Algoritmi za rad sa tekstom

Heširanje niski

Heširanje niski (eng. string hashing) podrazumeva konverziju niske u ceo broj iz određenog opsega. Algoritmi heširanja mogu biti korisni u rešavanju različitih problema nad tekstom, kao što su ispitivanje da li se jedna niska javlja u drugoj, pronalaženje najduže niske koja se javlja bar dva puta kao segment date niske, kompresija niske, određivanje broja palindromskih segmenata u datoj niski i dr. Tehnika heširanja niski je dobila na popularnosti kada su Rabin i Karp osmislili algoritam za traženje jedne niske u drugoj zasnovan na heširanju. Heširanje ima primenu i u verifikaciji lozinki: prilikom logovanja na neki sistem, računa se heš vrednost lozinke i šalje serveru na proveru. Lozinke se na serveru čuvaju u heširanom obliku, iz bezbedonosnih razloga.

Heširanje se često koristi da bi se algoritmi grube sile učinili efikasnijim. Razmotrimo problem upoređivanja dve niske karaktera s i t. Algoritam grube sile poredi date niske s i t karakter po karakter i složenosti je $O(\min\{n_1, n_2\})$, gde su n_1 i n_2 dužine niski s i t redom. Postavlja se pitanje da li postoji efikasniji algoritam za poređenje niski s i t. Svaku od niski s i t možemo konvertovati u celobrojnu vrednost i umesto niski uporediti dobijene celobrojne vrednosti. Konverziju vršimo pomoću *heš funkcije* (eng. hash function) h, a dobijene cele brojeve $h(s)$ i $h(t)$ nazivamo *heš vrednostima* (eng. hash values), ili skraćeno *heševima* niski s i t. Heš funkcija zadovoljava naredni uslov: ako su dve niske s i t jednake, i njihove heš vrednosti $h(s)$ i $h(t)$ moraju biti jednake. Naime, heš funkcija je uvek deterministička – za jedan ulaz uvek daje jedan isti izlaz. Obratimo pažnju da obratno ne mora da važi: naime, ako je $h(s) = h(t)$ ne mora nužno da važi $s = t$. Na primer, ako bismo hteli da vrednost heš funkcije bude jedinstvena za svaku nisku dužine do 15 karaktera koja se sastoji samo od malih slova engleske abecede, heš vrednosti niski ne bi stale u opseg 64-bitnih celih brojeva što je najširi primitivni celobrojni tip podataka (a svakako nam nije cilj da razmatramo cele brojeve sa proizvoljno mnogo cifara jer bi poređenje takvih brojeva bilo složenosti $O(n)$, gde je n broj cifara razmatranih brojeva). Upoređivanje niski svođenjem na upoređivanje njihovih heš vrednosti je složenosti $O(1)$, međutim ne treba zanemariti činjenicu da je samo heširanje niski s i t, kao što ćemo uskoro videti, složenosti $O(n_1 + n_2)$.

Dakle, cilj heširanja niski je preslikati skup niski u vrednosti iz fiksiranog opsega $[0, m)$. Pritom je, naravno, poželjno da verovatnoća da heš vrednosti dve različite niske budu jednake, odnosno da dođe do *kolizije* (eng. collision), bude što je moguće manja. U velikom broju slučajeva mogućnost dolaska do kolizije se ignoriše, čime se potencijalno narušava korektnost algoritma. Moguće je, pak, da se u slučajevima kada se dobiju jednake heš vrednosti dve niske ispita jednakost ove dve niske karakter po karakter, čime se potencijalno žrtvuje efikasnost algoritma. Međutim, očekuje se da do ove situacije neće često dolaziti. Odabir jedne od ove dve opcije zavisi od konkretne primene.

Definisanje heš funkcije

Jasno je da heš funkcija treba da zavisi od multiskupa karaktera koji se javljaju u niski i od redosleda karaktera u niski. Niske `maja` i `mara` bi trebalo da imaju različite heš vrednosti, kao i niske `maja` i `jama`. Dobar i široko rasprostranjen način definisanja heš funkcije niske s dužine n je korišćenjem *polinomijalne heš funkcije* (eng. polynomial rolling hash function):

$$\begin{aligned} h(s) &= (s[0] \cdot p^{n-1} + s[1] \cdot p^{n-2} + \ldots + s[n-1]) \bmod m \\ &= (\sum_{i=0}^{n-1} s[i] \cdot p^{n-i-1}) \bmod m \end{aligned} \quad (1)$$

pri čemu je $s[i]$ celobrojni kod i-tog karaktera niske s, a p i m su neki unapred odabrani, pozitivni brojevi. Pažljiv odabir parametara p i m je važan da bi se osigurala dobra svojstva heš funkcije. Često se za p bira prvi veći prost broj od broja karaktera u ulaznoj azbuci. Na primer, ako se niske sastoje samo od malih slova engleske abecede onda je dobar izbor $p = 31$ (mada se nekada u praksi za vrednost parametra p biraju i veliki prosti brojevi). Naime, pokazuje se da kada je vrednost parametra p manja od veličine azbuke, onda je `p` mogući karakter niske i lako je pronaći dve niske već dužine 2 kod kojih se javlja kolizija (npr. niske `10` i `0p`). Definicija polinomijalne heš funkcije odgovara predstavljanju broja s u brojevnom sistemu sa osnovom p. Poželjno je, takođe, da vrednost parametra m bude neki veliki broj jer je verovatnoća da dve slučajno odabrane niske imaju istu heš vrednost približno jednaka $1/m$. Ipak, izbegavaćemo prevelike vrednosti za parametar m jer je pogodno da vrednosti $m \cdot m$ i $p \cdot m$ ne izazivaju prekoračenje (što ćemo obrazložiti nešto kasnije). Kao i za parametar p, i za parametar m je pogodno birati neki prost broj. Prosti brojevi se koriste da se ne bi dešavao prevelik broj kolizija kada populacija koja se hešira ispoljava određena matematička svojstva, na primer kada su sve vrednosti umnošci istog broja, recimo parni brojevi. U narednim primerima za vrednost parametra m biraćemo vrednost $10^9 + 9$ koja je manja od 2^{30} te vrednost $m \cdot m$ staje u 64-bitni ceo broj. Primetimo i sledeće: kada izračunavanja ne bismo vršili po modulu m, odnosno ako u definiciji polinomijalne heš funkcije ne bi figurisala vrednost m, onda bi se operacije vršile po modulu broja podržanih vrednosti odgovarajućeg celobrojnog tipa (dakle po modulu 2^{32} ili 2^{64}).

