Algoritmi za rad sa tekstom

KMP - podsećanje

```
int main() {
  string str;
  cin >> str;
  int n;
  cin >> n;
  vector<int> kmp(str.size() + 1);
  kmp[0] = -1;
  for (int i = 0; i < str.size(); i++) {
    int k = i;
    while (k > 0 && str[i] != str[kmp[k]])
      k = kmp[k];
    kmp[i + 1] = kmp[k] + 1;
  }

  cout << kmp[str.size()]  << endl;
  return 0;
}
```

U naivnom algoritmu nakon pomeranja uzorka za jedno mesto udesno zaboravljaju se sve informacije o prethodno poklopljenim karakterima. Stoga je moguće da se jedan isti karakter teksta poredi sa različitim karakterima uzorka, što i vodi složenosti $O(mn)$ u najgorem slučaju, gde je sa m označena dužina uzorka, a sa n dužina teksta. Algoritam koji su osmislili Knut, Moris i Prat se zasniva na ideji da se iskoriste informacije dobijene prethodnim poređenjima karaktera. U ovom algoritmu se nikada iznova ne porede karakteri teksta koji su se prethodno poklopili sa uzorkom. Na ovaj način faza pretrage uzorka u tekstu je složenosti $O(n)$. Da bi ovo bilo izvodljivo, na početku algoritma vrši se preprocesiranje uzorka u cilju analize njegove strukture. Faza preprocesiranja je složenosti $O(m)$, te je ukupna složenosti Knut-Moris-Pratovog algoritma $O(m+n)$.

Neka je $x = x_0 \dots x_{k-1}$ niska dužine k nad azbukom A. Važi sledeće:

- u je prefiks niske x dužine b ako je $u = x_0 \dots x_{b-1}$ gde je $b \in \{0, \dots, k\}$.
- u je sufiks niske x dužine b ako je $u = x_{k-b} \dots x_{k-1}$ gde je $b \in \{0, \dots, k\}$.

Za prefiks (sufiks) u kažemo da je pravi prefiks (pravi sufiks) niske x ako je $u \neq x$, odnosno ako je njegova dužina b manja od k. Za nisku u kažemo da je *prefiks-sufiks* niske x ako je $u = x_0 \dots x_{b-1}$ i istovremeno $u = x_{k-b} \dots x_{k-1}$ za $b \in \{0, \dots, k-1\}$, odnosno ako je istovremeno i pravi prefiks i pravi sufiks niske x.

Primer: Neka je $x = abacab$. Pravi prefiksi niske x su $\epsilon, a, ab, aba, abac, abaca$, a pravi sufiksi $\epsilon, b, ab, cab, acab, bacab$. Dakle, prefiks-sufiksi niske x su ϵ dužine 0 i ab dužine 2.

Prazna niska je uvek prefiks-sufiks niske x, osim ako je niska x prazna – u tom slučaju ona nema prefiks-sufiks.

U narednom primeru ilustrovaćemo kako pojam prefiks-sufiksa može pomoći prilikom izračunavanja vrednosti za koju treba pomeriti uzorak u odnosu na tekst.

Primer:

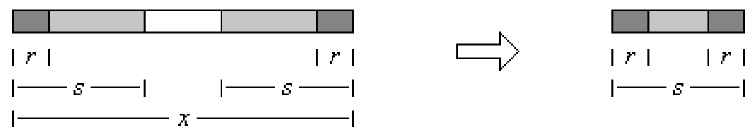
```
        0123456789...
tekst:  abcabcabd
uzorak: abcabd
           abcabd
```

Karakteri na pozicijama od 0 do 4 su se poklopili, dok je poređenje na poziciji 5 dovelo do razlike. Interesuje nas za koliko najmanje treba pomeriti uzorak udesno, tako da se manji deo uzorka i dalje poklapa sa tekstom? S obzirom da se nekoliko zadnjih karaktera poklopljenog prefiksa uzorka trebaju poklopiti sa nekoliko prvih karaktera poklopljenog prefiksa uzorka, nama je u stvari potreban neki prefiks-sufiks poklopljenog prefiksa uzorka. S obzirom da želimo da se pomerimo udesno što je manje moguće, mi u stvari tražimo maksimalni (najduži) prefiks-sufiks. U ovom primeru uzorak se može pomeriti za 3 pozicije i poređenja se mogu nastaviti od pozicije 5. Vrednost za koju se pomeramo u tekstu određena je najdužim prefiks-sufiksom poklopljenog prefiksa uzorka. U ovom primeru, poklopljeni prefiks je $abcab$ i njegova dužina je 5. Njegov najduži prefiks-sufiks je ab i dužine je 2. Dakle, uzorak pomeramo u odnosu na tekst za $5 - 2 = 3$ karaktera.

Dakle, u fazi preprocesiranja potrebno je odrediti dužinu najdužeg prefiks-sufiksa svakog prefiksa datog uzorka, a nakon toga se u fazi pretrage, na osnovu prefiksa uzorka koji se poklopio sa tekstom, utvrđuje za koliko mesta treba pomeriti uzorak u odnosu na tekst.

Teorema: Neka su r i s prefiks-sufiksi niske x, pri čemu važi $|r| < |s|$. Onda je niska r prefiks-sufiks niske s.

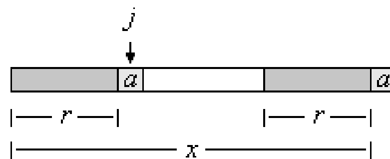
Dokaz: Niska r je prefiks niske x, pa je i pravi prefiks niske s jer je kraća od nje. r je istovremeno sufiks niske x, te je stoga i pravi sufiks niske s. Stoga je r prefiks-sufiks niske s (slika 1).



Slika 1: Prefiks-sufiksi r i s niske x.

Ako je s najduži prefiks-sufiks niske x, sledeći po redu najduži prefiks-sufiks se dobija kao najduži prefiks-sufiks niske s, itd.

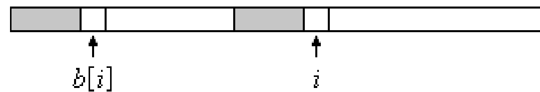
Prefiks-sufiks r niske x može se proširiti karakterom a, ako je ra prefiks-sufiks niske xa (slika 2).



Slika 2: Proširivanje prefiks-sufiksa.

U fazi preprocesiranja izračunavaju se vrednosti niza b dužine $m+1$, pri čemu $b[i]$ sadrži dužinu najdužeg prefiks-sufiksa prefiksa dužine i datog uzorka ($i = 0, \ldots, m$). S obzirom na to da prefiks ϵ dužine 0 nema prefiks-sufikse, vrednost $b[0]$ postavljamo na -1.

Pod pretpostavkom da su vrednosti $b[0], \ldots, b[i]$ već izračunate, vrednost $b[i+1]$ se izračunava proverom da li se prefiks-sufiks prefiksa $p_0 p_1 \ldots p_{i-1}$ može proširiti karakterom p_i. Ovo važi ako je $p_{b[i]} = p_i$ (slika 3). Prefiks-sufiksi se ispituju u opadajućem redosledu dužina iz skupa vrednosti $b[i], b[b[i]], \ldots$



Slika 3: Prefiks dužine i uzorka sa prefiks-sufiksom dužine $b[i]$.

