Knut-Moris-Pratov algoritam

U naivnom algoritmu za traženje uzorka u tekstu 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. Na primer, u slučaju teksta `aaaaccccccc` i uzorka `aaaab`, nakon uspešnog poređenja četiri karaktera `a` teksta i uzorka i nepoklapanja karaktera `c` teksta i karaktera `b` uzorka, uzorak bismo pomerili udesno za jednu poziciju. Nakon toga bismo ponovo (uspešno) poredili tri karaktera `a` teksta i uzorka (iako smo već mogli da zaključimo da će se ti karakteri poklopiti) a zatim nakon nepoklapanja karaktera `c` sa karakterom `a` opet pomerili uzorak za jednu poziciju udesno. Ovakav postupak vodi algoritmu 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 a koji je poznat pod nazivom *Knut-Moris-Pratov algoritam* (ili skraćeno *KMP algoritam*) zasniva se na ideji da se iskoriste informacije dobijene prethodnim poređenjima karaktera i da se nikada iznova ne porede karakteri teksta koji su se prethodno uspešno 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 složenost Knut-Moris-Pratovog algoritma $O(m+n)$. Knut-Moris-Pratov algoritam se najčešće koristi kada se uzorak traži u dugačkom tekstu čija je azbuka male kardinalnosti, na primer kada se pretražuju molekuli DNK koji su formulisani nad četvoroslovnom azbukom A, C, G, T.

Pored Knut-Moris-Pratovog algoritma, poznati efikasni algoritam za traženje uzorka u tekstu je i Bojer-Murov algoritam, koji poredi karaktere uzorka i teksta zdesna ulevo i koji recimo koristi alat `grep`.

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*, odnosno *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 niska u istovremeno i pravi prefiks i pravi sufiks niske x.

Primer: Neka je `x=abacab`. Označimo sa ϵ praznu nisku. 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 kada dođe do nepoklapanja karaktera uzorka i teksta.

Primer:

```
        012345678...
tekst:  abcabcabd...
uzorak: abcabd
           abcabd
```

Karakteri na pozicijama od 0 do 4 u uzorku i tekstu su se poklopili, dok se karakteri na poziciji 5 razlikuju. Interesuje nas za koliko najmanje treba pomeriti uzorak udesno u odnosu na tekst, tako da se kraći prefiks uzorka i dalje poklapa sa tekstom. Nekoliko poslednjih karaktera poklopljenog prefiksa uzorka treba da se poklope sa nekoliko prvih karaktera poklopljenog prefiksa uzorka, te je nama, u stvari, potreban neki prefiks-sufiks poklopljenog prefiksa uzorka. S obzirom na to da želimo da se pomerimo udesno što je manje moguće, mi u stvari tražimo maksimalni (najduži) prefiks-sufiks poklopljenog prefiksa uzorka. U ovom primeru, poklopljeni prefiks uzorka je `abcab` i njegova dužina je 5. Njegov najduži prefiks-sufiks je `ab` i dužine je 2. Dakle, uzorak pomeramo udesno u odnosu na tekst za $5 - 2 = 3$ karaktera. Ako je uzorak `abcabd` i tekst počinje sa `abcdxy`, maksimalni prefiks-sufiks pokloljenog prefiksa uzorka `abc` je ϵ te uzorak pomeramo udesno za $3 - 0 = 3$ karaktera.

Pošto broj poklopljenih karaktera uzorka i teksta zavisi i od samog teksta, odnosno ne znamo unapred za koje prefikse uzorka će nam biti potrebni najduži prefiksi-sufiksi, u fazi preprocesiranja potrebno je odrediti dužinu najdužeg prefiks-sufiksa svakog prefiksa datog uzorka. 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.

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

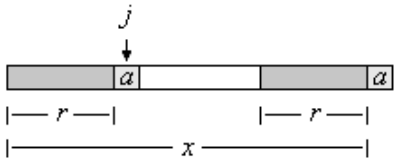
Teorema: Neka su r i s prefiks-sufiksi niske x, pri čemu važi $|r| < |s|$. Onda je niska r prefiks-sufiks niske s.

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

r prefiks-sufiks niske s (slika 1).

Ako je niska s najduži prefiks-sufiks niske x, sledeći po redu najduži prefiks-sufiks niske x se dobija kao najduži prefiks-sufiks niske s. Na sličan način dobija se i naredni po redu najduži prefiks-sufiks niske x.