Umesto heš funkcije zadate formulom (1) nekada se za heširanje koristi i naredna heš funkcija:

$$\begin{aligned} h'(s) &= (s[0] + s[1] \cdot p + \ldots + s[n-1] \cdot p^{n-1}) \bmod m \\ &= (\sum_{i=0}^{n-1} s[i] \cdot p^{i}) \bmod m \end{aligned} \quad (2)$$

kod koje je, nasuprot prethodno definisanoj heš funkciji, najveći stepen broja p uz prvi karakter niske s, a najmanji uz poslednji karakter niske.

Polinomijalna heš funkcija ima svojstvo da joj se vrednost može lako ažurirati kada se simboli dodaju ili uklanjaju sa krajeva niske (odatle potiče termin `rolling` u engleskom nazivu).

Ako bismo za $m = 10^9 + 9$ razmatrali skup niski sastavljenih od malih slova engleske abecede, takvih da je dužina niske manja ili jednaka 7, ukupan broj ovakvih niski bi iznosio:

$$26 + 26^2 + 26^3 + 26^4 + 26^5 + 26^6 + 26^7 = 8353082582 > 10^9 + 9 = m$$

U ovoj situaciji garantovana je pojava kolizije.

Problem: Napisati program kojim se računa heš vrednost niske s dužine n koja se sastoji samo od malih slova engleske abecede.

Prva varijanta Ukoliko se za heširanje niski koristi heš funkcija zadata jednačinom (1), izračunavanje heš vrednosti se sprovodi u duhu Hornerove šeme i linearne je složenosti u funkciji dužine niske. Svaki karakter niske s potrebno je konvertovati u ceo broj i u narednom algoritmu se koristi konverzija: $a \to 1, b \to 2, \ldots, z \to 26$. Konverzija $a \to 0$ nije pogodna jer bi onda vrednosti heš funkcije svih niski `a`, `aa`, `aaa`, `aaaa`, ... bile jednake 0. Slično, dve niske od kojih se jedna dobija od druge dodavanjem jednog ili većeg broja karaktera a na početak, na primer niske `na` i `ana` imale bi istu vrednost heš funkcije.

```
long long izracunajHeshVrednost(const string &s){
  int p = 31;
  int m = 1e9 + 9;
  long long h = 0;

  // vrednost hesh funkcije niske s racunamo u duhu Hornerove sheme
  for (int i = 0; i < s.size(); i++){
    h = (h * p + (s[i] - 'a' + 1)) % m;
  }
  return h;
}

int main(){
  vector<string> reci = {"ana", "a", "algoritam"};
  for (string s : reci){
    cout << "Hesh vrednost niske " << s << " je: "
         << izracunajHeshVrednost(s) << endl;
  }
  return 0;
}
```

Da bi se račun sveo na manje brojeve, u funkciji za računanje heš vrednosti niske s dužine n međurezultati su računati po modulu m, odnosno razmatran je niz međuvrednosti h_i za $i = 0, 1, \ldots, n$, pri čemu je cilj izračunati vrednost h_n:

$$\begin{aligned} h_0 &= 0 \\ h_{i+1} &= (h_i \cdot p + s[i]) \bmod m, \quad 0 \leq i < n \end{aligned}$$

Računanje međurezultata po modulu m matematički je korektno na osnovu pravila modularne aritmetike. Računanje heš vrednosti se vrši u stilu Hornerove šeme i vremenske složenosti je $O(n)$, gde je n dužina niske koja se hešira.

Primetimo da smo se u formuli za računanje vrednosti h_{i+1} koristili to da iz činjenice da je $h_i < m$ i da vrednost $p \cdot m$ ne izaziva prekoračenje važi da ni vrednost h_{i+1} neće izazivati prekoračenje.

Druga varijanta Ukoliko bismo za heširanje niski koristi heš funkciju zadatu jednačinom (2), onda bismo razmatrali niz h'_i za $i = n, n-1, \ldots, 0$, pri čemu je cilj izračunati vrednost h'_0:

$$\begin{aligned} h'_n &= 0 \\ h'_i &= (h'_{i+1} \cdot p + s[i]) \bmod m, \quad 0 \leq i < n \end{aligned}$$

```
long long izracunajHeshVrednost(const string &s){
  int p = 31;
  int m = 1e9 + 9;
  long long h = 0;

  // vrednost hesh funkcije niske s racunamo u duhu Hornerove sheme
  // razmatranjem karaktera niske u obratnom poretku
  for (int i = s.size() - 1; i >= 0; i--){
    h = (h * p + (s[i] - 'a' + 1)) % m;
  }
  return h;
}
```

Identifikovanje duplikata

Razmotrimo jedan problem kod koga se tehnikom heširanja niski dobija efikasniji algoritam.

Problem: Dato je n niski $s_1, s_2, \ldots, s_n$ od kojih je svaka dužine maksimalno m. Pronaći sve duplikate među niskama i podeliti ih na grupe jednakih niski.

