam za nalaženje sufiks veza podseća na prvu fazu KMP algoritma. Za svaki čvor p, krećemo od sufiks veze njegovog roditelja t, i tražimo čvor l koji ima izlaznu granu sa istom labelom kao grana koja spaja t, p. Pretragu vršimo praćenjem sufiks veza. Ukoliko nađemo takav čvor, sufiks veza čvora p će pokazivati na odgovarajuće dete čvora l, u suprotnom, sufiks veza pokazuje na koren.

Nalaženje sufiks veza kod Aho-Corasick algoritma

```
1  vector<aho_node*> aho_find_links(aho_node* root) {
2    vector<aho_node*> q = {root};
3    size_t qs = 0;
4    while (qs != q.size()) {
5      aho_node* t = q[qs++];
6      for (auto [c, p] : t->next) {
7        aho_node* l = t->link;
8        while (l && !l->next.count(c))
9          l = l->link;
10       if (l)
11         p->link = l->next[c];
12       else
13         p->link = root;
14       q.push_back(p);
15     }
16   }
17   return q;
18 }
```

Funkcija vraća niz svih čvorova stabla, sortiran po dubini. Ovaj niz će biti koristan u kasnijoj obradi. Neka je L ukupna dužina svih stringova u P. Vremenska složenost nalaženja svih sufiks veza je $O(L)$.[5] Dokaz je sličan dokazu složenosti kod KMP algoritma.



Sufiks veze za prefiksno stablo za kolekciju $\{aca, ada, a, da\}$.

Prefiksno stablo podseća na deterministički konačni automat, jer ima jedan polazni čvor i svaki čvor ima izlazne grane označene simbolima iz alfabeta Σ. Međutim, neki čvorovi, npr. listovi stabla nemaju izlazne grane za svaki simbol $x \in \Sigma$. Pomoću izračunatih sufiks veza možemo dopuniti stablo do pravog konačnog automata. Neka je u proizvoljan čvor stabla, a $x \in \Sigma$ proizvoljan simbol. Počev od čvora u, uključujući i u, krećemo se po sufiks vezama i tražimo prvi čvor koji ima izlaznu granu sa labelom x. Ukoliko takav čvor ne postoji, grana automata sa labelom x iz u ide ka korenu stabla. U suprotnom, neka je to čvor v. Tada ta grana ide ka čvoru na koji pokazuje grana sa labelom x iz čvora v.

Smisao ovakve definicije automata je sledeći. Ukoliko se kroz automat propusti string s, ako se automat nalazi u stanju $\delta(p)$, tada je p najduži prefiks nekog stringa iz P koji je ujedno sufiks stringa s. Tada je ovakva definicija automata valjana, odnosno, ovako definisan automat zaista raspoznaje sve različite prefikse stringova iz P. Zbog ovakve definicije, nije nužno eksplicitno pamtiti sve prelaze automata – dovoljno je pri svakom prelazu potražiti pomoću sufiks veza odgovarajući čvor. Slično kao kod KMP-a, ukoliko se kroz ovaj automat propusti string s, ukupna vremenska složenost za kretanje kroz automat će biti $O(|s|)$.[5]

Pošto svaka sufiks veza ide od čvora veće ka čvoru manje dubine, graf koji obrazuju ove veze je acikličan, a pošto svaki čvor osim tačno jednog ima izlaznu granu, radi se o obrnutom korenskom stablu. Nadalje ćemo ovo stablo zvati *stablo sufiks veza*. I originalni Aho-Corasick algoritam i njegove modifikacije se oslanjaju na preprocesiranje ovog stabla.

Originalni Aho-Corasick algoritam nalazi, za skup stringova P i svaki prefiks $s_{[0,i)}$ stringa s sve stringove $p \in P$ koji su sufiksi stringa $s_{[0,i)}$, u vremenskoj složenosti $O(L+|s|+k)$, gde je k broj poklapanja. Posmatrajmo šta se dešava nakon tačno i prelaza, odnosno, kada smo stigli do prefiksa $s_{[0,i)}$. Neka se automat nalazi u stanju $\delta(q)$. Posmatrajmo put od čvora q do korena kroz stablo sufiks veza. Svaki čvor na ovom putu koji ima pozitivnu višestrukost odgovara pojavljivanju nekog stringa $p \in P$ kao sufiks stringa $s_{[0,i)}$. Dakle, naš cilj treba da bude da efikasno otkrijemo sve ove čvorove, poželjno u linearnoj složenosti po njihovom broju. Zato uvodimo novi tip veze – rečničku vezu.

Definicija 4.3 *Rečnička veza za čvor u stabla sufiks veza je prvi čvor dostižan iz u kod kojeg je višestrukost pozitivna. Ukoliko takav čvor ne postoji, rečnička veza se ne definiše.*

Rečničke veze nam omogućavaju da preskočimo sve usputne čvorove koji ne odgovaraju celim stringovima iz P, odnosno, da direktno obiđemo sve čvorove sa pozitivnom višestrukošću. Ove veze se jednostavno konstruišu ukoliko je već izračunat niz čvorova sortiran po dubini.

Nalaženje rečničkih veza kod Aho-Corasick algoritma

```
1 void aho_find_dict(const vector<aho_node*>& q) {
2   int L = q.size();
3   q[0]->dict = nullptr;
4   for (int i=1; i<L; i++)
5     if (q[i]->id != -1)
6       q[i]->dict = q[i];
7     else
8       q[i]->dict = q[i]->link->dict;
9 }
```

Niz q je prethodno izračunati niz čvorova, jasno je da je q_0 koren i da je on jedini čvor koji nema sufiks vezu. Za sve ostale čvorove važi da, ako oni imaju pozitivnu višestrukost, onda rečnička veza pokazuje upravo na njih same, a inače će pokazivati na isti čvor na koji pokazuje njihova sufiks veza.