Sam algoritam preprocesiranja sastoji se od petlje u kojoj promenljiva j uzima redom vrednosti $b[i], b[b[i]], \ldots$. Prefiks-sufiks dužine j može se proširiti karakterom p_i ako je $p_j = p_i$. Ako to nije slučaj, naredni najduži prefiks-sufiks se određuje postavljanjem $j = b[j]$. Petlja se najkasnije prekida ako se nijedan prefiks-sufiks ne može proširiti ($j = -1$). Na kraju izvršavanja petlje vrednost promenljive j uvećana za 1 sadržaće dužinu najdužeg prefiks-sufiksa niske $p_o ... p_i$ i nju treba smestiti na poziciju $b[i+1]$.

```
#include<iostream>
#include<vector>
using namespace std;

void kmpPreprocesiraj(const string &p, vector<int> &b)
{
  int m = b.size();
  int i = 0;
  int j = -1;

  // prazna niska nema prefiks-sufikse
  b[i]=j;
  // za prefiks duzine i polazne niske
  // racunamo najduzi prefiks-sufiks
  while (i<m){
    // proveravamo da li se najduzi prefiks-sufiks
    // prefiksa koji se zavrsava na prethodnoj poziciji u niski
    // moze prosiriti: to vazi ako je p[i]=p[j]
    while (j>=0 && p[i]!=p[j])
      j=b[j];
    i++;
```

```
    j++;
    b[i]=j;
  }
}

int main(){

  string uzorak = "ababaa";
  int n = uzorak.size();

  vector<int> b(n+1);
  kmpPreprocesiraj(uzorak,b);
  cout << "Duzine najduzih prefiks-sufiksa date niske su ";
  for (int i=0; i<b.size(); i++){
    cout << b[i] << " ";
  }
  cout << endl;

  return 0;
}
```

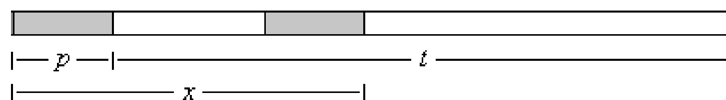
Primer: Za uzorak `p=ababaa` bi recimo vrednost $b[5]$ bila jednaka 3 jer je za prefiks `ababa` dužine 5 najduži prefiks-sufiks jednak *aba* i dužine je 3. U nastavku su prikazane vrednosti dužina najdužih prefiks-sufiksa za sve prefikse datog uzorka:

```
b[0]=-1
prazna niska nema prefiks-sufiks
--------
i=0, j=-1
i=1, j=0
b[1]=0
najduzi prefiks-sufiks prefiksa a je prazna niska
--------
i=1, j=0
p_1!=p_0 => j=-1
i=2, j=0
b[2]=0
najduzi prefiks-sufiks prefiksa ab je prazna niska
--------
i=2, j=0
p_2=p_0
i=3, j=1
b[3]=1
najduzi prefiks-sufiks prefiksa aba je niska a duzine 1
--------
i=3, j=1
```

```
p_3=p_1
i=4, j=2
b[4]=2
najduzi prefiks-sufiks prefiksa abab je niska ab duzine 2
--------
i=4, j=2
p_4=p_2
i=5, j=3
b[5]=3
najduzi prefiks-sufiks prefiksa ababa je niska a duzine 3
--------
i=5, j=3
p_5!=p_3 => j=1
p_5!=p_1 => j=0
p_5=p_0
i=6, j=1
b[6]=1
najduzi prefiks-sufiks prefiksa ababaa je niska a duzine 1
--------

i:     0  1  2  3  4  5  6
b[i]: -1  0  0  1  2  3  1
```

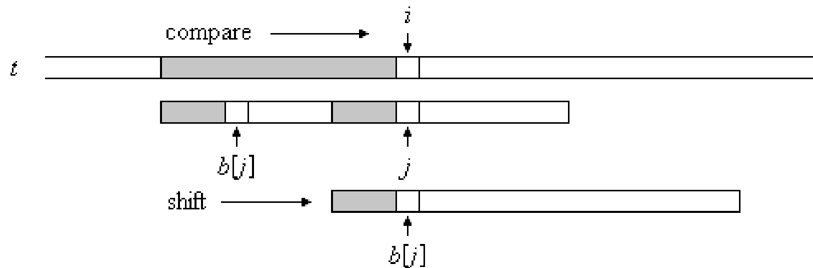
Prikazani algoritam preprocesiranja je mogao biti primenjen umesto na nisku p na nisku pt, gde je sa p označen uzorak, a sa t tekst. Ako se računaju prefiks-sufiksi do dužine m ($|p| = m$), onda prefiks-sufiks dužine m nekog prefiksa x niske pt odgovara pronalaženju uzorka p u tekstu t (slika 4).



Slika 4: Prefiks-sufiks dužine m prefiksa x niske pt.

Algoritam kojim se traži uzorak u tekstu jako je sličan algoritmu preprocesiranja uzorka. Kada se u unutrašnjoj `while` petlji naiđe na nepoklapanje na poziciji j, razmatra se najduži prefiks-sufiks poklapajućeg prefiksa dužine j uzorka (slika 5). Nastavljamo sa poređenjem na poziciji $b[j]$ i ako se naredni karakter ne poklapa, razmatra se naredni prefiks-sufiks po dužini i sve tako dok ili ne dođemo u situaciju da nema prefiksa-sufiksa uzorka ($j = -1$) ili dok se naredni karakter ne poklopi. Nakon toga imamo novi poklapajući prefiks uzorka i nastavljamo sa spoljašnjom `while` petljom.

Ukoliko smo naišli na poklapanje svih m karaktera uzorka (slučaj $j = m$), evidentiramo da smo pronašli uzorak u tekstu počev od pozicije $i - j$. Nakon toga, uzorak se pomera udesno na osnovu svog najdužeg prefiks-sufiksa.



Slika 5: Pomeranje uzorka kada se naiđe na nepoklapanje na poziciji j.

```
#include<iostream>
#include<vector>
using namespace std;

void kmpPreprocesiraj(const string &p, vector<int> & b)
{
  int m = b.size();
  int i = 0;
  int j = -1;

  // prazna niska nema prefiks-sufikse
  b[i]=j;
  // za svaki prefiks polazne niske
  // racunamo najduzi prefiks-sufiks
  while (i<m){
    // proveravamo da li se najduzi prefiks-sufiks
    // prefiksa uzorka koji se zavrsava na prethodnoj poziciji
    // moze prosiriti: to vazi ako je p[i]=p[j]
    while (j>=0 && p[i]!=p[j])
      j=b[j];
    i++;
    j++;
    b[i]=j;
  }
}

void kmpTrazi(const string &t, const string &p, vector<int> b)
{

  int n = t.size();
  int m = p.size();
  int i = 0;
  int j = 0;
```

```
  // za sve prefikse teksta trazimo duzinu
  // najduzeg prefiks-sufiksa
  while (i<n){
    // proveravamo da li se najduzi prefiks-sufiks
    // prefiksa niske t koji se zavrsava na prethodnoj poziciji
    // moze prosiriti: to vazi ako je t[i]=p[j]
    while (j>=0 && t[i]!=p[j])
      j=b[j];
    i++;
    j++;
    if (j==m){
      cout << "Uzorak se nalazi u tekstu pocev od pozicije " << i-j << endl;
      j=b[j];
    }
  }
}

int main(){

  string tekst = "abrakadabra";
  string uzorak = "ra";

  int m = uzorak.size();
  vector<int> b(m+1);

  kmpPreprocesiraj(uzorak,b);

  int n = tekst.size();
  kmpTrazi(tekst,uzorak,b);

  return 0;
}
```

Primer: Razmotrimo koja se poređenja izvršavaju u algoritmu KMP, prilikom traženja uzorka `ababac` u tekstu koji počinje sa `ababbabaa`. Uspešna poređenja karaktera su istaknuta zvezdicom, a poređenja kod kojih se nije naišlo na poklapanje karakterom +.

```
0  1  2  3  4  5  6  7  8  9 ...
a  b  a  b  b  a  b  a  a  ...
a* b* a* b* a+
      a  b  a+
            a+
               a* b* a* b+
                     a  b+ ...
```

Koja je složenost funkcije za preprocesiranje uzorka? Unutrašnja petlja smanjuje vrednost j bar za 1, jer je $b[j] < j$. Petlja se završava najkasnije kada vrednost j postane -1 (a na početku je j jednako -1), te se stoga može smanjiti najviše onoliko puta koliko je prethodno bila povećana naredbom `j++`. S obzirom na to da se naredba `j++` izvršava u spoljašnjoj petlji tačno m puta, ukupan broj izvršavanja unutrašnje `while` petlje je ograničen sa m, te je stoga ukupna složenost algoritma preprocesiranja $O(m)$. Potpuno analogno se može zaključiti da je složenost algoritma traženja uzorka u tekstu $O(n)$, te je ukupna složenost Knut-Moris-Pratovog algoritma $O(m + n)$.

U prethodnom primeru smo imali finu ilustraciju složenosti faze pretrage, s obzirom na to da poređenja (uspešna i neuspešna) formiraju "stepenice" koje u najgorem slučaju mogu biti jednako visoke i duge, te je u najgorem slučaju potrebno $2n$ poređenja prilikom traženja uzorka u tekstu.