Primer: Ako se sa ulaza učitaju redom niske `ana`, `a`, `dvorana`, `ana`, `banana`, `ana`, `kopakabana` i `banana`, očekujemo da dobijemo 5 različitih grupa: prva se sastoji od tri pojavljivanja niske `ana`, druga od jednog pojavljivanja niske `a`, treća grupa od jednog pojavljivanja niske `dvorana`, četvrta od dva pojavljivanja niske `banana` i peta od jednog pojavljivanja niske `kopakabana`.

Direktan pristup bi poredio svaku nisku sa svakom drugom: poređenje dve niske maksimalne dužine m karakter po karakter je složenosti $O(m)$, a ukupan broj poređenja niski je reda $O(n^2)$, te bi ukupna složenost odgovarajućeg algoritma bila $O(n^2m)$.

Efikasnije rešenje dobija se sortiranjem svih niski i traženjem duplikata u sortiranom nizu. Sortiranje niski bi uključivalo $O(n \log n)$ upoređivanja niski, a svako poređenje niski bilo bi u najgorem slučaju složenosti $O(m)$ (mada je očekivano da ono često bude efikasnije, odnosno da se razlika pronađe na prvih nekoliko karaktera), te bi ukupna složenost sortiranja niski iznosila $O(nm \log n)$. Dodatno, prolazak kroz skup niski u sortiranom redosledu i identifikovanje istih niski je složenosti $O(nm)$, te je ukupna složenost ovog algoritma $O(nm \log n)$.

Pristup zasnovan na heširanju niski redukuje vreme poređenja dve niske na $O(1)$. Na taj način dobijamo algoritam složenosti $O(nm + n \log n)$, gde složenost $O(nm)$ potiče od računanja heš vrednosti svih niski, a $O(n \log n)$ od sortiranja niski na osnovu njihovih heš vrednosti. Dodatno, prolazak kroz heš vrednosti niski u sortiranom poretku i identifikovanje istih niski je složenosti $O(n)$.

```
void grupisiIsteNiske(const vector<string> &niske){
  // broj niski
  int n = niske.size();
  // vektor parova hesh vrednosti i pozicije niske u polaznom nizu
  vector<pair<long long, int>> h(n);
  // izracunavamo hesh vrednost svake niske;
  // uz hesh vrednost niske cuvamo i njen indeks u polaznom nizu
  for(int i = 0; i < n; i++)
    h[i] = {izracunajHeshVrednost(niske[i]),i};

  // sortiramo niz hesh vrednosti
  sort(h.begin(),h.end());

  // svaki element vektora sadrzi niz indeksa niski
  // koje su medjusobno jednake
  vector<vector<int>> grupe;

  // prolazimo skupom svih niski u sortiranom poretku
  for(int i = 0; i < n; i++){
    // ukoliko se radi o prvoj niski u sortiranom poretku
    // ili o niski koja nije jednaka prethodnoj u sortiranom poretku
    // onda je potrebna nova grupa
    if (i == 0 || h[i].first != h[i-1].first){
      vector<int> novaGrupa;
      grupe.push_back(novaGrupa);
    }
    // u poslednju (tekucu) grupu dodajemo na kraj indeks niske
    // koja ima tekucu hesh vrednost
```

```
    grupe.back().push_back(h[i].second);
  }

  // stampamo niske po grupama
  for (int i = 0; i < grupe.size(); i++){
    cout << "Grupa broj " << i << endl;
    for (int j = 0; j < grupe[i].size(); j++)
      cout << niske[grupe[i][j]] << " ";
    cout << endl;
  }
}

int main(){
  vector<string> reci = { "ana", "a", "dvorana", "ana", "banana",
                          "ana", "kopakabana", "banana"};
  grupisiIsteNiske(reci);
  return 0;
}
```

Računanje heš vrednosti segmenata niske

Problem: Data je niska s dužine n. Napisati algoritam kojim se za date parove indeksa i i j, $0 \leq i \leq j < n$ izračunavaju heš vrednosti segmenta $s[i..j]$ polazne niske (podniske susednih karaktera polazne niske s od pozicije i zaključno sa pozicijom j).

Primer: Za nisku s koja je jednaka `ananas` i za vrednosti $i = 1$ i $j = 4$, potrebno je izračunati heš vrednost niske `nana`.

Prva varijanta Razmotrimo najpre prvu varijantu polinomijalne heš funkcije koja je zadata formulom (1). Prema definiciji heš funkcije važi:

$$\begin{aligned} h(s[i..j]) &= (s[i] \cdot p^{j-i} + s[i+1] \cdot p^{j-i-1} + \ldots + s[j]) \bmod m \\ &= \left(\sum_{k=i}^{j} s[k] \cdot p^{j-k}\right) \bmod m \end{aligned} \tag{3}$$

Ova vrednost može se direktno izračunati u vremenskoj složenosti $O(j-i)$, odnosno u najgorem slučaju u složenosti $O(n)$, gde je n dužina niske s. Pretpostavimo da je veliki broj puta, npr. l puta, potrebno računati heš vrednosti različitih segmenata date niske. Naivni pristup bi ovaj problem rešio u vremenu $O(ln)$ u najgorem slučaju. Da li možemo efikasnije rešiti ovaj problem?