Konačno, opišimo rad celog algoritma. Prvo, konstruišemo prefiksno stablo ubacivanjem svih stringova iz P. Zatim nalazimo sufiks veze i rečničke veze. Zatim, obrađujemo string s, u svakom trenutku pamtimo pokazivač na trenutnu poziciju u automatu. Nakon svakog dodatog slova, pomoću rečničkih veza obiđemo sva sufiksna poklapanja sa stringovima iz P.

Glavna funkcija Aho-Corasick algoritma

```
vector<pair<int, int>> aho_main(
  const vector<string>& P,
  const string& s
) {
  aho_node* root = new aho_node;
  for (int i=0; i<(int)P.size(); i++)
    aho_insert(root, P[i], i);
  vector<aho_node*> q = aho_find_links(root);
  aho_find_dict(q);
  aho_node* curr = root;
  vector<pair<int, int>> result;
  for (int i=1; i<=(int)s.size(); i++) {
    while (curr && !curr->next.count(s[i-1]))
      curr = curr->link;
    curr = curr ? curr->next[s[i-1]] : root;
    for (auto l = curr->dict; l; l = l->link->dict)
      result.emplace_back(l->id, i-(int)P[l->id].size());
  }
  for (auto l : q)
    delete l;
  return result;
}
```

Na osnovu svega do sad opisanog, složenost algoritma je linearna po zbiru veličina ulaza i izlaza. Jedan od nedostataka ovakvog pristupa je što izlaz može biti dosta veliki, na primer, ako je $P = \{a, a^2, \ldots, a^k\}$ a $s = a^n$, ukupna veličina ulaza je $\Theta(k^2 + n)$ dok je veličina izlaza $\Theta(kn)$, što je za, na primer $k = \Theta(\sqrt{n})$ superlinearna funkcija dužine ulaza. Ukoliko je potrebno eksplicitno enumerisati sva pojavljivanja prethodno opisani algoritam je optimalan. U suprotnom, moguće varijacije su da treba da se prijavi broj pojavljivanja ili prvo pojavljivanje svakog stringa $p \in P$ u s. Ovo je moguće postići u linearnom vremenu po veličini ulaza modifikacijom glavnog algoritma. Naime, za svaki čvor prefiksnog stabla zapamtimo sve trenutke i kada smo, nakon obrade prefiksa $s_{[0,i)}$ završili baš u tom čvoru. Broj pojavljivanja nekog stringa $p \in P$ je onda ukupan broj trenutaka kada smo bili u podstablu stabla sufiks veza sa korenom u čvoru $\delta(p)$, dok je prvo pojavljivanje jednako minimumu svih trenutaka pojavljivanja u tom podstablu. Obe ove vrednosti se mogu efikasno izračunati postprocesiranjem stabla nakon obilaska celog stringa s, primenom dinamičkog programiranja.

5 Sufiksne strukture podataka

5.1 Sufiks niz

Sufiks niz je struktura podataka koja omogućava brzo traženje pojavljivanja stringa unutar stringa za koji se konstruiše sufiks niz, tačnije, vremenska složenost pretrage je sublinearna funkcija dužine stringa unutar kojeg se vrši pretraga. Pored ovoga, pomoću sufiks niza se mogu brzo vršiti poređenja podstringova unutar samog stringa.[7]

Sufiks niz za string s je niz sortiranih nepraznih sufiksa tog stringa. Formalno,

Definicija 5.1 *Sufiks niz za string s dužine $|s| = n$ je niz p koji se sastoji od n različitih celih brojeva iz skupa $\{0, \ldots, n-1\}$ takav da je niz sufiksa čije su početne pozicije $p_0, p_1, \ldots, p_{n-1}$ leksikografski rastući niz.*

Za svaki string postoji jedinstven sufiks niz, zato što je leksikografsko uređenje totalno a ne postoje dva jednaka sufiksa. Primera radi, nađimo sufiks niz za string `banana`. Označimo sa u_i string $s_{[i,n)}$. Svi sufiksi ovog stringa su:

u_0 `banana`
u_1 `anana`
u_2 `nana`
u_3 `ana`
u_4 `na`
u_5 `a`

Sortiranjem dobijamo niz sufiksa:

u_5 `a`
u_3 `ana`
u_1 `anana`
u_0 `banana`
u_4 `na`
u_2 `nana`

Sortirani niz sufiksa je $u_5, u_3, u_1, u_0, u_4, u_2$, pa je sufiks niz $p = (5, 3, 1, 0, 4, 2)$.

5.1.1 Konstrukcija

Sufiks niz se može konstruisati prostim sortiranjem svih sufiksa u vremenskoj složenosti $O(n^2 \log n)$, ukoliko je dostupan algoritam za sortiranje opšte namene koji radi u složenosti $O(n \log n)$. Primer takvog algoritam je *mergesort*. Radi uštede memorijskog prostora sortiraćemo samo niz celih brojeva $0, 1, \ldots, n-1$, dok ćemo kao funkciju za poređenje koristiti funkciju koja je "svesna" stringa s i koja za data dva sufiksa određuje koji je manji.