Problem: Reč w je periodična ako postoji neprazna reč $p = p_1p_2$ i prirodan broj $n \geq 2$ tako da je $w = p^np_1$. Na primer, reč *abacabacabacab* je periodična jer se ponavlja *abac*, pri čemu se poslednje ponavljanje ne završava celo već se zaustavlja sa *ab*, tj. reč je $(abac)^3ab$. Napisati program koji proverava da li je uneta reč periodična.

Zadatak možemo rešiti grubom silom, tako što ćemo za svaku vrednost d takvu da je $2d \leq |w|$ proveriti da li je reč periodična pri čemu je period prefiks reči w dužine d. Jednostavno se dokazuje da je reč periodična sa periodom p čija je dužina d, ako i samo za svako i za koje je $0 \leq i$ i $i + d < |w|$ važi da je $w_i = w_{i+d}$. Zadatak se onda rešava sa dve ugnežđene linearne pretrage – u spoljnoj proveravamo sve potencijalne vrednosti dužine d, a u unutrašnjoj proveravamo da li postoji vrednost i takva da je $w_i \neq w_{i+d}$. Ako u unutrašnjoj petlji utvrdimo da takvo i ne postoji, tada je niska periodična. Ako pronađemo takvo i, možemo prekinuti unutrašnju petlju (reč nije periodična sa periodom dužine d) i preći na sledeće d (za jedan veće). Ako takvo d ne postoji, tada možemo konstatovati da reč nije periodična.

```
// provera da li je niska periodicna
bool periodicna(const string & str){

  int m = str.size();
  // proveravamo za svaku mogucu duzinu periode
  for (int d = 1; 2 * d <= m; d++) {
    bool greska = false;
    for (int i = 0; i + d < m; i++)
      // ako naidjemo na nepoklapanje, znamo da prefiks duzine d
      // nije perioda niske
      if (str[i] != str[i + d]) {
         greska = true;
         break;
      }
    // ako smo uspesno poklopili sve karaktere niske
```

```
    // pronasli smo periodu
    if (!greska) {
      return true;
    }
  }
  return false;
}

int main() {

  string rec;
  cin >> rec;
  cout << (periodicna(rec) ? "Rec je periodicna"
       : "Rec nije periodicna") << endl;
  return 0;
}
```

Složenost najgoreg slučaja ovog algoritma je kvadratna. Zaista, unutrašnja linearna pretraga može u najgorem slučaju zahtevati $O(|w|)$ iteracija, i ona se ponavlja $O(|w|)$ puta. Ipak, ako je niska nasumična, realno je očekivati da će se za većinu vrednosti d veoma brzo ustanovljavati da je $w_i \neq w_{i+d}$, pa program može raditi dosta brže od najgoreg slučaja.

Efikasnije rešenje se zasniva na narednoj teoremi:

Teorema: Niska w je periodična ako i samo ako ima prefiks-sufiks x čija je dužina najmanje polovina niske, tj. ako postoje neprazni x, s i t takvi da je $w = xs = tx$ i $2 \cdot |x| \geq |w|$.

Na primer, ako je niska `abacabacaba`, tada je traženi prefiks-sufiks x jednak `abacaba`, ostatak s jednak je `caba`, dok je t jednak `abac`.

Dokažimo prethodnu teoremu. Prvo, pretpostavimo da je niska periodična. Tada postoji neprazno $p = p_1 p_2$ tako da je $w = p^n p_1$, za neko $n \geq 2$. Tada je $t = p_1 p_2 = p$, $x = p^{n-1} p_1$, dok je $s = p_2 p_1$. Važi da je $|x| = (n-1)|p| + |p_1|$, a pošto je $n \geq 2$, važi da je $(n-1) \cdot |p| \geq |p|$, pa je $|x| \geq |t|$ i $2 \cdot |x| \geq |x| + |t| = |w|$.

Dokažimo suprotni smer.

Dokažimo najpre indukcijom da ako važi da je $w = xs = tx$ za neprazne t i s, tada postoje reči u i v tako da je $t = uv$, $s = vu$ i $w = (uv)^n \cdot u$. Pretpostavimo da tvrđenje važi za sve reči dužine manje od k i pretpostavimo da je reč w dužine k.

- Ako bi važilo $|x| < |t|$, onda bi iz $w = tx$ sledilo $2 \cdot |x| \geq |w| = |x| + |t| > 2 \cdot |x|$ te dobijamo kontradikciju
- Dakle mora da važi $|x| \geq |t|$. Tada postoji y tako da je $x = ty$, pa na osnovu $xs = tx$ sledi $ys = x$ tj. $ys = ty$. Pošto je s i t neprazno važi da je $|ty| = |x| < |w|$, pa na osnovu induktivne hipoteze postoje

u i v takvi da je $t = uv$, $s = vu$ i n takvo da je $y = (uv)^n u$. Tada je $x = ty = uv(uv)^n u = (uv)^{n+1} u$, pa tvrđenje sledi.

Dokažimo sad da je reč periodična. Neka su u, v i n takvi da je $t = uv$, $s = vu$ i $w = (uv)^n u$ (oni postoje na osnovu prethodne diksusije). Prvo, $n \geq 1$ (ako bi važilo $n = 0$, tada bi iz $w = xs = tx$ važilo da je $w = s = t = u$, a da je x prazna reč, pa odatle sledi da je $|x| = 0$, što je kontradikcija sa pretpostavkom da je $2|x| \geq |w|$). Ako bi važilo da je $n = 1$, tada bi važilo da je $w = uvu$, pa iz $w = tx = xs$ mora da važi da je $x = u, t = uv, s = vu$, što sa $2|x| \geq |w|$ povlači da je $2|u| \geq 2|u| + |v|$ odakle sledi da je $|v| = 0$, odnosno da je $v = \epsilon$ te da je $t = s = x = u$. Tada je $p = p_2 = u$, dok je p_1 prazno, pa je $w = u^2$ i periodična je. Preostaje još slučaj $n \geq 2$. Međutim, tada se može uzeti da je $p_1 = u$, $p_2 = v$, $p = uv$ i važi da je $w = p^n u$, pri čemu je $n \geq 2$, pa je reč ponovo periodična.

Dakle, rešenje se zasniva na tome da pronađemo dužinu d najdužeg prefiks-sufiksa reči w i da se proveri da li važi da je $2d \geq |w|$. To možemo uraditi već prikazanom funkcijom `kmpPreprocesiraj`.

```
void kmpPreprocesiraj(const string & p, vector<int> & b)
{
  int m = b.size();
  int i = 0;
  int j = -1;

  b[i]=j;
  // za svaki prefiks polazne niske
  // racunamo najduzi prefiks-sufiks
  while (i<m){
    // proveravamo da li se najduzi prefiks-sufiks
    // prefiksa koji se zavrsava na prethodnoj poziciji u niski
    // moze prosiriti: to vazi ako je p[i]=p[j]
    while (j>=0 && p[i]!=p[j])
      j=b[j];
    i++;
    j++;
    b[i]=j;
  }
}

int main() {

  string rec;
  cin >> rec;
  int m = rec.size();

  vector<int> b(m+1);
  // racunamo duzinu najduzeg prefiks-sufiksa niske rec
```

```
  kmpPreprocesiraj(rec,b);

  if (2 * b[m] >= m)
    cout << "Niska je periodicna" << endl;
  else
    cout << "Niska nije periodicna" << endl;

  return 0;
}
```

Najduži segment koji je palindrom

Problem: Data je niska s koja sadrži samo mala slova. Prikazati najduži segment niske s koji je palindrom niske s. Ako ima više najdužih segmenata, prikazati segment čiji početak ima najmanji indeks.