Zapišimo čemu su jednake heš vrednosti prefiksa niske s dužina redom $j+1$ i i:

$$\begin{aligned} h(s[0..j]) &= (s[0] \cdot p^{j} + s[1] \cdot p^{j-1} + \ldots + s[j]) \bmod m \\ h(s[0..i-1]) &= (s[0] \cdot p^{i-1} + s[1] \cdot p^{i-2} + \ldots + s[i-1]) \bmod m \end{aligned}$$

Primetimo da je:

$$
\begin{aligned}
h(s[0..j]) - p^{j-(i-1)} \cdot h(s[0..i-1]) &= (s[i] \cdot p^{j-i} + s[i+1] \cdot p^{j-(i+1)} + \ldots + s[j]) \bmod m \\
&= h(s[i..j]) \bmod m \qquad (4)
\end{aligned}
$$

Pošto i promenljive i i j uzimaju vrednosti od 0 do $n-1$, izraz $j-(i-1)$ uzima vrednosti od 1 do n, te je u stvari potrebno izračunati vrednost p^k za sve vrednosti $0 \leq k < n$. Dakle, ako za datu nisku znamo heš vrednosti svih njenih prefiksa i vrednosti p^k za $k = 0, 1, \ldots, n-1$, onda na osnovu jednačine (4) u složenosti $O(1)$ možemo izračunati heš vrednost proizvoljnog segmenta te niske. Heš vrednost svih prefiksa niske s i vrednosti p^k mod m za svako $k = 1, 2, \ldots, n$ mogu se izračunati u vremenskoj složenosti $O(n)$, pa je ukupna složenost računanja heš vrednosti l različitih segmenata $O(n+l)$.

```
int p = 31;
long long m = 1e9 + 9;

// stepeni broja p i njihovi inverzi po modulu m
vector<long long> pStepen, invpStepen;
// hesh vrednosti svih prefiksa niske
vector<long long> heshPr;

// funkcija za mnozenje po modulu m
long long puta_mod(long long a, long long b, long long m){
  return ((a % m) * (b % m)) % m;
}

// funkcija za brzo stepenovanje po modulu m
long long stepen_mod(long long a, long long b, long long m) {
  if (b == 0)
    return 1;
  long long rez = stepen_mod(a, b / 2, m);
  rez = puta_mod(rez, rez, m);
  if (b % 2)
    return puta_mod(rez, a, m);
  else
    return rez;
}

// racunanje inverza koriscenjem male Fermaove teoreme
long long modInverz(long long a, long long m){
  return stepen_mod(a, m - 2, m);
}

void obradiStepeneBrojaP(int n){
  pStepen.resize(n);
```

```
  invpStepen.resize(n);

  // racunamo stepene broja p i inverze stepena broja p
  pStepen[0] = 1;
  invpStepen[0] = 1;
  for(int i = 1; i < n; i++){
    pStepen[i] = (pStepen[i-1] * p) % m;
    invpStepen[i] = modInverz(pStepen[i], m);
  }
}

void izracunajHeshevePrefiksa(string s){
  int n = s.size();
  heshPr.resize(n+1,0);
  // racunamo hesheve svih prefiksa datog stringa
  for(int i = 0; i < n; i++)
    heshPr[i+1] = (heshPr[i]*p + (s[i]-'a'+1)) % m;
}

long long heshVrSegmenta(string const& s, int i, int j){
  int n = s.size();
  // hesh vrednost segmenta racunamo preko hesh vrednosti prefiksa
  long long hesh = (heshPr[j+1] -
      puta_mod(heshPr[i], stepen_mod(p,j-i+1,m), m) + m) % m;
  return hesh;
}

int main(){
  string rec = "banana";
  int n = rec.size();

  obradiStepeneBrojaP(n);
  izracunajHeshevePrefiksa(rec);

  int i, j;
  int k, l;
  cout << "Unesi indeks pocetka i kraja segmenta" << endl;
  cin >> i >> j;
  long long hesh = heshVrSegmenta(rec,i,j);
  cout << "Hesh vrednost segmenta " <<  rec.substr(i,j-i+1)
      << " je: " << hesh << endl;
  return 0;
}
```

Primetimo da se prilikom računanja heš vrednosti segmenta izvršava množenje heš vrednost prefiksa odgovarajućim stepenom broja p po modulu m. Da ovaj proizvod ne bi doveo do prekoračenja, potreban je uslov da $m \cdot m$ ne izaziva prekoračenje.

Druga varijanta Razmotrimo sada korišćenje druge predložene heš funkcije. Prema njoj, heš vrednost segmenta $s[i,j]$ jednaka je:

$$\begin{aligned} h'(s[i..j]) &= (s[i] + s[i+1] \cdot p + \ldots + s[j] \cdot p^{j-i}) \bmod m \\ &= \Big(\sum_{k=i}^{j} s[k] \cdot p^{k-i}\Big) \bmod m \qquad (5) \end{aligned}$$

Primetimo da ukoliko obe strane jednakosti (5) pomnožimo sa p^i dobijamo:

$$\begin{aligned} h'(s[i..j]) \cdot p^i &= \Big(\sum_{k=i}^{j} s[k] \cdot p^{k}\Big) \bmod m \\ &= (h'(s[0..j]) - h'(s[0..i-1])) \bmod m \qquad (6) \end{aligned}$$

Stoga ako znamo heš vrednost svih prefiksa date niske s, možemo na osnovu jednačine (6) izračunati heš vrednost proizvoljnog segmenta niske s. Ova tehnika odgovara računanju zbira elemenata proizvoljnog segmenta niza brojeva kao razlike odgovarajućih prefiksnih suma. Niz heš vrednosti svih prefiksa niske s računamo inkrementalno u vremenskoj složenosti $O(n)$, gde je n dužina niske s.