Implementacija klase za poređenje sufiksa stringa

```
1  struct suffix_cmp {
2    const string& s;
3    suffix_cmp(const string& s) : s(s) {}
4    bool operator() (int u, int v) const {
5      if (u == v)
6        return false;
7      return lexicographical_compare(
8        s.begin()+u, s.end(),
9        s.begin()+v, s.end());
10   }
11 };
```

Vremenska složenost poređenja dva sufiksa je $O(n)$, a kako *mergesort* sortira niz sa $O(n \log n)$ poziva funkcije za poređenje, ukupna vremenska složenost je $O(n^2 \log n)$.

Glavni algoritam za nalaženje sufiks niza, složenosti $O(n^2 \log n)$

```
1  vector<int> sarray_slow(const string& s) {
2    int n = s.size();
3    vector<int> p(n);
4    iota(p.begin(), p.end(), 0);
5    sort(p.begin(), p.end(), suffix_cmp(s));
6    return p;
7  }
```

Napomena. Funkcija `iota` puni zadati opseg uzastopnim vrednostima počev od trećeg parametra, u ovom slučaju 0. Koristimo je da niz p inicijalizujemo vrednostima $p_i = i$. Prema standardu jezika, počev od C++11, funkcija `sort` zove funkciju za poređenje $O(n \log n)$ puta.[10] Treći parametar je funkcija ili drugi objekat koji ima implementiran `operator()` koji može

da poredi elemente niza.

Označimo sa k dužinu najdužeg stringa koji se javlja više od jednom u stringu s. Nije teško pokazati da, ako se pažljivo implementira, funkcija poređenja dva sufiksa radi u vremenskoj složenosti $O(k)$, pa se složenost konstrukcije sufiks niza može i bolje proceniti sa $O(kn \log n)$.

5.1.2 Brži algoritmi za konstrukciju

Algoritmi dati u nastavku efikasno rešavaju problem nalaženja sortiranog niza svih cikličnih pomeraja stringa. Prvo ćemo pokazati kako se problem nalaženja sufiks niza svodi na sortiranje svih cikličnih pomeraja stringa. Neka je $s \in \Sigma^+$. Proširimo alfabet Σ dodavanjem novog simbola, koji ćemo označiti sa \$ koji je po uređenju manji od svih simbola iz Σ. Tada je $s' = s\$ \in (\Sigma \cup \{\$\})^+$. Neka je $n = |s|, n' = n + 1 = |s'|$. Posmatrajmo sve ciklične pomeraje stringa s'. Pošto se karakter \$ javlja samo jednom u stringu s', svi ovi stringovi su različiti pa postoji jedinstven leksikografski poredak. Leksikografski najmanji pomeraj biće $n' - 1$, jer je taj pomeraj jedini koji počinje karakterom \$ koji je po definiciji manji od svih ostalih.

Teorema 5.1 *Ako su i, j proizvoljni ciklični pomeraji stringa s' različiti od $n' - 1$, tada važi $s'_{[i,i+n')} < s'_{[j,j+n')}$ akko je $s_{[i,n)} < s_{[j,n)}$.* □

Odavde sledi da ako je niz $q_0, q_1, \ldots, q_n$ sortiran niz cikličnih pomeraja stringa s' tada je niz $p_i = q_{i+1}, i \in \{0, \ldots, n-1\}$ sufiks niz stringa s.

Svođenje konstrukcije sufiks niza na sortiranje cikličnih pomeraja

```
1 template<class T>
2 vector<int> sarray_scs(const string& s, T func) {
3   vector<int> v = func(s + '\0');
4   v.erase(v.begin());
5   return v;
6 }
```

Konačno, opišimo algoritam za sortiranje cikličnih pomeraja stringa s dužine n. Za svako $k \in \{0, 1, \ldots, \lceil \log_2(n) \rceil\}$ nađimo poredak svih cikličnih podstringova dužine 2^k. Ovaj poredak ćemo opisati nizom celih brojeva $C^{(k)}$, gde stringu $s_{[i,i+2^k)}$ pridružujemo broj $C_i^{(k)}$ na takav način da, ako su podstringovima $s_{[i,i+2^k)}$ i $s_{[j,j+2^k)}$ pridruženi isti brojevi, tada su oni jednaki, a ako

je jednom pridružen manji broj, tada je taj podstring leksikografski manji. Za $k = 0$, možemo jednostavno postaviti $C_i^{(0)} = \mathrm{Ord}(s_i)$.

Neka je $k > 0$. Naš cilj je da odredimo niz $C^{(k)}$ u vremenskoj složenosti $O(n \log n)$. Posmatrajmo podstringove $s_{[i,i+2^k)}$ i $s_{[j,j+2^k)}$ i posmatrajmo uređene parove $u_i = (C_i^{(k-1)}, C_{(i+2^{k-1}) \bmod n}^{(k-1)})$ i $u_j = (C_j^{(k-1)}, C_{(j+2^{k-1}) \bmod n}^{(k-1)})$. Tada je poredak ovih podstringova jednak poretku parova u_i, u_j. Ako za svaki ciklični podstring odredimo ovaj uređeni par, zatim sve dobijene parove sortiramo, a zatim ovim parovima dodelimo ordinalne vrednosti, dobićemo upravo niz $C^{(k)}$. Ovo se može jednostavno izvesti bilo kojim algoritmom za sortiranje koji radi u složenosti $O(n \log n)$.