Zadatak je moguće rešiti analizom svih segmenata, proverom da li je tekući segment palindrom i određivanjem najdužeg pronađenog palindroma. Pošto je provera da li je niska palindrom složenosti $O(n)$, a segmenata ima $O(n^2)$, složenost ovog pristupa je $O(n^3)$.

```
// provera da li je s[i, j] palindrom
bool palindrom(const string& s, int i, int j){
  while (i < j && s[i] == s[j]) {
    i++;
    j--;
  }
  // ako je i<j onda smo nasli s[i]!=s[j]
  // te segment nije palindrom, inace jeste
  return i >= j;
}

int main() {
  string s;
  cin >> s;
  int n = s.size();

  int maxDuzina = 0, maxPocetak = 0;
  // za sve moguce segmente date niske
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      // ako je u pitanju palindrom
      if (palindrom(s, i, j)) {
        int duzina = j - i + 1;
        // proveravamo da li je duzi od tekuceg maksimuma
        if (duzina > maxDuzina) {
```

```
          maxDuzina = duzina;
          maxPocetak = i;
        }
      }

  // stampamo najduzi palindrom
  cout << s.substr(maxPocetak, maxDuzina) << endl;

  return 0;
}
```

Implementacija se može malo pojednostaviti i dugački palindromi se mogu pronaći brže, ako se primeti da segmente možemo analizirati redom počev od najdužeg segmenta pa unazad do segmenta dužine 1. Primetimo da će sa ovim redosledom obilaska prvi segment koji je palindrom upravo biti najduži palindrom koji tražimo. Primetimo da izlaz iz ugnežđenih petlji nije moguće ostvariti naredbom `break` (time bi se izašlo iz unutrašnje, ali ne i spoljašnje petlje), već izlaz moramo realizovati pomoću pomoćne logičke promenljive. Složenost najgoreg slučaja ovog pristupa je i dalje $O(n^3)$.

```
int main() {

  string s;
  cin >> s;
  // duzina niske s
  int n = s.size();
  // potrebno za prekid dvostruke petlje
  bool nasli = false;

  // proveravamo sve duzine redom, od najvece do najmanje
  // dok ne naidjemo na prvi palindrom
  for(int d = n; d >= 1 && !nasli; d--) {
    // proveravamo rec odredjenu indeksima [p, p + d - 1]
    for(int p = 0; p + d - 1 < n && !nasli; p++) {
      // ako smo naisli na palindrom
      if (palindrom(s, p, p + d - 1)) {
        // ispisujemo ga
        cout << s.substr(p, d) << endl;
        // prekidamo dvostruku petlju
        nasli = true;
      }
    }
  }

  return 0;
}
```

Palindromi poseduju određeno svojstvo inkrementalnosti koje nam može pomoći da pronađemo efikasniji algoritam. Naime, ako je poznat centar palindroma (to može biti bilo neko slovo, bilo pozicija tačno između dva susedna slova) i ako znamo da se k slova oko tog centra slikaju kao u ogledalu (i time grade palindrom), onda za proveru da li se $k+1$ slova oko tog centra slikaju kao u ogledalu ne treba proveravati sve iz početka, već je dovoljno samo proveriti da li su dva slova na spoljnim pozicijama ($k+1$. slovo levo tj. desno od centra) jednaka. Zato efikasnije rešenje dobijamo ako za svako slovo reči odredimo najduži palindrom neparne dužine takav da mu je izabrano slovo centar i za svaku poziciju između dva slova odredimo najduži polinom parne dužine kojima je ta pozicija centar.

Da bismo odredili palindrom sa centrom u slovu s_i, širimo palindrom s_i u desno i u levo za k slova dok se nalazimo unutar reči ($i - k \geq 0, i + k < n$) i dok su odgovarajuća slova jednaka ($s_{i-k} = s_{i+k}$). U trenutku kada se to prvi put naruši dobijamo najduži palindrom sa centrom u s_i (ako se izađe iz reči dalje proširivanje nije moguće, a ako se pronađe različit par slova dalja proširivanja ne mogu više da daju palindrom).

Određivanje najdužeg palindroma sa centrom između dva slova vršimo na veoma sličan način.

Za svaku poziciju (a njih ima $2n - 1$ tj. $O(n)$) nalazimo najduži palindrom sa centrom u njoj šireći tekući palindrom nalevo i nadesno i globalno najduži palindrom nalazimo kao najduži od tih palindroma.

Širenje se obavlja jednim prolaskom i zahteva vreme $O(n)$, pa je ukupna složenost algoritma $O(n^2)$.

```
int main() {

  string s;
  cin >> s;
  // duzina ucitane reci
  int n = s.size();

  // duzina i pocetak najduzeg palindroma
  int maxDuzina = 0, maxPocetak;

  // prolazimo kroz sva slova reci
  for (int i = 0; i < n; i++) {
    int duzina, pocetak;

    // nalazenje najduzeg palindroma neparne duzine ciji je centar
    // slovo s[i]
    int k = 1;
    while (i - k >= 0 && i + k < n && s[i - k] == s[i + k])
      k++;
```

```
    // duzina i pocetak maksimalnog palindroma
    duzina = 2 * k - 1;
    pocetak = i - k + 1;

    // azuriramo maksimum ako je to potrebno
    if (duzina > maxDuzina) {
      maxDuzina = duzina;
      maxPocetak = pocetak;
    }

    // nalazenje najduzeg palindroma parne duzine ciji je centar
    // izmedju slova s[i] i s[i+1]
    k = 0;
    while(i - k >= 0 && i + k + 1 < n && s[i - k] == s[i + k + 1])
      k++;
    // duzina i pocetak maksimalnog palindroma
    duzina = 2 * k;
    pocetak = i - k + 1;

    // azuriramo maksimum ako je to potrebno
    if (duzina > maxDuzina) {
      maxDuzina = duzina;
      maxPocetak = pocetak;
    }
  }

  // izdvajamo i ispisujemo odgovarajuci palindrom
  cout << s.substr(maxPocetak, maxDuzina) << endl;
  return 0;
}
```

Postoji nekoliko tehnika koje mogu da malo skrate implementaciju prethodnog algoritma, objedinjavajući slučajeve palindroma parne i neparne dužine. Zamislimo da se pre prvog, nakon poslednjeg i između svaka dva susedna slova reči postavi specijalni karakter `|`. Na primer, reč `aabcbab` dopunjavamo do reči `|a|a|b|c|b|a|b|`. Na taj način dobijamo to da su sada svi centri palindroma karakteri ovako dopunjene reči i dovoljno je analizirati samo palindrome neparne dužine u njemu. Ovo dopunjavanje je moguće realizovati i fizički (u programu kreirati dopunjenu nisku), što može malo da olakša implementaciju po cenu malo sporijeg programa (doduše ne asimptotski) i dodatnog zauzeća memorije.

```
string dopuni(const string& s) {
  string rez = "^";
  for (int i = 0; i < s.size(); i++)
    rez += "|" + s.substr(i, 1);
  rez += "|$";
  return rez;
```

```
}

int main() {
  string s;
  cin >> s;
  string t = dopuni(s);

  // dovoljno je pronaci najveci palindrom neparne duzine u prosirenoj
  // reci

  int maxDuzina = 0, maxCentar;
  // proveravamo sve pozicije u dopunjenoj reci
  for (int i = 1; i < t.size() - 1; i++) {
    // prosirujemo palindrom sa centrom na poziciji i dokle god je to
    // moguce
    int d = 0;
    while (t[i - d - 1] == t[i + d + 1])
      d++;

    // azuriramo maksimum ako je potrebno
    if (d > maxDuzina) {
      maxDuzina = d;
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  cout << s.substr(maxPocetak, maxDuzina) << endl;
}
```

Da bismo olakšali izlaganje indekse u dopunjenoj reči ćemo nazivati pozicije, a u originalnoj reči samo indeksi. Na primer,

```
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14    - pozicije
|   a   |   a   |   b   |   c   |   b   |   a   |   b   |
    0       1       2       3       4       5       6         - indeksi
```

Primetimo nekoliko činjenica. Ako je polazna reč dužine n, ukupno imamo $N = 2n+1$ poziciju. Slova polazne reči se nalaze na neparnim pozicijama, dok se na parnim pozicijama nalazi specijalni znak `|`. Slovo sa indeksom k se nalazi na poziciji $p = 2k+1$, što znači da se na neparnoj poziciji p nalazi slovo originalne reči sa indeksom $k = \lfloor \frac{p}{2} \rfloor$.