Primetimo da je za računanje vrednosti heš funkcije nekog segmenta prema formuli (6) potrebno izvršiti deljenje izraza $h(s[0..j]) - h(s[0..i-1])$ vrednošću p^i po modulu m. Za to je potrebno odrediti multiplikativni inverz broja p^i po modulu m, odnosno broj x tako da važi $p^i \cdot x \equiv 1 \pmod{m}$, a zatim izvršiti množenje izraza $h(s[0..j]) - h(s[0..i-1])$ brojem x. Može se za svako p^i gde je $0 \leq i < n$ unapred izračunati vrednost mutliplikativnog inverza po modulu m (podsetimo se da je računanje multiplikativnog inverza broja po modulu m složenosti $O(\log m)$).

Pošto sva izračunavanja vršimo po modulu m, a s obzirom na to da smo na početku pretpostavili da su brojevi m i p prosti, po maloj Fermaovoj teoremi važi da je

$$a^{m-1} \equiv 1 \pmod{m},$$

odnosno multiplikativni inverz po modulu m broja a koji je uzajamno prost sa p je $a^{m-2} \bmod m$. Ako u ovu jednačinu umesto a uvrstimo vrednost p^i, dobijamo multiplikativni inverz za proizvoljno p^i po modulu m. Dakle, ako su unapred poznate heš vrednosti svih prefiksa niske s i vrednosti multiplikativnog inverza broja $p^i, 1 \leq i < n$ po modulu m, izračunavanje heš vrednosti proizvoljnog segmenta polazne niske može se na osnovu formule (6) izvršiti u vremenu $O(1)$. Primetimo da je faza preprociranja koja se sastoji od računanja heš vrednosti svih prefiksa niske i vrednosti inverza broja p^i za svako i vremenske složenosti $O(n \cdot \log m)$.

```
void izracunajHeshevePrefiksa(string s){
  int n = s.size();
  heshPr.resize(n+1,0);
  // racunamo hesheve svih prefiksa datog stringa
  for(int i = 0; i < n; i++)
    heshPr[i+1] = (heshPr[i] + (s[i]-'a'+1) * pStepen[i]) % m;
}

long long heshVrSegmenta(string const& s, int i, int j){
  int n = s.size();
  // hesh vrednost segmenta racunamo preko hesh vrednosti prefiksa
  long long hesh = puta_mod((heshPr[j+1] - heshPr[i] + m) % m,
                            invpStepen[i], m);
  return hesh;
}
```

Broj različitih segmenata niske

Problem: Data je niska s dužine n, koja se sastoji samo od malih slova engleske abecede. Izračunati broj različitih segmenata ove niske algoritmom složenosti $O(n^2 \log n)$.

Primer: Ako je niska s jednaka `ananas`, onda postoji tri različita segmenta dužine 1: `a`, `n` i `s`, tri različita segmenta dužine 2: `an`, `na` i `as`, tri različita segmenta dužine 3: `ana`, `nan` i `nas`, tri različita segmenta dužine 4: `anan`, `nana`, `anas`, dva različita segmenta dužine 5: `anana`, `nanas` i jedan segment dužine 6: `ananas`, te niska s ima ukupno $3+3+3+3+2+1 = 15$ različitih segmenata.

Prilikom rešavanja ovog problema, potrebno je porediti samo segmente istih dužina jer oni mogu biti međusobno jednaki.

Ukoliko koristimo heš funkciju zadatu formulom (2), umesto da računamo i poredimo tačne heš vrednosti dva segmenta iste dužine korišćenjem modularnih multiplikativnih inverza, dovoljno je izračunati heš vrednosti segmenata pomnožene nekim (istim) stepenom broja p. Pretpostavimo da imamo izračunate heš vrednosti dva segmenta x i y, jednog pomnoženog sa p^i, a drugog sa p^j. Bez narušavanja opštosti možemo pretpostaviti da je $i < j$; onda heš vrednost segmenta x množimo sa p^{j-i}, čime dobijamo obe heš vrednosti pomnožene istim stepenom broja p. Heš vrednosti segmenata x i y možemo uporediti poređenjem vrednosti $h(x) \cdot p^i$ i $h(y) \cdot p^i$, bez potrebe za računanjem modularnog multiplikativnog inverza stepena broja p.

Primer: Razmotrimo na primer segmente $s[2..4]$ i $s[5..7]$ dužine 3 niske s koja je dužine 10. Iz jednačine (6) sledi:

$$
\begin{aligned}
h_1 &= h(s[2..4]) \cdot p^2 = h(s[0..4]) - h(s[0..1]) \bmod m \\
h_2 &= h(s[5..7]) \cdot p^5 = h(s[0..7]) - h(s[0..4]) \bmod m
\end{aligned}
$$

Umesto da računamo multiplikativne inverze brojeva p^2 i p^5, da bismo dobili tačne heš vrednosti segmenata $s[2..4]$ i $s[5..7]$, možemo porediti vrednosti $h_1 \cdot p^{5-2}$ i h_2.