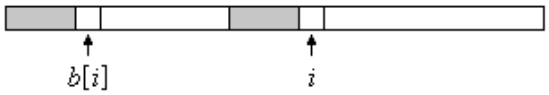
Razmotrimo prefiks-sufiks r prefiksa x uzorka i neka je a karakter uzorka nakon prefiksa x (slika 2). Niska ra biće prefiks-sufiks niske xa, ako je ra prefiks niske xa. Ovo zapažanje nam je od važnosti kako bismo mogli da inkrementalno računamo najduže prefiks-sufikse svih prefiksa datog uzorka.



Slika 2: Proširivanje prefiks-sufiksa.

U fazi preprocesiranja uzorka izračunavaju se vrednosti niza b dužine $m+1$, tako da $b[i]$ 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 neki prefiks-sufiks prefiksa $p_0 p_1 \ldots p_{i-1}$ dužine i može proširiti karakterom p_i. Pošto je $b[i]$ dužina najdužeg prefiks-sufiksa prefiksa uzorka dužine i, karakteri $p_0, p_1, \ldots, p_{b[i]-1}$ i $p_{i-b[i]}, \ldots, p_{i-1}$ uzorka se poklapaju i potrebno je proveriti da li važi $p_{b[i]} = p_i$ (slika 3). Ako se ne poklapaju, razmatra se prvi kraći prefiks-sufiks prefiksa uzorka dužine i. On će biti jednak najdužem prefiks-sufiksu najdužeg prefiks-sufiksa prefiksa uzorka dužine i, odnosno $b[b[i]]$. Stoga će prefiks-sufiksi prefiksa uzorka dužine i biti ispitivani u opadajućem redosledu dužina iz skupa vrednosti $b[i], b[b[i]], \ldots$



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

Dakle, prilikom računanja vrednosti $b[i+1]$, promenljiva j će u petlji uzimati 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 prefiksa uzorka dužine i 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 $p_0 \ldots p_i$ i nju treba smestiti na poziciju $b[i+1]$.

```
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 uzorka duzine i
    // 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: Vrednost $b[5]$ uzorka `p = ababaa` jednaka je 3 jer za prefiks `ababa` dužine 5 važi da je njegov najduži prefiks-sufiks jednak `aba` i dužine je 3. U nastavku je ilustrovan postupak izračunavanja vrednosti dužina najdužih prefiks-sufiksa za sve prefikse uzorka p.

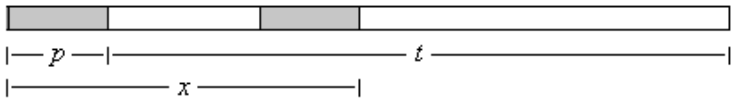
```
b[0] = -1 // prazna niska nema prefiks-sufiks
--------
// i = 0, j = -1
i = 1, j = 0 // uvecavamo i i j za 1
b[1] = 0 // smestamo narednu vrednost u niz b
// najduzi prefiks-sufiks prefiksa a je prazna niska
```

```
--------
// i = 1, j = 0
p_1 != p_0 => j = b[0] = -1 // ne uspeva poredjenje karaktera
i = 2, j = 0 // uvecavamo i i j za 1
b[2] = 0 // smestamo narednu vrednost u niz b
// najduzi prefiks-sufiks prefiksa ab je prazna niska
--------
// i = 2, j = 0
p_2 = p_0 // uspeva poredjenje karaktera
i = 3, j = 1 // uvecavamo i i j za 1
b[3] = 1 // smestamo narednu vrednost u niz b
// najduzi prefiks-sufiks prefiksa aba je niska a duzine 1
--------
// i = 3, j = 1
p_3 = p_1 // uspeva poredjenje karaktera
i = 4, j = 2 // uvecavamo i i j za 1
b[4] = 2 // smestamo narednu vrednost u niz b
// najduzi prefiks-sufiks prefiksa abab je niska ab duzine 2
--------
// i = 4, j = 2
p_4 = p_2 // uspeva poredjenje karaktera
i = 5, j = 3 // uvecavamo i i j za 1
b[5] = 3 // smestamo narednu vrednost u niz b
// najduzi prefiks-sufiks prefiksa ababa je niska a duzine 3
--------
// i = 5, j = 3
p_5 != p_3 => j = b[3] = 1 // ne uspeva poredjenje karaktera
p_5 != p_1 => j = b[1] = 0 // ne uspeva poredjenje karaktera
p_5 = p_0 // uspeva poredjenje karaktera
i = 6, j = 1 // uvecavamo i i j za 1
b[6] = 1 // smestamo narednu vrednost u niz b
// najduzi prefiks-sufiks prefiksa ababaa je niska a duzine 1
--------


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

Prikazani algoritam preprocesiranja uzorka je mogao biti primenjen umesto na uzorak p na nisku pt, gde je sa p označen uzorak, a sa t tekst u kome se uzorak traži. Ako se računaju dužine maksimalnih prefiks-sufiksa svih prefiksa niske pt do dužine m ($|p| = m$), onda ako je dužina maksimalnog prefiks-sufiksa jednaka m, to odgovara pronalaženju uzorka p u tekstu t (slika 4).