Za svaku poziciju i ($0 \leq i < N$) dužinu palindroma sa centrom na toj poziciji možemo odrediti na isti način, bez obzira na to da li je na toj poziciji slovo ili specijalni znak `|`. Recimo da ćemo ovde podrazumevati dužinu palindroma u originalnoj reči (a ne dopunjenoj) i ona je jednaka broju karaktera sa leve strane

date pozicije u dopunjenoj reči koji su jednaki odgovarajućim karakterima sa desne strane te pozicije u dopunjenoj reči. Na primer, u prethodnom primeru za poziciju 7 to je 3, jer je palindrom `bcb` dužine 3, što odgovara tome da tri karaktera `|b|` sa leve strane karaktera `c` u dopunjenoj reči odgovaraju karakterima `|b|` sa desne strane karaktera `c`.

Na početku, dužinu palindroma d postavljamo na 1, ako je pozicija neparna tj. 0 ako je parna. Zaista, ako je pozicija neparna, na njoj se nalazi slovo koje je samo za sebe palindrom dužine 1 (iz drugog ugla gledano, levo i desno od nje se nalaze `|`, pa je bar jedan karakter jednak). Ako je pozicija parna, oko nje se nalaze dva slova (osim u slučaju krajnjih pozicija) i ne znamo unapred da li su ona jednaka, tako da dužinu palindroma inicijalno postavljamo na 0. Ovim postižemo da su brojevi $i - d$ i $i + d$ parni, što znači da su $i - d - 1$ i $i + d + 1$ neparni i ako su u opsegu $[0, N)$, oni ukazuju na naredna dva karaktera čiju jednakost treba proveriti. Ako su karakteri na odgovarajućim indeksima jednaki (to su indeksi $\lfloor \frac{i-d-1}{2} \rfloor$ i $\lfloor \frac{i+d+1}{2} \rfloor$) onda se d uvećava za 2 (iz jednog ugla gledano, ta dva jednaka karaktera se dodaju tekućem palindromu pa mu se dužina povećava za 2, a iz drugog ugla gledano, ispred prvog i iza drugog se nalaze specijalni znaci `|` koji su sigurno jednaki i njihovu jednakost nije potrebno eksplicitno proveravati). Time se održava i invarijanta da su brojevi $i + d$ i $i - d$ parni i petlja se može nastaviti na isti način sve dok se naiđe na dva različita slova ili se ne naiđe van opsega dopunjene reči.

Recimo još i da se i provera pripadnosti indeksa tj. pozicija opsegu reči može eliminisati ako se polazna reč proširi dodatnim specijalnim karakterima na početku i na kraju (oni moraju biti različiti i obično se koriste `^` i `$`, jer se ti karakteri koriste za označavanje početka i kraja u regularnim izrazima).

```
string dopuni(const string& s) {
  string rez = "^";
  for (int i = 0; i < s.size(); i++)
    rez += "|" + s.substr(i, 1);
  rez += "|$";
  return rez;
}

int main() {
  string s;
  cin >> s;
  string t = dopuni(s);

  // dovoljno je pronaci najveci palindrom neparne duzine u prosirenoj
  // reci

  int maxDuzina = 0, maxCentar;
  // proveravamo sve pozicije u dopunjenoj reci
  for (int i = 1; i < t.size() - 1; i++) {
```

```
    // prosirujemo palindrom sa centrom na poziciji i dokle god je to
    // moguce
    int d = 0;
    while (t[i - d - 1] == t[i + d + 1])
      d++;

    // azuriramo maksimum ako je potrebno
    if (d > maxDuzina) {
      maxDuzina = d;
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  cout << s.substr(maxPocetak, maxDuzina) << endl;
}
```

Posmatrajmo sada kako izgleda dužina najdužeg palindroma za svaku od pozicija u reči `babcbabcbaccba`. Obeležimo ovu dužinu sa d_p. Obeležimo dopunjenu reč sa t. U prvom redu narednog prikaza dati su indeksi i u reči s, u drugom redu proširena reč t, u trećem redu date su pozicije p, a u poslednjem vrednost d_p.

```
   0     1     2     3     4     5     6     7     8     9     10    11    12    13
|  b  |  a  |  b  |  c  |  b  |  a  |  b  |  c  |  b  |  a  |  c  |  c  |  b  |  a  |
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
0  1  0  3  0  1  0  7  0  1  0  9  0  1  0  5  0  1  0  1  0  1  2  1  0  1  0  1  0
```

Posmatrajmo palindrom sa centrom na poziciji 11 – njegova dužina je $d_{11} = 9$ i prostire se od pozicije 2 do pozicije 20.

Posmatrajmo sada dužinu najdužeg palindroma sa centrom na poziciji 12. Pošto je naš palindrom simetričan oko pozicije 11, poziciji 12, odgovara pozicija 10. Znamo da je $d_{10} = 0$. To je zato što je $t_9 \neq t_{11}$. Međutim, mi znamo da je $t_9 = t_{13}$ (pošto su obe unutar našeg palindroma), pa je zato $t_{11} \neq t_{13}$ i važi da je $d_{12} = d_{10} = 0$. Primetimo da ovo možemo konstatovati bez ikakve potrebe za upoređivanjem karaktera.

Posmatrajmo sada dužinu najdužeg palindroma sa centrom na poziciji 13. Njemu odgovara palindrom sa centrom na poziciji 9. Važi da je $d_9 = 1$, jer je $t_8 = t_{10}$ i $t_7 \neq t_{11}$. Na osnovu simetrije palindroma sa centrom u 11, važi da je $t_8 = t_{14}$, da je $t_{10} = t_{12}$, $t_7 = t_{15}$ i da je $t_{11} = t_{11}$. Zato je $t_{14} = t_{12}$ i $t_{15} \neq t_{11}$, pa je $d_{13} = d_9 = 1$.

Naizgled, važi da je za svako i unutar šireg palindroma sa centrom u nekoj poziciji C broj d_i jednak broju $d_{i'}$, gde se i' određuje kao simetrična pozicija poziciji i u odnosu na C (važi da je rastojanje od C do i i i' jednako, pa je $C - i' = i - C$, tj. $i' = C - (i - C)$). No, to nije uvek tačno.

Posmatrajmo d_{15} i njemu odgovarajuću vrednost d_7. One nisu jednake. Zašto? Posmatrajmo šta je ono što možemo zaključiti iz simetrije palindroma sa centrom na poziciji 11. Važi da je t_2 do t_{12} jednako odgovarajućim karakterima t_{20} do t_{10} – to je garantovano simetrijom i nije potrebno proveravati. Međutim, važi da je $d_7 = 7$. Znamo zato i da je $t_1 = t_{13}$, međutim, ne možemo da tvrdimo da je $t_{21} = t_9$, zato to t_{21} nije više deo palindroma sa centrom na poziciji C – proveru da li je $t_{21} = t_9$ je potrebno posebno izvršiti.

Dakle, važi sledeće. Pretpostavimo da je $[L, R]$ palindrom sa centrom na poziciji C (tada je $C - L = R - C$), da je i neki indeks unutar tog palindroma (neka je $C < i < R$) i da je $i' = C - (i - C)$ njemu simetričan indeks. Ako je palindrom sa centrom u i' sadržan u palindromu (L, R) (bez uračunatih krajeva) tj. ako je $L < i' - d_{i'}$, tj. $d_{i'} < i' - L = (C - (i - C)) - (C - (R - C)) = R - i$ tj. $d_{i'} < R - i$, tada je $d_i = d_{i'}$. Dokažimo ovo. Za svaku vrednost $0 \leq j \leq d_{i'}$ treba dokazati da je $t_{i-j} = t_{i+j}$. Zaista, pošto važi da je $L < i' - j$ i da je $i + j < R$, važi da je $t_{i-j} = t_{i'+j}$ i da je $t_{i+j} = t_{i'-j}$. Međutim, pošto je i' centar palindroma dužine $d_{i'}$, važi da je $t_{i'-j} = t_{i'+j}$. Zato je na poziciji i centar palindroma dužine bar $d_{i'}$. Dokažimo da je ovo i gornje ograničenje, tj. dokažimo da je $t_{i-d_{i'}-1} \neq t_{i+d_{i'}+1}$. Pošto je $i + d_{i'} < R$, važi da je $i + d_{i'} + 1 \leq R$, pa je $t_{i-d_{i'}-1} = t_{i'+d_{i'}+1}$ i $t_{i+d_{i'}+1} = t_{i'-d_{i'}-1}$. Međutim, pošto je palindrom sa centrom u i' dužine $d_{i'}$ važi da je $t_{i'-d_{i'}-1} \neq t_{i'+d_{i'}+1}$.