Za kraj, neka je $k = \lceil \log_2(n) \rceil$. Odavde je $2^k \geq n$. U kontekstu konstrukcije sufiks niza, nijedna dva ciklična pomeraja dužine 2^k neće biti jednaka, pa će niz $C^{(k)}$ sadržati različite brojeve. Odavde sortiran niz cikličnih pomeraja možemo naći kao inverznu permutaciju niza $C^{(k)}$. Važno je napomenuti da će poredak podstringova dužine 2^k biti jednak poretku podstringova dužine n, zato što dužina najdužeg zajedničkog prefiksa za bilo koja dva različita podstringa (bilo koje dužine) iznosi $n - 1$, upravo jer postoji karakter \$ koji se javlja samo jednom u stringu.

Kako ovaj algoritam ima $O(\log n)$ faza, a svaka faza radi u složenosti $O(n \log n)$, ukupna vremenska složenost je $O(n \log^2 n)$. Memorijska složenost je $O(n \log n)$, ali se može smanjiti na $O(n)$ pamćenjem samo nizova $C^{(k)}$ prethodne i trenutne faze.

Algoritam za sortiranje cikličnih pomeraja, složenosti $O(n \log^2(n))$

```
vector<int> scs_fast(const string& s) {
  int n = s.size();
  vector<int> c(s.begin(), s.end());
  for (int h=1; h<n; h*=2) {
    vector<array<int, 3>> t(n);
    for (int i=0; i<n; i++)
      t[i] = {c[i], c[(i+h)%n], i};
    sort(t.begin(), t.end());
    vector<int> cnew(n);
    cnew[t[0][2]] = 0;
    int numc = 1;
    for (int i=1; i<n; i++)
      if (t[i][0] == t[i-1][0] && t[i][1] == t[i-1][1])
        cnew[t[i][2]] = numc-1;
      else
        cnew[t[i][2]] = numc++;
    swap(c, cnew);
  }
  vector<pair<int, int>> g(n);
  for (int i=0; i<n; i++)
    g[i] = {c[i], i};
  sort(g.begin(), g.end());
  vector<int> p(n);
  for (int i=0; i<n; i++)
    p[i] = g[i].second;
  return p;
}
```

Prethodno opisani algoritam za nalaženje cikličnih pomeraja je moguće modifikovati tako da radi u složenosti $O(n \log n)$. Naime, ukoliko bismo sortirali parove u_i u linearnom vremenu, dobili bismo upravo tu vremensku složenost. Svaki par se sastoji iz dva broja iz skupa $\{0, 1, \ldots, n-1\}$. Možemo primeniti ideju iz algoritma *radix sort*. Naime, *radix sort* se oslanja na *counting sort*, koji ima jednostavnu implementaciju i može da sortira bilo koji niz od n elemenata čiji su ključevi za poređenje brojevi iz skupa $\{0, 1, \ldots, k-1\}$ u vremenskoj složenosti $O(n+k)$. Pritom, moguće je implementirati *counting sort* kao stabilan algoritam sortiranja, odnosno algoritam koji ekvivalentnim elementima ne menja relativni poredak. Algoritam *radix sort* za uređene parove bi prvo pomoću *counting sort*-a sortirao sve parove po drugom elementu, a zatim po prvim, vodeći računa da se ne naruši prethodno ustanovljen poredak korišćenjem stabilne varijante *counting sort*-a.

Moguće je pojednostaviti prethodni algoritam, odnosno, svesti ga na samo jedno pozivanje *counting sort*-a. Naime, pored nizova $C^{(k)}$ čuvaćemo eksplicitno i permutaciju $p^{(k)}$ koja odgovara poretku podstringova dužine 2^k. Posmatrajmo šta se dešava ukoliko pomoću *counting sort*-a sortiramo vrednosti $(j - 2^{k-1}) \mod n$, gde je ključ $C^{(k-1)}_{(j-2^{k-1}) \mod n}$, uzete redom za svako j iz niza $p^{(k-1)}$. Stabilan *counting sort* će urediti ove indekse $j' = (j - 2^{k-1}) \mod n$ po vrednosti $C^{(k-1)}_{j'}$, dok će, ukoliko više njih ima istu vrednost, očuvati prethodno ustanovljeni poredak. Kako je ovaj prethodni poredak indukovan vrednostima $C^{(k-1)}_{j} = C^{(k-1)}_{(j'+2^{k-1}) \mod n}$, dobijamo upravo leksikografski redosled parova $(C^{(k-1)}_{j'}, C^{(k-1)}_{(j'+2^{k-1}) \mod n})$. Sada na osnovu ovog sortiranog niz računamo nove vrednosti $C^{(k)}$, dok je $p^{(k)}$ dobijen upravo pomenutim sortiranjem.

Konačno rešenje, odnosno sortiran niz cikličnih pomeraja je upravo niz $p^{(k)}$ za $k = \lceil \log_2(n) \rceil$.

Algoritam za sortiranje cikličnih pomeraja, složenosti $O(n \log n)$

```
vector<int> scs_faster(const string& s) {
  int n = s.size(), k = 256, sz = 0;
  vector<int> p(n), c(s.begin(), s.end());
  vector<vector<int>> g(max(n, k));
  for (int i=0; i<n; i++)
    g[c[i]].push_back(i);
  for (auto& gr : g) {
    for (int i : gr)
      p[sz++] = i;
    gr.clear();
  }
  for (int h=1; h<n; h*=2) {
    vector<int> pnew(n), cnew(n);
    for (int j : p) {
      int jp = (j+n-h)%n;
      g[c[jp]].push_back(jp);
    }
    sz = 0;
    for (auto& gr : g) {
      for (int i : gr)
        pnew[sz++] = i;
      gr.clear();
    }
    cnew[pnew[0]] = 0;
    int numc = 1;
    for (int i=1; i<n; i++) {
      int s0 = pnew[i-1], s1 = pnew[i];
      if (c[s1] == c[s0] && c[(s1+h)%n] == c[(s0+h)%n])
        cnew[s1] = numc-1;
      else
        cnew[s1] = numc++;
    }
    swap(c, cnew);
    swap(p, pnew);
  }
  return p;
}
```