Dakle, prolazićemo redom kroz sve moguće dužine $l = 1, 2, \ldots, n$ segmenata niske s i konstruisati niz heš vrednosti svih segmenata dužine l koji su pomnoženi nekim (istim, maksimalnim) stepenom broja p. Broj različitih elemenata u nizu jednak je zbiru broja različitih segmenata dužine l u niski, za svako moguće l.

```
int izbrojRazliciteSegmente(const string &s){
  int n = s.size();
  int p = 31;
  int m = 1e9 + 9;

  // racunamo stepene broja p po modulu m
  vector<long long> pStepen(n);
  pStepen[0] = 1;
  for(int i = 1; i < n; i++)
    pStepen[i] = (pStepen[i-1] * p) % m;

  // racunamo hesh vrednosti svih prefiksa date niske
  vector<long long> h(n+1,0);
  for(int i = 0; i < n; i++)
    h[i+1] = (h[i] + (s[i] - 'a' + 1) * pStepen[i]) % m;

  // broj razlicitih segmenata
  int brSegmenata = 0;
  // za svaku mogucu duzinu segmenta
  for(int l = 1; l <= n; l++){
    // skup hesh vrednosti segmenata duzine l pomnozenih sa p^{n-1}
    set<long long> heshevi_duzine_l;
    // prolazimo kroz sve segmente duzine l
    for (int i = 0; i <= n - l; i++){
      // hesh vrednost segmenta racunamo
      // kao razliku hes vrednosti odgovarajucih prefiksa
      long long hTekuce = (h[i+l] - h[i] + m) % m;
      // racunamo hes vrednost segmenta pomnozenu sa p^{n-1} po modulu m
      // tako sto je mnozimo sa p^{n-i-1}
      hTekuce = (hTekuce * pStepen[n - i - 1]) % m;
      // hesh vrednost dodajemo u skup,
      // isti segmenti imace istu hesh vrednost pa se nece dva puta racunati
      heshevi_duzine_l.insert(hTekuce);
    }
    brSegmenata += heshevi_duzine_l.size();
  }
  return brSegmenata;
}
```

```
int main(){
  string rec = "banana";
  int broj = izbrojRazliciteSegmente(rec);
  cout << "Broj razlicitih segmenata reci " << rec
       << " je: " << broj << endl;
  return 0;
}
```

Primetimo da se prilikom računanja proizvoda heš vrednosti segmenta i vrednosti p^{n-1} po modulu m množe dve vrednosti iz opsega $[0, m)$ te da ne bi bilo prekoračenja prilikom računanja ovog proizvoda vrednost parametra m biramo tako da $m \cdot m$ staje u 64-bitni ceo broj.

Operacije za rad sa uređenim skupom su složenosti $O(\log n)$, a različitih segmenata ima ukupno $O(n^2)$, te složenost ovog algoritma iznosi $O(n^2 \log n)$, gde je n dužina date niske.

Istaknimo i to da se u ovom kontekstu, kada radimo sa različitim segmentima niske, isplati uložiti dodatno vreme za računanje heš vrednosti svih prefiksa date niske, jer ćemo izračunate heš vrednosti veći broj puta koristiti.

Za vežbu ostavljamo naredni problem.

Problem: Za datu nisku s izračunati najdužu nisku koja se javlja bar dva puta kao segment niske s.

Obrada kolizija

Prethodno definisana polinomijalna heš funkcija je često dovoljno dobra i prilikom testiranja ne dolazi do kolizije. Na primer, za odabir vrednosti parametra $m = 10^9 + 9$ verovatnoća da dođe do kolizije je samo $1/m = 10^{-9}$, pod pretpostavkom da su sve heš vrednosti jednako verovatne. Međutim, ukoliko nisku s poredimo sa 10^6 različitih niski, verovatnoća da dođe do bar jedne kolizije iznosi oko $10^6 \cdot 10^{-9} = 10^{-3}$, dok ako poredimo 10^6 niski međusobno verovatnoća da dođe do bar jedne kolizije je 1. Poslednji od pomenutih scenarija poznat je pod nazivom *rođendanski paradoks* (eng. birthday paradox) i odnosi se na sledeći kontekst: ako se u jednoj sobi nalazi n osoba, verovatnoća da neke dve osobe imaju rođendan istog dana za $n = 23$ iznosi oko 50%, dok za $n = 70$ ona iznosi čak 99.9%.

Postoji jednostavni trik kojim se može smanjiti verovatnoća da do kolizije dođe – računanjem vrednosti dve različite heš funkcije (šta više, može se iskoristiti i ista heš funkcija za različite vrednosti parametara p i/ili m). Ako je vrednost parametra m oko 10^9 za obe heš funkcije, onda je ovo uporedivo sa korišćenjem jedne heš funkcije za vrednost parametra m koja je približno jednaka 10^{18}. Sada, ako poredimo 10^6 niski međusobno, verovatnoća da dođe do kolizije smanjiće se na $(10^6 \cdot 10^6) \cdot 10^{-18} = 10^{-6}$.

U jeziku C++ na raspolaganju nam je i struktura `hash` koja se može upotrebiti za računanje heš vrednosti niski (ali i heš vrednosti ostalih osnovnih tipova podataka).

```
hash<string> h;
cout << h("abrakadabra") << endl;
```

z-algoritam

z-niz (eng. z-array, z-function) niske s dužine n je niz dužine n koji na poziciji $k = 0, 1, \ldots, n-1$ sadrži dužinu najdužeg segmenta niske s koji počinje na poziciji k i prefiks je niske s. Dakle, ako važi $z[k] = p$ onda se segment $s[0..p-1]$ poklapa sa segmentom $s[k..k+p-1]$, i karakteri $s[p]$ i $s[k+p]$ su različiti ili je pak niska s dužine $k+p$. Segment unutar niske koji se preklapa sa nekim prefiksom date niske nazivamo i *z-kutija* (eng. z-box). Primer z-kutije ilustrovan je na slici 1. Na poziciji i ne postoji z-kutija ako je vrednost $z[i] = 0$.

```
|0|1|2|3|4|5|6|7|8|9|10|
|A|B|C|X|X|X|A|B|Y|Y| Y|
            |   |
             ---
           z-kutija
```

Slika 1: Ilustracija z-kutije koja počinje na poziciji 6 i završava se na poziciji 7.

Mnogi problemi nad niskama mogu se efikasno rešiti korišćenjem z-niza; na primer, ispitivanje da lli se jedna niska javlja unutar druge, kompresija niske (u smislu određivanja najkraće niske takve da se polazna niska može predstaviti nadovezivanjem te niske određen broj puta), itd. Vrednost niza z na poziciji 0 jednaka je dužini niske jer je kompletna niska uvek prefiks same sebe, međutim ta vrednost ovde neće biti od značaja.