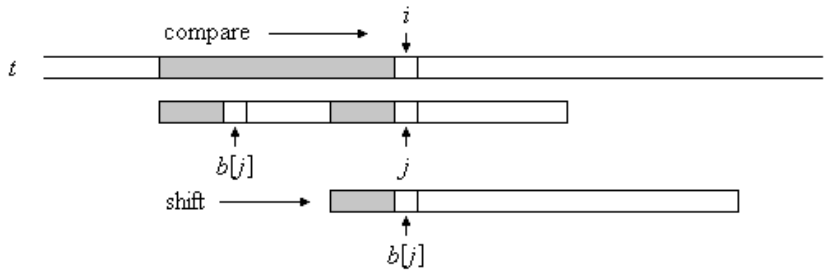
Dakle, algoritam kojim se traži uzorak u tekstu je jako sličan algoritmu preprocesiranja uzorka. Kada se u unutrašnjoj `while` petlji naiđe na nepoklapanje



Slika 4: Prefiks-sufiks dužine $m = |p|$ prefiksa x niske pt.

karaktera na poziciji j u uzorku i poziciji i u tekstu, razmatra se najduži prefiks-sufiks poklapajućeg prefiksa dužine j uzorka (slika 5). Porede se karakter na poziciji $b[j]$ u uzorku i karakter na poziciji i u tekstu i ako se naredni karakter ne poklapa, razmatra se naredni prefiks-sufiks poklapajućeg prefiksa uzorka po dužini i sve tako dok ili ne dođemo u situaciju da prefiks-sufiks prefiksa uzorka ne postoji ($j = -1$) ili dok se naredni karakter uzorka i teksta ne poklope. 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 u odnosu na tekst na osnovu svog najdužeg prefiks-sufiksa.



Slika 5: Pomeranje uzorka udesno u odnosu na tekst kada se naiđe na nepoklapanje karaktera na poziciji j u uzorku i karaktera na poziciji i u tekstu.

```
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++;
    // ako smo poklopili svih m karaktera uzorka izvestavamo o tome
    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  ...
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 promenljive 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 ukupna složenost preprocesiranja uzorka $O(m)$. Potpuno analogno se može zaključiti da je složenost algoritma za traženje 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.

Ispitivanje periodičnosti niske

Reč w je *periodična* ako postoji neprazna reč $p = p_1p_2$ i prirodan broj $n \geq 2$ tako da je $w = p^n p_1$. Na primer, reč *abacabacabacab* je periodična jer se u njoj ponavlja reč *abac*, pri čemu se poslednje ponavljanje ne završava celo već se zaustavlja sa *ab*, tj. reč je jednaka $(abac)^3ab$.

Problem: Napisati program koji proverava da li je reč w periodična.

Problem 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 w periodična sa periodom p čija je dužina d, ako i samo ako za svako i za koje je $0 \leq i$ i $i + d < |w|$ važi da je $w_i = w_{i+d}$. Problem 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 $2|x| \geq |w|$.

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.

Pretpostavimo da je niska w periodična i pokažimo da ona ima prefiks-sufiks dužine bar polovine niske. Iz uslova da je niska w periodična sledi da postoji neprazna reč $p = p_1p_2$ tako da je $w = p^np_1$, za neko $n \geq 2$. Tada je $t = p_1p_2 = p$, $x = p^{n-1}p_1$, dok je $s = p_2p_1$. Zaista proverimo: $xs = p^{n-1}p_1 \cdot p_2p_1 = p^np_1 = w$ i $tx = p \cdot p^{n-1}p_1 = p^np_1 = w$.

Pokažimo i da je $2|x| \geq |w|$. Važi da je $|x| = (n-1)|p| + |p_1|$, a iz $n \geq 2$ sledi $(n-1) \cdot |p| \geq |p|$, pa je $|x| = (n-1) \cdot |p| + |p_1| \geq |p| + |p_1| = |t| + |p_1| \geq |t|$, te je $2 \cdot |x| \geq |x| + |t| = |w|$.

Dokažimo suprotni smer.

Pretpostavimo da niska w ima prefiks-sufiks x dužine bar $|w|/2$ i da važi $w = xs = tx$. Pokažimo da je niska w periodična.

Jasno je da su dužine niski s i t jednake. Iz uslova $|x| \geq |w|/2$ sledi da je $|s| = |t| \leq |x|$.

Neka je $|w| = q|t|+r, 0 \leq r < |t|$. Važi $q \geq 2$ (u protivnom bi važilo $|w| \leq |t|+r < 2|t|$ što zajedno sa pretpostavkom $|w| \leq 2|x|$ daje $2|w| < 2|x| + 2|t| \leq 2|w|$ i dobili bismo kontradikciju).

Neka je $w = w_1w_2 \dots w_qa$, gde je $|w_1| = |w_2| = \dots = |w_q| = |t|$. Iz $w = tx$ sledi $w_1 = t$ i $w_2 \dots w_qa = x$. S obzirom na to da je i $w = xs = w_2 \dots w_qas$ važi redom $w_2 = w_1, w_3 = w_2, \dots, w_{q-1} = w_q$. Iz uslova $as = w_qa$ dobija se da je a prefiks niske w_q. Odavde dobijamo $w = t^qa$, pri čemu je a prefiks niske t, te je reč w periodična.

Dakle, rešenje ovog problema se zasniva na tome da pronađemo dužinu d najdužeg prefiks-sufiksa reči w i da se proveri da li važi $2d \geq |w|$. 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;
}
```