Ako je $[L, R]$ palindrom sa centrom na poziciji C i ako je i neki indeks unutar tog palindroma ($C < i < R$), ali takav da je $d_{i'} \geq R - i$, onda možemo samo da zaključimo da je $d_i \geq R - i$.

Ovo inspiriše naredni algoritam, poznat pod imenom *Manačerov algoritam*. Slično kao u prethodnoj verziji obrađujemo sve pozicije i od 0 do $N - 1$. Pri tom održavamo indekse C i R takve da je $[C - (R - C), R]$ palindrom. Ako je $i \geq R$, tada palindrom sa centrom u i određujemo iz početka, povećavajući za dva dužinu palindroma d_i koja kreće od 0 ili 1 (u zavisnosti od parnosti pozicije i), sve dok je to moguće, isto kao u prethodno opisanom algoritmu. Međutim, ako je $i < R$, tada određujemo $i' = C - (i - C)$ i ako važi da je $d_{i'} < R - i$, postavljamo odmah $d_i = d_{i'}$. Ako je $d_{i'} \geq R - i$, tada dužinu d_i postavljamo na početnu vrednost $R - i$ tj. na $R - i + 1$ tako da su $i - d_i$ i $i + d_i$ parni brojevi, i onda je postepeno povećavamo za 2, sve dok je to moguće (opet, veoma slično kao u prethodno opisanom algoritmu). Ako je pronađeni palindrom sa centrom u i takav da mu desni kraj prevaziđe poziciju R, onda njega proglašavamo za novi palindrom $[L, R]$, postavljajući mu centar $C = i$ i desni kraj $R = i + d_i$. Na početku možemo inicijalizovati $R = C = 0$ (time obezbeđujemo da ne može da važi da je $i < R$ i da se na početku neće koristiti simetričnost okružujućeg palindroma).

```
int main() {
  string s;
  cin >> s;

  // broj pozicija u reci s (pozicije su ili slova originalnih reci, ili su
```

```
// izmedju njih). Npr. niska abc ima 7 pozicija i to |a|b|c|
int N = 2 * s.size() + 1;

// d[i] je duzina najduzeg palindroma ciji je centar na poziciji i
vector<int> d(N);

// znamo da je [L, R] palindrom sa centrom u C
int C = 0, R = 0; // L = C - (R - C)
for (int i = 0; i < N; i++) {
  // karakter simetrican karakteru i u odnosu na centar C
  int i_sim = C - (i - C);
  if (i < R && i + d[i_sim] < R)
    // nalazimo se unutar palindroma [L, R], ciji je centar C
    // palindrom sa centrom u i_sim i palindrom sa centrom u i su
    // celokupno smesteni u palindrom (L, R)
    d[i] = d[i_sim];
  else {
    // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
    // se nalazimo unutar palindroma [L, R], ali je palindrom sa
    // centrom u i_sim takav da nije celokupno smesten u (L, R) - u
    // tom slucajmo znamo da je duzina palindroma sa centrom u i bar
    // R-i, a da li je vise od toga, treba proveriti
    d[i] = i <= R ? R - i : 0;

    // prosirujemo palindrom dok god je to moguce

    // osiguravamo da je i + d[i] stalno paran broj
    if ((i + d[i]) % 2 == 1)
      d[i]++;

    // dok god su pozicije u opsegu i slova na odgovarajucim
    // indeksima jednaka uvecavamo d[i] za 2 (jedno slovo s leva i
    // jedno slovo zdesna)
    while (i - d[i] - 1 >= 0 && i + d[i] + 1 < N &&
        s[(i - d[i] - 1) / 2] == s[(i + d[i] + 1) / 2])
      // ukljucujemo dva slova u palindrom, pa mu se duzina uvecava za 2
      d[i] += 2;
  }

  // ako palindrom sa centrom u i prosiruje desnu granicu
  // onda njega uzimamo za palindrom [L, R] sa centrom u C
  if (i + d[i] > R) {
    C = i;
    R = i + d[i];
  }
}
```

```
  // pronalazimo najvecu duzinu palindroma i pamtimo njegov centar
  int maxDuzina = 0, maxCentar;
  for (int i = 0; i < N; i++) {
    if (d[i] > maxDuzina) {
      maxDuzina = d[i];
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  cout << s.substr(maxPocetak, maxDuzina) << endl;

  return 0;
}
```

Moguće je dokazati da je složenost ovog algoritma linearna (intuitivno, pronalaženje kratkih palindroma zahteva mali broj izvršavanja unutrašnje petlje, dok pronalaženje jednog dugačakog palindroma zahteva duže izvršavanje unutrašnje petlje, ali dovodi do toga da će se u narenim koracima u velikom broju slučajeva u potpunosti izbegava njeno izvršavanje). Preciznije, moguće je pokazati da svako izvršavanje unutrašnje `while` petlje povećava vrednost promenljive R (jer je u slučaju `else` grane $i + d[i] \geq R$, a svako uspešno izvršavanje `while` petlje uvećava vrednost $d[i]$ za 2, te će važiti $i + d[i] > R$) koja se nigde ne smanjuje, a čija je maksimalna vrednost N, te je ukupan broj izvršavanja unutrašnje petlje ograničen sa $O(N)$.

Možemo razmotriti i varijantu koja eliminiše proveru pripadnosti indeksa opsegu reči time što eksplicitno pravi dopunjenu rec.

```
// da bismo uniformno posmatrali palindrome i parne i neparne duzine,
// prosirujemo string dodajuci ^ i $ oko njega i umecuci | izmedju
// svih slova npr. abc -> ^|a|b|c$
string dopuni(const string& s) {
  string rez = "^";
  for (int i = 0; i < s.size(); i++)
    rez += "|" + s.substr(i, 1);
  rez += "|$";
  return rez;
}

int main() {
  string s;
  cin >> s;

  // jednostavnosti radi dopunjavamo rec s
```

```
string t = dopuni(s);
// d[i] je najveci broj takav da je [i - d[i], i + d[i]] palindrom
// to je ujedno i duzina najduzeg palindroma ciji je centar na
// poziciji i (pozicije su ili slova originalnih reci, ili su
// izmedju njih)
vector<int> d(t.size());
// znamo da je [L, R] palindrom sa centrom na poziciji C
int C = 0, R = 0; // L = C - (R - C)
for (int i = 1; i < t.size() - 1; i++) {
  // karakter simetrican karakteru i u odnosu na centar C
  int i_sim = C - (i - C);
  if (i < R && i + d[i_sim] < R)
    // nalazimo se unutar palindroma [L, R], ciji je centar C
    // palindrom sa centrom u i_sim i palindrom sa centrom u i su
    // celokupno smesteni u palindrom (L, R)
    d[i] = d[i_sim];
  else {
    // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
    // se nalazimo unutar palindroma [L, R], ali je palindrom sa
    // centrom u i_sim takav da nije celokupno smesten u (L, R) - u
    // tom slucajmo znamo da je duzina palindroma sa centrom u i bar
    // R-i, a da li je vise od toga, treba proveriti
    d[i] = i <= R ? R-i : 0;
    // prosirujemo palindrom dok god je to moguce krajnji karakteri
    // ^$ obezbedjuju da nije potrebno proveravati granice
    while (t[i - d[i] - 1] == t[i + d[i] + 1])
  d[i]++;
  }

  // ako palindrom sa centrom u i prosiruje desnu granicu
  // onda njega uzimamo za palindrom [L, R] sa centrom u C
  if (i + d[i] > R) {
    C = i;
    R = i + d[i];
  }
}

// pronalazimo najvecu duzinu palindroma i pamtimo njegov centar
int maxDuzina = 0, maxCentar;
for (int i = 1; i < t.size() - 1; i++)
  if (d[i] > maxDuzina) {
    maxDuzina = d[i];
    maxCentar = i;
  }

// ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
```

```
  cout << s.substr((maxCentar - maxDuzina) / 2, maxDuzina) << endl;

  return 0;
}
```

z-algoritam

z-niz niske s dužine n sadrži na poziciji $k = 0, 1, \ldots, n-1$ dužinu najdužeg segmenta niske s koji počinje na poziciji k i prefiks je niske s. Dakle, ako važi $z[k] = p$ onda je niska $s[0..p-1]$ jednaka sa niskom $s[k..k+p-1]$. Mnogi problemi nad niskama mogu se efikasno rešiti korišćenjem z-niza, kao što su ispitivanje da li se neka niska sadrži u drugoj niski, kompresija niske u smislu određivanja najkraće niske tako da se polazna niska može predstaviti nadovezivanjem određenog broja te niske itd. Vrednost niza z na poziciji 0 je suštinski jednaka dužini niske jer je kompletna niska uvek prefiks same sebe, ali nam ovde neće biti od značaja.