Primer: z-niz niske `ACBACDACBACBACDA` prikazan je na slici 2. 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.

```
|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|
```

Slika 2: Ilustracija z-niza.

Ostaje pitanje kako što efikasnije konstruisati z-niz za datu nisku.

Algoritam grube sile

Direktno rešenje bi se sastojalo u tome da se za svaki indeks iznova traži pozicija najdužeg poklapajućeg prefiksa niske koji počinje na tekućoj poziciji u niski.

```
vector<int> izracunajZNizTrivijalno(string s) {
  int n = s.size();
  // inicijalizujemo sve vrednosti z-niza na 0
  vector<int> z(n,0);

  for (int i = 1; i < n; i++) {
    // sve dok ne izađemo iz opsega niske
    // i odgovarajući karakteri se poklapaju
    while (i + z[i] < n && s[z[i]] == s[i+z[i]]){
      // inkrementiramo vrednost z-niza na odgovarajucoj poziciji
      z[i]++;
    }
  }
  return z;
}
```

Ovo rešenje bi imalo dve ugnježdene petlje i bilo bi složenosti $O(n^2)$. Na primer, ako razmatramo nisku `AAAAAAAAAAAAA` algoritam grube sile za računanje z-niza bi izvršavao $\Theta(n^2)$ koraka. U praksi, očekuje se da se na razlike odgovarajućih karaktera naiđe dosta brže.

z-algoritam

z-algoritam (eng. z-algorithm) je algoritam za konstrukciju z-niza složenosti $O(n)$. Dobio je ime po strukturi podataka koja se njome konstruiše, tj. po z-nizu. Do njega je prvi došao Gustav Fridrih Hartman 1975. godine, ali je svoju popularnost stekao tek kasnije, nakon što su Martin Eskardo i Rajnard Vilhelm predstavili efikasniju varijantu algoritma. Ideja z-algoritma jeste da se vrednosti z-niza izračunavaju iterativno, sleva nadesno, na osnovu prethodno izračunatih vrednosti z-niza.

U algoritmu se održava najdesniji opseg indeksa $[l, d]$ za koji važi da se segment $s[l..d]$ poklapa sa nekim prefiksom niske s (najdesnija z-kutija). Činjenicu da je prefiks $s[0..d-l]$ jednak segmentu $s[l..d]$ koristimo za računanje vrednosti z-niza na pozicijama $l+1, l+2, \ldots, d$.

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 niske s počev od pozicija i i 0;

- $l < i \leq d$ – 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). Naime segmenti $s[l..d]$ i $s[0..d-l]$ se poklapaju, pa pošto je $l < i \leq d$ poklapaju se i segmenti $s[i..d]$ i

$s[i-l..d-l]$ pa prilikom računanja vrednosti $z[i]$ možemo krenuti od vrednosti $z[i-l]$ koja je već izračunata (slika 3). Međutim, vrednost $z[i-l]$ u nekim slučajevima može biti prevelika, jer kada se doda poziciji i može da prevaziđe vrednost indeksa d, a to nije dobro jer mi ne znamo ništa o karakterima niske s nakon pozicije d. Recimo, u primeru prikazanom na slici 4 vrednost $z[i]$ ne bi bilo ispravno postaviti na $z[i-l] = 3$ jer bi nas to izvelo van granica opsega $[l, d]$. Dakle, maksimalna dužina poklopljenog dela može biti jednaka broju karaktera od tekuće pozicije i do poslednje pregledane pozicije d, a to je $d - i + 1$. Stoga se početna vrednost $z[i]$ postavlja na

$$z_0[i] = \min\{d - i + 1, z[i-l]\}.$$

Nakon toga utvrđujemo da li se vrednost $z[i]$ može povećati pokretanjem trivijalnog algoritma od pozicije $i + z_0[i]$.

```
 _______________________    _____ z-kutija ________
|                       |  |                       |
| 0 |  |  |i-l|  |  |d-l|  | l |  |  | i |  |  | d |  |
          |_____________|            |_____________|
```

Slika 3: Poklapanje segmenata $[0, d-l]$ i $[l, d]$ povlači i poklapanje segmenata $[i-l, d-l]$ i $[i, d]$ te za računanje vrednosti $z[i]$ možemo iskoristiti vrednost $z[i-l]$.

```
 ___________________________      ________z-kutija_______
|                           |     |                       |
| 0 | 1 |   ... |i-l|   |d-l|     | l | ...       | i | d |
| A | B | C | X | A | B | C | ... | A | B | C | X | A | B | F |
|___________|   |___________|                     |___________|
```

Slika 4: Ilustracija slučaja kada vrednost $i + z[i-l]$ prevazilazi desni kraj najdesnije z-kutije.

Dakle, algoritam razmatra dva slučaja koji se razlikuju samo po inicijalizaciji vrednosti $z[i]$ nakon čega se postupak svodi na primenu trivijalnog algoritma. Ukoliko dođe do poklapanja dela niske počev od pozicije i sa nekim prefiksom date niske i ako je desna granica preklopljenog segmenta veća od prethodne vrednosti desne granice ($i + z[i] - 1 > d$) ažuriramo granice najdesnije z-kutije $[l, d]$.

Razmotrimo implementaciju z-algoritma.

```
// funkcija koja izracunava sve elemente z-niza
vector<int> izracunajZNiz(const string &s) {
  int n = s.size();
  // inicijalizujemo sve vrednosti z-niza na 0
  vector<int> z(n,0);
  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 i+z[i]-1;
    // od pozicije i+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 najdesnije z-kutije
    // 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;
}
```

Primetimo da se `while` petlja može uspešno izvršiti samo kada je $d-i+1 < z[i-l]$, odnosno kada je postojalo poklapanje dela niske do kraja najdesnije z kutije sa nekim prefiksom niske.