Složenost ovog algoritma jednaka je složenosti funkcije za preprocesiranje u KMP algoritmu, odnosno jednaka je $O(|w|)$.

Najduži segment koji je palindrom

Problem: Data je niska s koja sadrži samo mala slova. Odrediti najduži segment niske s koji je palindrom. Ako ima više najdužih segmenata koji su palindromi, prikazati segment čiji početak ima najmanji indeks. Na primer, ako je niska s jednaka `ananas` potrebno je odštampati segment `anana`, za nisku `najjaci` segment `ajja`, a za nisku `list` segment `l`.

Provera svih segmenata

Problem 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 dužine n palindrom složenosti $O(n)$, a segmenata niske dužine n ukupno ima $O(n^2)$, složenost ovog pristupa je $O(n^3)$.

```
// provera da li je segment 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;
}
```

Provera svih segmenata u redosledu opadajuće dužine

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 za koji se utvrdi da je palindrom upravo biti najduži palindrom koji tražimo. Ako segmente iste dužine razmatramo u rastućem redosledu indeksa leve granice, u slučaju kada postoji više palindroma iste najveće dužine, kao prvi će biti pronađen onaj koji ima najmanju vrednost indeksa leve granice, kao što je i traženo.

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

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

  // proveravamo sve duzine d 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;
}
```

Primetimo da izlaz iz ugnežđenih petlji nije moguće ostvariti naredbom `break` (na taj način 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 $O(n^3)$.

Provera centara

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 na taj način 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 susedna slova odredimo najduži palindrom parne dužine kojima je ta pozicija centar.

Da bismo odredili najduži 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 $i+k < n$) 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 slovu 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 slova s_i i s_{i+1} vršimo na analogan način: dok smo unutar reči ($i-k \geq 0$ i $i+k+1 < n$) širimo palindrom sve dok važi $s_{i-k} = s_{i+k+1}$.

Za svaku poziciju koja može biti centar palindroma (a njih ima n za slova niske i $n-1$ za pozicije između dva slova niske, tj. ukupno $2n-1$, odnosno $O(n)$) nalazimo najduži palindrom sa centrom u njoj šireći tekući palindrom nalevo i nadesno. Globalno najduži palindrom nalazimo kao najduži od dobijenih palindroma.

Širenje za fiksirani centar palindroma se obavlja u jednom prolasku i zahteva vreme $O(n)$, pa je ukupna složenost ovog 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;
}
```

Eksplicitna dopuna reči i pozicija

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 fizički realizovati tako što se u programu eksplicitno kreira dopunjena niska. Ovo može da malo olakša implementaciju po cenu nešto sporijeg programa (doduše ne asimptotski) i dodatnog zauzeća memorije.

Napomenimo još i to 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: ti karakteri moraju biti međusobno različiti i različiti od ostalih karaktera u niski. Obično se u te svrhe koriste `^` i `$`, jer se ti karakteri koriste za označavanje početka i kraja u regularnim izrazima.