5.1.3 LCP niz

Niz najdužih zajedničkih prefiksa (*longest common prefix*) je niz koji se često zajedno koristi sa sufiks nizom. Za string s dužine n, čiji je sufiks niz $p_0, p_1, \ldots, p_{n-1}$, LCP niz se sastoji od nenegativnih celih brojeva $q_0, q_1, \ldots, q_{n-2}$,

gde q_i označava dužinu najdužeg zajedničkog prefiksa stringova $s_{[p_i,n)}$ i $s_{[p_{i+1},n)}$.

Najduži zajedički prefiks zadovoljava jednu veoma važnu osobinu.

Teorema 5.2 *Neka je s_1, s_2, s_n leksikografski sortiran niz stringova. Neka je $q_i = LCP(s_i, s_{i+1})$. Ako je $i < j$, tada je $LCP(s_i, s_j) = \min\{q_i, q_{i+1}, \ldots, q_{j-1}\}$.* □

Kako se LCP niz konstruiše nad sortiranim nizom sufiksa jednog stringa, on se uz odgovarajuću strukturu podataka za nalaženje minimuma u podnizu može koristiti za određivanje najdužeg zajedničkog prefiksa bilo koja dva sufiksa.

Pre nego što opišemo algoritam za konstrukciju LCP niza, dokažimo sledeću teoremu.

Teorema 5.3 *Neka je s string dužine n, čiji je sufiks niz $p_0, \ldots, p_{n-1}$, LCP niz $q_0, \ldots, q_{n-2}$. Definišemo $r = p^{-1}$ odnosno, $p_{r_i} = i$. Neka je $s_{[i,n)}$ njegov sufiks takav da je $i > 0$ i $p_{n-1} \neq i, i-1$. Tada je $q_{r_i} \geq q_{r_{i-1}} - 1$.*

Dokaz: Uvedimo oznake $i' = i - 1$, $j = p_{r_i+1}$, $j' = p_{r_{i'}+1}$. Drugim rečima, sufiks i' je onaj koji prethodi i, odnosno ima dužinu za 1 veću. j je sufiks koji se nalazi odmah posle i u sufiks nizu. Slično, j' je sufiks koji se nalazi odmah posle i' u sufiks nizu. Ukoliko je $q_{r_{i'}} \leq 1$, onda tvrđenje očigledno važi, jer je $q_{r_i} \geq 0$. Neka je $q_{r_{i'}} \geq 2$. Pošto je j' posle i' u sufiks nizu važi $s_{[j',n)} > s_{[i',n)}$ a pošto je $q_{r_{i'}} \geq 1$ važi $s_{i'} = s_{j'}$. Ako odbacimo prvi karakter sufiksa i' i j' ponovo dobijamo sufikse, $i'+1, j'+1$ i važi $s_{[j'+1,n)} > s_{[i'+1,n)}$, odnosno $s_{[j'+1,n)} > s_{[i,n)}$ ili ekvivalentno $r_{j'+1} > r_i$. Dužina najdužeg zajedničkog prefiksa za $i'+1, j'+1$ je $q_{r_{i'}} - 1$. Na osnovu teoreme 5.2 imamo da je $q_{r_{i'}} - 1 = LCP(s_{[i,n)}, s_{[j'+1,n)}) = \min\{q_{r_i}, q_{r_i+1}, \ldots, q_{r_{j'+1}-1}\}$ odnosno $q_{r_{i'}} - 1 \leq q_{r_i}$ □.

Algoritam za konstrukciju LCP niza[8] radi na sledeći način. Kao ulaz se prosleđuju string i izračunati sufiks niz tog stringa. Prvo se računa inverz sufiksnog niza. Zatim se sufiksi obrađuju redom, opadajuće po dužini, odnosno, uzimaju se redom sufiksi čije su početne pozicije $i = 0, 1, \ldots, n-1$ tim redom. U svakom trenutku održavamo promenljivu k koja je manja ili jednaka od trenutne vrednosti q_{r_i} koju tražimo. U početku je $k = 0$. Ukoliko je $r_i = n - 1$, q_{r_i} se i ne definiše i samo postavljamo $k = 0$. U suprotnom, samo povećavamo k za po 1 sve dok se odgovarajući karakteri sufiksa i i $j = p_{r_i+1}$ poklapaju. Kada dođemo do kraja nekog od ovih sufiksa ili se karakteri ne poklope, prekidamo, upisujemo $q_{r_i} := k$ i zatim, na osnovu

teoreme 5.3 smemo da postavimo $k := \max\{k-1, 0\}$.