z-niz niske `ACBACDACBACBACDA` jednak je:

```
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|
|A|C|B|A|C|D|A|C|B|A| C| B| A| C| D| A|
|-|0|0|2|0|0|5|0|0|7| 0| 0| 2| 0| 0| 1|
```

Vrednost $z[6] = 5$ označava da je segment `ACBAC` (koji počinje na poziciji 6 i dužine je 5) prefiks niske s, dok segment `ACBACB` (koji počinje na istoj poziciji i dužine je 6) nije.

Ostaje pitanje kako konstruisati z-niz za datu nisku. Direktno rešenje bi se sastojalo u tome da se za svaki indeks traži iznova pozicija najdužeg poklapajućeg prefiksa niske koji počinje na tekućoj poziciji u niski. Ovo rešenje bi imalo dve ugnježdene petlje i bilo bi složenosti $O(n^2)$.

```
vector<int> izracunajZNizTrivijalno(string s) {
  int n = s.size();
  vector<int> z(n,0);

  for (int i = 1; i < n; i++) {
```

```
    while (i+z[i] < n && s[z[i]] == s[i+z[i]]){
      z[i]++;
    }
  }
  return z;
}
```

z-algoritam je algoritam za konstrukciju z-niza složenosti $O(n)$. U njemu se vrednosti z-niza izračunavaju sleva nadesno, na osnovu prethodno izračunatih vrednosti niza z. U algoritmu se održava opseg indeksa $[l, d]$ koji označava da je segment $s[l..d]$ prefiks niske s pri čemu je d maksimalno moguće za fiksirano l. Činjenicu da je prefiks $s[0..d-l]$ jednak segmentu $s[l..d]$ koristimo za računanje z-vrednosti za pozicije $l+1, l+2, \ldots, d$.

Dakle, za tekući indeks i za koji treba izračunati vrednost $z[i]$ moguća su dva slučaja:

- $i > d$ – tekuća pozicija je van opsega $[l, d]$ koji smo obradili, te nemamo nikakvih informacija o poziciji i. Stoga vrednost $z[i]$ računamo trivijalnim algoritmom, tj. poređenjem karakter po karakter. Ako dobijemo $z[i] > 0$, onda je potrebno ažurirati opseg $[l, d]$
- $i \leq d$ – tekuća pozicija je unutar poklopljenog segmenta $[l, d]$ te možemo iskoristiti već izračunate vrednosti z-niza za inicijalizaciju vrednosti $z[i]$ (umesto da krećemo od vrednosti 0). Segmenti $s[l..d]$ i $s[0..d-l]$ se poklapaju, pa se poklapaju i segmenti $s[i..d]$ i $s[i-l..d-l]$ pa prilikom računanja vrednosti $z[i]$ možemo krenuti od vrednosti $z[i-l]$. Međutim, vrednost $z[i-l]$ u nekim slučajevima može biti prevelika, jer kada se primeni na poziciju i može da prevaziđe vrednost indeksa d, a to nije dobro jer mi ne znamo ništa o karakterima niske nakon pozicije d. Dakle, maksimalna vrednost poklopljenog dela može biti jednaka broju karaktera od tekuće pozicije i do poslednje pregledane pozicije d, a to je $d-i+1$. Stoga se za početnu vrednost postavlja $z_0[i] = \min\{d-i+1, z[i-l]\}$. Nakon toga utvrđujemo da li se vrednost $z[i]$ može povećati pokretanjem trivijalnog algoritma.

Dakle, algoritam se deli na dva slučaja koji se razlikuju samo po inicijalizaciji vrednosti $z[i]$ nakon čega se postupak svodi na primenu trivijalnog algoritma.

```
#include<iostream>
#include<vector>
using namespace std;

// funkcija koja izracunava sve elemente z-niza
vector<int> izracunajZNiz(const string &s) {

  int n = s.size();
  vector<int> z(n);
  int l = 0;
```

```
  int d = 0;

  for (int i = 1; i < n; i++) {
    // ako je tekuca pozicija unutar opsega [l,d]
    // koristimo prethodno izracunatu vrednost za inicijalizaciju
    if (i <= d)
      z[i] = min(d-i+1,z[i-l]);
    // preskacemo proveru karaktera od pozicije i do pozicije z[i];
    // poredimo karakter po karakter u niski i sve dok
    // se karakteri poklapaju povecavamo vrednost z[i]
    while (i+z[i] < n && s[z[i]] == s[i+z[i]])
      z[i]++;
    // ako je nova vrednost desnog kraja intervala poklapanja
    // veca od prethodne vrednosti, azuriramo interval [l,d]
    if (i+z[i]-1 > d){
      l = i;
      d = i+z[i]-1;
    }
  }
  return z;
}

int main(){

  string niz="ACBACDACBACBACDA";
  int n=niz.size();

  vector<int> zNiz = izracunajZNiz(niz);
  cout << "z-niz date niske je ";
  for (int i=0; i<n; i++){
    cout << zNiz[i] << " ";
  }
  cout << endl;

  return 0;
}
```

Primer: Razmotrimo konstrukciju z-niza za nisku `ACBACDACBACBACDA`:

Na početku, opseg $[l,d] = [0,0]$ te vrednosti z-niza na pozicijama 1, 2 i 3 računamo trivijalnim algoritmom i dobijamo $z[1] = z[2] = 0, z[3] = 2$. Nakon izračunate vrednosti $z[3]$ ažuriramo opseg $[l,d] = [3,4]$.

```
       l_d
       | |
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|
|A|C|B|A|C|D|A|C|B|A| C| B| A| C| D| A|
```

```
|-|0|0|2|0|0|5|?|?|?| ?| ?| ?| ?| ?| ?|
```

Nakon toga, vrednost 4 dobijamo na osnovu $z[4] = z[1] = 0$. Vrednosti $z[5]$ i $z[6]$ dobijamo pokretanjem trivijalnog algoritma i dobijamo $z[5] = 0$, a a $z[6] = 5$, i tekući $[l, d]$ opseg postaje $[6, 10]$.

```
             l________d
             |        |
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|
|A|C|B|A|C|D|A|C|B|A| C| B| A| C| D| A|
|-|0|0|2|0|0|5|?|?|?| ?| ?| ?| ?| ?| ?|
```

Nakon ovog koraka, možemo efikasno izračunati naredne z-vrednosti, jer znamo da su niske $s[0..4]$ i $s[6..10]$ jednake. Najpre, s obzirom da je $z[1] = z[2] = 0$ dobijamo i da je $z[7] = z[8] = 0$. Nakon toga, s obzirom na to da je $z[3] = 2$, znamo da je $z[9] \geq 2$. Međutim, nemamo informacije o niski nakon pozicije 10, pa poredimo segmente karakter po karakter.

```
             l________d
             |        |
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|
|A|C|B|A|C|D|A|C|B|A| C| B| A| C| D| A|
|-|0|0|2|0|0|5|0|0|?| ?| ?| ?| ?| ?| ?|
```

Ispostavlja se da je $z[9] = 7$, pa je novi $[l, d]$ opseg jednak $[9, 15]$.

```
                   l________________d
                   |                |
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|
|A|C|B|A|C|D|A|C|B|A| C| B| A| C| D| A|
|-|0|0|2|0|0|5|0|0|7| ?| ?| ?| ?| ?| ?|
```

Nakon ovoga, sve preostale vrednosti z-niza mogu se odrediti na osnovu informacija koje se već nalaze u z-nizu.

```
                   l________________d
                   |                |
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|
|A|C|B|A|C|D|A|C|B|A| C| B| A| C| D| A|
|-|0|0|2|0|0|5|0|0|7| 0| 0| 2| 0| 0| 1|
```

Pokažimo da je prikazani algoritam linearne vremenske složenosti. Nakon svakog neuspešnog poređenja karaktera, tekuća iteracija `for` petlje se završava, a ukupan broj iteracija je n. Stoga ukupno možemo imati najviše n nepoklapanja karaktera. Svaki karakter koji se poklopi u unutrašnjoj `while` petlji se više nikada ne poredi, te je i broj uspešno poklopljenih karaktera n. Odavde se dobija da je ukupna složenost algoritma $O(n)$.