Dužinu palindroma razmatraćemo u odnosu na polaznu (a ne dopunjenu) reč.

```
// da bismo uniformno posmatrali palindrome i parne i neparne duzine,
// prosirujemo nisku dodajuci ^ i $ oko njega i umecuci | izmedju
// svih slova, na primer 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;
  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;
}
```

Implicitna dopuna reči i pozicija

Da bismo olakšali naredno izlaganje, indekse u dopunjenoj reči (bez dodatih oznaka početka i kraja reči) nazivaćemo 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 $N = 2n + 1$ pozicija u dopunjenoj reči. Slova polazne reči se nalaze na neparnim pozicijama, dok se na parnim pozicijama nalazi specijalni karakter `|`. Slovo sa indeksom k se nalazi na poziciji $p = 2k + 1$, što znači da se na neparnoj poziciji p nalazi slovo polazne 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 karakter `|`. Recimo da ćemo ovde razmatrati dužinu palindroma u polaznoj 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 5, jer je palindrom `abcba` dužine 5, što odgovara tome da pet karaktera `|a|b|` sa leve strane karaktera `c` u dopunjenoj reči odgovaraju karakterima `|b|a|` sa desne strane karaktera `c`. Slično važi i za parne pozicije (na primer 2).

Na početku, dužinu palindroma d postavljamo na 1, ako je pozicija i neparna tj. 0 ako je parna. Zaista, ako je pozicija neparna, na njoj se nalazi slovo polazne reči koje je samo za sebe palindrom dužine 1 (iz drugog ugla gledano, levo i desno od ove pozicije se nalaze karakteri `|`, 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. Na ovaj način postižemo da su brojevi $i - d$ i $i + d$ parni, što znači da su brojevi $i - d - 1$ i $i + d + 1$ neparni i ako su u opsegu $[0, N)$, oni ukazuju na naredna dva karaktera polazne niske čiju jednakost treba proveriti. Ako su karakteri polazne reči 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). Na taj način se održava i invarijanta da su brojevi $i + d$ i $i - d$ parni i petlja se može nastaviti na isti način sve dok se ne naiđe na dva različita slova ili se izađe van opsega dopunjene reči.

```
int main() {
  string s;
  cin >> s;
  // broj pozicija u reci s
  int N = 2 * s.size() + 1;
  int maxDuzina = 0, maxCentar;
  // za svaki moguci centar palindroma
  for (int i = 0; i < N; i++) {
    // d postavljamo na 0 za parnu, a na 1 za neparnu poziciju
    int d = 0;
    if (i % 2 == 1)
      d = 1;

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

```
    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;
  return 0;
}
```

Manačerov algoritam

Posmatrajmo sada kako izgleda dužina najdužeg palindroma sa centrom u svakoj od pozicija p u reči `s = babcbabcbaccba`. Obeležimo ovu dužinu sa d_p. Obeležimo dopunjenu reč sa t. U prvom redu narednog prikaza dati su indeksi i u polaznoj reči s, u drugom redu proširena reč t, u trećem redu date su pozicije p u proširenoj reči, 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, na primer, najduži palindrom sa centrom u poziciji 11 – njegova dužina je $d_{11} = 9$ i prostire se od pozicije 2 do pozicije 20.

Posmatrajmo sada dužinu najdužeg palindroma sa centrom u poziciji 12. Pošto je prethodno određeni 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 pozicije 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 novim upoređivanjem karaktera.

Posmatrajmo sada dužinu najdužeg palindroma sa centrom u 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 poziciju 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 sada vrednost d_{15} i njoj 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 simetrično u odnosu na poziciju 11 važi da je $t_{21} = t_9$, zato što t_{21} nije više deo palindroma sa centrom na poziciji C. Dakle, proveru da li važi $t_{21} = t_9$ je potrebno posebno izvršiti.

Stogax, 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' u potpunosti sadržan u palindromu (L, R) (bez uračunatih krajeva) tj. ako je $L < i' - d_{i'}$, tj. $d_{i'} < i' - L = (C - (i - C)) - (C - (R - C)) = R - i$, 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 se na poziciji i nalazi 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 nazivom *Manačerov algoritam*. Slično kao u prethodnoj verziji algoritma obrađujemo sve pozicije i od 0 do $N - 1$. Pri tom održavamo indekse C i R takve da je $[C - (R - C), R]$ najdesniji do sad pronađeni 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 vrednost $i' = C - (i - C)$ 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 $R = C = 0$ (na taj način 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)
  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
      // bar R - i, a da li je vise od toga, treba proveriti
      d[i] = i <= R ? R - i : 0;

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

      // dok god su pozicije u dozvoljenom 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 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 dugačkog palindroma zahteva duže izvršavanje unutrašnje petlje, ali dovodi do toga da će se u narednim koracima u velikom broju slučajeva u potpunosti izbegavati njeno izvršavanje. Preciznije, 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 na taj način što eksplicitno pravi dopunjenu reč.

```
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;
}
```