Algoritam za nalaženje LCP niza

```
1  vector<int> lcp_array(const string& s, const vector<int>& p)
2  {
3    int n = s.size(), k = 0;
4    vector<int> q(n-1), r(n);
5    for (int i=0; i<n; i++)
6      r[p[i]] = i;
7    for (int i=0; i<n; i++) {
8      if (r[i] != n-1) {
9        int j = p[r[i] + 1];
10       while (i+k < n && j+k < n && s[i+k] == s[j+k])
11         k++;
12       q[r[i]] = k;
13       k = max(0, k-1);
14     } else {
15       k = 0;
16     }
17   }
18   return q;
19 }
```

Vremenska složenost ovog algoritma je $O(n)$ za računanje inverza i petlju koja redom obrađuje sufikse, plus linearna po ukupnom broju uvećavanja promenljive k. Kako u tačno jednoj iteraciji (kad je $r_i = n-1$) k smanjujemo direktno na 0, a u svim ostalim iteracijama k smanjujemo za najviše 1, ukupno je smanjujemo za najviše $2n-2$. Kako je početna vrednost 0 a krajnja ne više od $n-1$, to znači da je k povećavamo najviše $3n-3$, odnosno $O(n)$ puta, pa ceo algoritam radi u složenosti $O(n)$.

5.1.4 Primene

Traženje jednog stringa u drugom

Ukoliko je potrebno tražiti veliki broj stringova, recimo njih m unutar jednog istog stringa s dužine n, pritom, za svaki od tih stringova je neophodno odmah naći odgovor pre nego što se počne sa obradom sledećeg, bez preprocesiranja stringa s biće nam potrebno bar $O(nm)$ vremena. Ukoliko su stringovi koji se traže mnogo manje dužine od n, recimo, svaki je dužine d, onda bi algoritam složenosti $O(md \log n)$ radio mnogo brže.

Pomoću sufiks niza realizacija algoritma koji traži string t dužine d u stringu s je krajnje jednostavna. Binarnom pretragom po sufiks nizu tražimo prvu poziciju takvu da je odgovarajući sufiks leksikografski veći ili jednak traženom stringu. Kako se poređenje bilo kog sufiksa i stringa p može izvršiti u vremenu $O(d)$, složenost pretrage je $O(d \log n)$.

Sledeća implementacija nalazi *lower bound* za string t, odnosno, najmanji broj i takav da je $s_{[p_i,n)} \leq t$, ili $i = n$ ukoliko takav broj ne postoji.

Traženje lower bound-a pomoću sufiks niza

```
int sarray_lb(
  const string& s,
  const vector<int>& p,
  const string& t
) {
  int n = s.size(), l = 0, r = n-1, i = n;
  while (l <= r) {
    int mid = (l+r) >> 1, j = p[mid];
    if (lexicographical_compare(
      s.begin()+j, s.end(),
      t.begin(), t.end()))
    {
      l = mid + 1;
    } else {
      i = mid;
      r = mid - 1;
    }
  }
  return i;
}
```

Prethodni algoritam se može iskoristiti za traženje pojavljivanja stringa t u stringu s, tako što za nađeni broj i uporedimo stringove $s_{[p_i,p_i+d)}$ i t, naravno, ukoliko je $p_i + d \leq n$.

```
1  int sarray_find(
2    const string& s,
3    const vector<int>& p,
4    const string& t
5  ) {
6    int i = sarray_lb(s, p, t), n = s.size(), d = t.size();
7    if (i == n)
8      return -1;
9    int j = p[i];
10   if (j+d <= n && equal(t.begin(), t.end(), s.begin()+j))
11     return j;
12   return -1;
13 }
```

Leksikografski najmanji ciklični pomeraj

Pomoću algoritma za sortiranje cikličnih pomeraja možemo lako utvrditi koji ciklični pomeraj je najmanji, tako što direktno primenimo tu funkciju na string bez dodavanja novog, minimalnog karaktera. Štaviše, jednostavno možemo naći i minimalnu periodu stringa s dužine n, odnosno, najmanji prirodan broj l takav da je $s_i = s_{(i+l) \bmod n}$ za svako i. To je upravo broj različitih elemenata niza $C^{(k)}$ za $k = \lceil \log_2(n) \rceil$.

Leksikografsko poređenje podstringova

Ukoliko kod algoritma za sortiranje cikličnih pomeraja upamtimo sve nizove $C^{(k)}$, možemo vrlo jednostavno vršiti leksikografsko poređenje podstringova, uključujući i ciklične podstringove. Ukoliko treba da uporedimo dva stringa različitih dužina, recimo $l_1 < l_2$, prvo uporedimo kraći od njih sa prefiksom dužeg dužine l_1. Ukoliko dobijemo da su ti stringovi različiti, prekidamo, inače, po definiciji, leksikografski manji string je kraći. Zato pretpostavimo da upoređujemo podstringove iste dužine, neka su to $s_{[i,i+l)}, s_{[j,j+l)}$. Ukoliko je $l = 2^k$ za neko k, možemo prosto uporediti vrednosti $C_i^{(k)}$ i $C_j^{(k)}$. U suprotnom, nađimo najveće k takvo da je $2^k < l$. Pritom, jasno je da važi $2^k > \frac{l}{2}$. Ideja je da prvo uporedimo prefikse ovih stringova dužine 2^k isto kao u prethodnom slučaju. Ukoliko dobijemo da su prefiksi jednaki, uporedimo njihove sufikse dužine 2^k – rezultat ovog poređenja biće rezultat poređenja celih stringova.