Često je stvar izbora da li koristiti heširanje niski ili z-algoritam. Za razliku od heširanja, z-algoritam uvek radi i ne postoji rizik od kolizije. Međutim,

z-algoritam je nešto teži za implementaciju i postoje neki problemi koji se mogu rešiti samo heširanjem.

Traženje uzorka u tekstu

Neka su $T = t_0t_1 \dots t_{n-1}$ i $P = p_0p_1 \dots p_{m-1}$ dve niske (dva niza karaktera iz konačne azbuke). Za prvi od njih reći ćemo da je *tekst*, a za drugi da je *uzorak*. Segment niske T je niska $t_it_{i+1} \dots t_j$ uzastopnih karaktera iz T.

Problem: Za dati tekst $T = t_0t_1 \dots t_{n-1}$ i uzorak $P = p_0p_1 \dots p_{m-1}$ ustanoviti da li postoji segment niske T jednak P, a ako postoji, pronaći sva njegova pojavljivanja u tekstu.

Tipična situacija u kojoj se nailazi na ovaj problem je kad se traži neka reč u tekstualnoj datoteci. Problem ima primenu i u raznim drugim oblastima, na primer u molekularnoj biologiji, gde je korisno pronaći neke uzorke u okviru velikih molekula RNK ili DNK.

Naivni algoritam

Direktni algoritam bi poredio uzorak P sa svim mogućim segmentima $t_kt_{k+1} \dots t_{k+m-1}$ teksta T dužine m, pri čemu k uzima vrednosti redom 0, $1, \dots, n - m + 1$. Upoređivanje uzorka sa segmentom vrši se karakter po karakter sleva udesno, sve dok se ne ustanovi da su svi karakteri uzorka jednaki odgovarajućim karakterima segmenta (u tom trenutku prekida se dalje pregledanje segmenta), ili dok se ne naiđe na neslaganje $p_i \neq t_{k+i-1}$ za neko i, $0 \leq i \leq m - 1$:

$$\begin{array}{ccccccccc}
 & & & & j = k+i-1 & & & & \\
 & & & & \downarrow & & & & \\
t_0 & \dots & t_k & \dots & t_j & \dots & t_{k+m-1} & \dots & t_{n-1} \\
 & & p_0 & \dots & p_i & \dots & p_{m-1} & & \\
 & & & & \uparrow & & & & \\
 & & & & i & & & &
\end{array}$$

U drugom slučaju uzorak se "pomera" za jedan karakter udesno, odnosno nastavlja se sa proverom jednakosti p_0 sa t_{k+1}. Broj upoređivanja karaktera manji je od mn, pa je složenost ovog algoritma $O(mn)$ u najgorem slučaju.

```
using namespace std;
#include <iostream>

// algoritam grube sile za trazenje uzorka u tekstu
void pronadji(const string& uzorak, const string& tekst)
{
  int m = uzorak.size();
```

```
  int n = tekst.size();

  int postoji = 0;
  // za sve moguce pocetke pojavljivanja uzorka u tekstu
  for (int i = 0; i <= n-m; i++) {

    int j;
    // preklapamo karaktere teksta i uzorka
    // dok ne naidjemo na neslaganje
    for (j = 0; j < m; j++)
      if (tekst[i+j] != uzorak[j])
         break;

    // ako je j stiglo do m, to znaci da je pronadjen
    // kompletan uzorak u tekstu
    if (j == m){
      postoji = 1;
      cout << "Uzorak je pronadjen na poziciji " << i << endl;
    }
  }
  if (!postoji)
    cout << "Uzorak se ne nalazi u tekstu" << endl;
}

int main()
{
  string tekst = "abrakadabra";
  string uzorak = "ra";
  pronadji(uzorak, tekst);
  return 0;
}
```

Rabin-Karpov algoritam

Rabin i Karp su 1987. godine osmislili algoritam koji rešava ovaj problem korišćenjem koncepta heširanja stringova. Najpre se izračunava heš vrednost niske P i heš vrednosti svih prefiksa teksta T. Korišćenjem ovih heš vrednosti moguće je uporediti proizvoljni segment niske T dužine m sa niskom P u konstantnom vremenu. Dakle, na ovaj način poredimo svaki segment teksta dužine m sa uzorkom. Ukupna složenost algoritma je $O(n+m)$: $O(m)$ je složenost izračunavanja heš vrednosti uzorka i svih segmenata teksta dužine m (za koje su nam potrebni heš vrednosti svih prefiksa date niske čije je izračunavanje složenosti $O(n)$), a $O(n)$ je ukupna složenost poređenja svakog segmenta teksta T dužine m sa uzorkom.

```
#include<iostream>
#include<vector>
using namespace std;

// Rabin-Karpov algoritam za trazenje uzorka u tekstu
void rabinKarp(const string &uzorak, const string &tekst){

  int M = uzorak.size();
  int N = tekst.size();
  int p = 31;
  int m = 1e9 + 9;

  // racunamo stepene broja p
  vector<long long> pStepen(max(M,N));
  pStepen[0] = 1;
  for(int i=1; i<pStepen.size(); i++)
    pStepen[i] = (pStepen[i-1]*p)%m;

  // racunamo hesh vrednosti svih prefiksa datog teksta
  vector<long long> hesh(N+1,0);
  for(int i=0; i<N; i++)
    hesh[i+1] = (hesh[i]+(tekst[i]-'a'+1)*pStepen[i])%m;

  // racunamo hesh vrednost datog uzorka
  long long heshUzorka = 0;
  for(int i=0; i<M; i++)
    heshUzorka = (heshUzorka + (uzorak[i]-'a'+1)*pStepen[i])%m;

  // vektor pocetnog indeksa pojavljivanja uzorka u tekstu
  vector<int> pojave;
  for (int i=0; i<=N-M; i++){

    long long hTekuce = (hesh[i+M]+m-hesh[i])%m;
    // ako se poklapaju hesh vrednosti, pamtimo indeks i
    if (hTekuce == (heshUzorka*pStepen[i])%m)
      pojave.push_back(i);
  }
  if (pojave.size())
    cout << "Uzorak se u tekstu pojavljuje na pozicijama: ";
  for (int i=0; i<pojave.size(); i++)
    cout << pojave[i] << " ";
}

int main(){

  string tekst = "banana";
```

```
  string uzorak = "ana";
  rabinKarp(uzorak, tekst);
  return 0;
}
```

Algoritam zasnovan na z-nizu

Često se prilikom obrade niski konstruiše jedna duža niska koja se sastoji od većeg broja niski razdvojenih specijalnim karakterima. U ovom slučaju, možemo konstruisati nisku $P\#T$, gde su uzorak P i tekst T razdvojeni specijalnim karakterom # koji se ne javlja u niskama. z-niz niske $P\#T$ nam može reći gde se u tekstu T pojavljuje uzorak P jer će takve pozicije imati z-vrednost jednaku dužini uzorka P.

Na primer, ako je `P=ra`, a `T=abrakadabra`, z-niz niske $P\#T$ ima sledeći sadržaj:

```
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|
|r|a|#|a|b|r|a|k|a|d| a| b| r| a|
|-|0|0|0|0|2|0|0|0|0| 0| 0| 2| 0|
```

Pozicije 5 i 12 imaju vrednost 2 (jednaku dužini uzorka) što nam govori da se počev od ovih pozicija u tekstu javlja uzorak P. Složenost algoritma je linearna jer je dovoljno konstruisati z-niz i proći kroz njegove vrednosti.