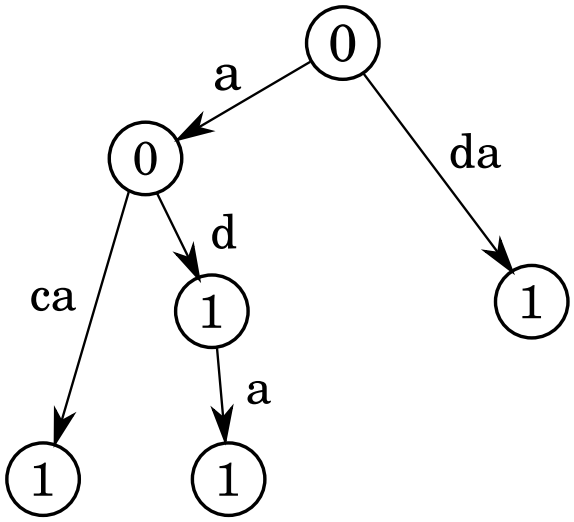
Jasno je da ovaj algoritam radi u složenosti $O(1)$ ali zato koristi $O(n \log n)$

memorije za nizove $C^{(k)}$. Moguće je dostići istu vremensku složenost za poređenje sa samo $O(n)$ dodatne memorije korišćenjem LCP niza. Ideja je da, pri poređenju podstringova $s_{[i,i+l)}, s_{[j,j+l)}$ nađemo $d = LCP(s_{[i,n)}, s_{[j,n)})$, Ukoliko je $d < l$, samo uporedimo karaktere s_{i+d}, s_{j+d}, inače, stringovi su jednaki. Na osnovu teoreme 5.2 problem se svodi na nalaženje minimuma u podsegmentu niza. Ovaj problem se može efikasno rešiti u vremenskoj složenosti $O(1)$ sa $O(n)$ preprocesiranja i $O(n)$ utrošene memorije.[6]

5.2 Sufiks stablo

Sufiks stablo je struktura podataka koja uopštava sufiks niz i usko je povezana sa njim. Da bismo definisali sufiksno stablo, definišimo prvo kompresovano prefiksno stablo.

Definicija 5.2 *Kompresovano prefiksno stablo za skup nepraznih stringova P se dobija tako što se u prefiksnom stablu obrišu svi čvorovi višestrukosti* 0 *sa tačno jednim detetom, osim korena, a zatim se dodaju grane koje odgovaraju obrisanim putevima u stablu i imaju labele koje odgovaraju obrisanim putevima.*



Kompresovano prefiksno stablo za skup $\{aca, ad, ada, da\}$.

U primeru sa slike, čvor $\delta(ad)$ ne brišemo iako on ima jedno dete. Najzad, definišimo sufiks stablo.

Definicija 5.3 *Sufiks stablo za string s je kompresovano prefiksno stablo svih nepraznih sufiksa stringa s.*

Iako na prvi pogled deluje da bi ovakvo stablo zauzimalo previše memorije i iz tog razloga bilo nepraktično, pokazuje se suprotno.

Teorema 5.4 *Kompresovano prefiksno stablo od n stringova sadrži najviše $2n$ čvorova.*

Dokaz. Indukcijom po n. Za $n = 1$ tvrđenje očigledno važi. Posmatrajmo stablo za skup stringova $P' = P - p$. Neka je q najduži prefiks stringa p koji se javlja u P'. Ukoliko ne postoji čvor $\delta(q)$, put od korena sa labelom q će odgovarati unutrašnjosti neke grane, pa ovu granu delimo na dve umetanjem novog čvora koji će onda biti $\delta(q)$. Zatim, ukoliko je $|q| < |p|$ dodajemo novu granu iz čvora $\delta(q)$ ka novom čvoru i upisujemo joj labelu $p_{[|q|,|p|)}$. Broj čvorova se povećao za najviše 2, što dokazuje tvrđenje. □

Naizgled, čak iako stablo ima mali broj čvorova, ukupna dužina labela svih grana može biti $\Theta(n^2)$, što bi impliciralo veliku memorijsku složenost. Kod sufiks stabla, sve labele grana su podstringovi stringa s pa umesto stringa labelu možemo predstaviti kao par l, r koji bi označavao da je labela te grane jednaka stringu $s_{[l,r)}$.

Postoje algoritmi koji konstruišu sufiks stablo direktno iz datog stringa u linearnom vremenu.[9] U nastavku će biti prezentovan jednostavan algoritam koji na osnovu izračunatih sufiks i LCP nizova konstruiše sufiks stablo.

Struktura čvora sufiks stabla

```
1 struct stree_node {
2   int l, r, id;
3   stree_node* p;
4   map<char, stree_node*> next;
5   stree_node(int l, int r, int id, stree_node* p)
6     : l(l), r(r), id(id), p(p) {}
7 };
```

Svaki čvor sufiks stabla pamti labelu grane koja spaja taj čvor i njegovog roditelja, odnosno podstring stringa s, pokazivač na svog roditelja kao i pokazivače na svoju decu. Umesto višestrukosti pamti se redni broj id takav da je taj čvor jednak $\delta(s_{[id,n)})$ ukoliko postoji, inače -1, što odgovara čvorovima višestrukosti 0. Za koren stabla se ne definiše roditelj a brojeve

l, r postavljamo na 0. Iz definicije sufiks stabla zaključujemo da sve grane koji izlaze iz jednog čvora imaju labele kojima se razlikuje prvo slovo, što opravdava izbor mape za čuvanje izlaznih grana. Algoritam radi na sledeći način. U leksikografskom redosledu se dodaju sufiksi stringa s. Ovaj redosled je određen sufiks nizom p. Prvi string se dodaje kao zasebna grana. Za svaki naredni string, odnosno i-ti za $i > 0$, važi da je najduži zajednički prefiks njega i bilo kog drugog stringa upravo jednak q_{i-1} (Teorema 5.2). Penjemo se od kraja prethodno ubačenog sufiksa do dubine q_{i-1} u stablu, mereno po broju slova u labelama. Ukoliko se pozicija na ovoj visini nalazi na sredini grane, delimo tu granu na dva dela. Konačno, ostaje nam da dodamo deo završni deo sufiksa dužine $(n - p_i) - q_{i-1}$, naravno, samo ako je on neprazan.

Implementacija algoritma za konstrukciju sufiks stabla

```
using node = stree_node;
node* suffix_tree(
  const string& s,
  const vector<int>& p,
  const vector<int>& q
) {
  int n = s.size();
  auto root = new node(0, 0, 0, 0);
  auto curr = root->next[s[p[0]]]
    = new node(p[0], n, p[0], root);
  for (int i=1; i<n; i++) {
    int t = q[i-1], d = n - p[i-1];
    while (d && d - t >= (curr->r - curr->l)) {
      d -= curr->r - curr->l;
      curr = curr->p;
    }
    if (d > t) {
      int m = curr->r - (d - t);
      node* mid = new node(curr->l, m, -1, curr->p);
      mid->next[s[m]] = curr;
      curr->p->next[s[curr->l]] = mid;
      curr->l = m;
      curr->p = mid;
      curr = mid;
    }
    if (p[i] + t < n)
      curr = curr->next[s[p[i] + t]]
        = new node(p[i] + t, n, p[i], curr);
  }
  return root;
}
```

Vremenska složenost ovog algoritma je $O(n)$. Ovo se može dokazati tako što posmatramo vrednost $2i - y$, gde je y dubina čvora *curr* u stablu, merena po broju grana. Svaka iteracija i spoljne *for* i unutrašnje *while* petlje uvećava vrednost ovog izraza za bar 1, pa je njihov ukupan broj manji od $2n$.

Sufiks stablo omogućava da se nađu prvo, poslednje, broj pojavljivanja nekog stringa t unutar stringa s u linearnom vremenu procedurom spuštanja slično kao kod prefiksnog stabla. Kod traženja svih pojavljivanja stringa korisna je sledeća teorema:

Teorema 5.5 *Kod sufiksnog stabla ne postoji podstablo koje sadrži više čvorova višestrukosti* 0 *nego čvorova višestrukosti* 1.

Dokaz. Pretpostavimo suprotno, da podstablo ima k čvorova i da čvorova višestrukosti 0 ima više od $\frac{k}{2}$. Na osnovu definicije sufiks stabla, svaki takav čvor ima bar dvoje dece, pa je ukupan broj dece svih čvorova u podstablu veći od k, što je nemoguće, jer je ukupan broj dece svih čvorova tačno $k - 1$.

Ovo nam omogućava da, za neki string t, od pozicije $\delta(t)$, ukoliko postoji, izvršimo pretragu u dubinu ili širinu da bismo našli sve čvorove višestrukosti 1 u podstablu pozicije $\delta(t)$. Ako neki ovakav čvor odgovara sufiksu id, znamo da je $t = s_{[id,id+|t|)}$ odnosno, id je pozicija jednog pojavljivanja stringa t u stringu s. Kako je složenost pretrage linearna po veličini podstabla, i kako je ukupan broj čvorova podstabla ne više od duplo veći od broja čvorova višestrukosti 1, vremenska složenost algoritma je $O(|t| + k)$, gde je k broj pojavljivanja stringa t u s.

Od konstruisanog sufiks stabla se može dobiti sufiks niz traženjem praznog stringa, ili ekvivalentno, puštanjem pretrage u dubinu od korena i pamćenjem id vrednosti za sve čvorove za koje je $id \neq -1$.

Nalaženje svih pojavljivanja stringa t u stringu s

```
using node = stree_node;
vector<int> stree_find_all(
  node* root,
  const string& s,
  const string& t
) {
  vector<int> result;
  node* curr = root;
  int pos = 0;
  for (char x : t)
    if (pos == curr->r) {
      if (!curr->next.count(x))
        return result;
      curr = curr->next[x];
      pos = curr->l + 1;
    } else if (x != s[pos])
      return result;
    else
      pos++;
  vector<node*> q = {curr};
  size_t qs = 0;
  while (qs != q.size()) {
    node* tmp = q[qs++];
    if (tmp->id != -1)
      result.push_back(tmp->id);
    for (auto [x, ch] : tmp->next)
      q.push_back(ch);
  }
  return result;
}
```

Napomena. Prikazani kod ne vraća sve pozicije u rastućem, već u proizvoljnom redosledu.

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

[4] Jurafsky D, Martin J.H. *Speech and Language Processing*. Pearson Education International, 2000

[5] Aho A.V, Corasick M.J. *Efficient string matching: an aid to bibliographic search*. Comm. A.C.M. 18(6) p333-340, 1975

[6] Bender M.A., Farach-Colton M. *The LCA Problem Revisited*. Gonnet G.H., Viola A. (eds) LATIN 2000: Theoretical Informatics. LATIN 2000. Lecture Notes in Computer Science, vol 1776. Springer, Berlin, Heidelberg

[7] Manber U, Myers G. *Suffix arrays: a new method for on-line string searches*. SIAM J Comput. 1993;22(5):935–48.

[8] Kasai T, Lee G, Arimura H, Arikawa S, Park K. *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*. Amir A. (eds) Combinatorial Pattern Matching. CPM 2001. Lecture Notes in Computer Science, vol 2089. Springer, Berlin, Heidelberg

[9] Ukkonen E. *On-line construction of suffix trees*. Algorithmica (1995) 14: 249.

[10] https://en.cppreference.com/w/cpp/algorithm/sort