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 $[l, d] = [3, 4]$ (slika 5).

Nakon toga, vrednost z-niza na poziciji $i = 4$ dobijamo na osnovu vrednosti

```
       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|?|?|?|?|?|?| ?| ?| ?| ?| ?| ?|
```

Slika 5: Računanje prve tri vrednosti z-niza.

$z[4] = z[i-l] = z[4-3] = z[1] = 0$. Vrednosti $z[5]$ i $z[6]$ dobijamo pokretanjem trivijalnog algoritma (jer za $i = 5$ i $i = 6$ i $d = 4$ važi $i > d$) i dobijamo $z[5] = 0$, a $z[6] = 5$. Pošto za $i = 6$ važi $i + z[i] - 1 = 6 + 5 - 1 = 10$, desna granica tekuće z-kutije je veća od prethodne vrednosti 4, pa tekući $[l, d]$ opseg najdesnije z-kutije postaje $[6, 10]$ (slika 6).

```
             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|?|?|?| ?| ?| ?| ?| ?| ?|
```

Slika 6: Računanje vrednosti z-niza na pozicijama $\{4, 5, 6\}$.

Nakon ovog koraka, možemo efikasno izračunati naredne vrednosti z-niza, 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 $z[7] = z[8] = 0$ (slika 7). Nakon toga, pošto je $z[3] = 2$ i $d - i + 1 = 10 - 9 + 1 = 2$, znamo da je $z[9] \geq 2$. Pošto nemamo informacije o karakterima niske nakon pozicije 10, poredimo segmente dalje 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|?| ?| ?| ?| ?| ?| ?|
```

Slika 7: Računanje vrednosti z-niza na pozicijama $\{7, 8\}$.

Ispostavlja se da je $z[9] = 7$. Pošto za $i = 9$ važi $i + z[i] - 1 = 9 + 7 - 1 = 15$, desna granica tekuće z-kutije je veća od prethodne vrednosti 10, pa je novi $[l, d]$ opseg najdesnije z-kutije jednak $[9, 15]$ (slika 8).

Nakon ovoga, sve preostale vrednosti z-niza mogu se odrediti na osnovu informacija koje se već nalaze u z-nizu (slika 9).

Pokažimo da je prikazani algoritam linearne vremenske složenosti, iako sadrži dve ugnježđene petlje. Naime, nakon svakog neuspešnog poređenja karaktera, tekuća iteracija `for` petlje se završava, a ukupan broj iteracija spoljašnje `for` petlje je n. Stoga ukupno možemo imati najviše n nepoklapanja karaktera. Svaki

```
                          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| ?| ?| ?| ?| ?| ?|
```

Slika 8: Računanje vrednosti na poziciji 9 z-niza i nove z-kutije.

```
                          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|
```

Slika 9: Računanje poslednjih nekoliko elemenata z-niza.

karakter niske s koji se uspešno poklopi u unutrašnjoj `while` petlji se više nikada ne poredi, te je i broj uspešno poklopljenih karaktera maksimalno n. Dakle, u svim iteracijama spoljašnje `for` petlje ukupan broj izvršavanja unutrašnje `while` petlje je reda $O(n)$, te je ukupna složenost algoritma $O(n)$.

Formalno gledano, moglo bi se pokazati da svaka iteracija unutrašnje `while` petlje povećava desnu granicu segmenta d najdesnije z-kutije. Naime, svaka iteracija `while` petlje poklapa neki novi karakter niske, desno od granice najdesnije z-kutije, te će se vrednost promenljive d uvećati za onoliko koliki je broj iteracija `while` petlje. Pošto je maksimalna vrednost za d jednaka $n-1$, a početna je jednaka 0, dobijamo da se unutrašnja petlja u svim iteracijama spoljašnje petlje ukupno neće izvršiti više od $n-1$ puta. Odavde sledi da je složenost algoritma $O(n)$.

Često se prilikom rešavanja nekog problema nad niskama javlja izbor da li koristiti heširanje niski ili z-niz. Za razliku od heširanja, z-algoritam za konstrukciju z-niza ima zagarantovanu korektnost i ne postoji rizik da dođe do kolizije, ali je nešto teži za implementaciju. Međutim, postoje neki problemi koji se mogu rešiti isključivo tehnikom heširanja niski.

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 iz konačne azbuke. Za prvu od njih (koja je po pravilu duža) reći ćemo da je *tekst* (eng. text), a za drugu da je *uzorak* (eng. pattern). 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.

Primer: Razmotrimo primer teksta `abrakadabra` i uzorka `ra`. Uzorak se u tekstu javlja dva puta, počev od pozicije 2 i pozicije 9.

Tipična situacija u kojoj se nailazi na ovaj problem je kad se neka reč traži u tekstualnoj datoteci. Uopšteno, ovaj problem ima primenu u obradi prirodnih jezika – npr. u proveri pravopisa i gramatike, u analizi sentimenata i sl. Takođe, problem traženja uzorka u tekstu ima primenu u pretraživačima, u sistemima za preporučivanje, u analizi društvenih mreža i sl. Problem traženja uzorka u tekstu ima primenu i u molekularnoj biologiji, gde je često potrebno pronaći neke uzorke u okviru velikih molekula RNK ili DNK.

Naivni algoritam

Direktni algoritam za traženje uzorka u tekstu bi poredio uzorak P sa svim mogućim segmentima $t_k t_{k+1} \dots t_{k+m-1}$ teksta T dužine m, za $k = 0, 1, \dots, n - m + 1$. 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 karaktera, odnosno kada se desi $p_i \neq t_{k+i-1}$ za neko i, $0 \leq i \leq m - 1$ (slika 10).

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

Slika 10: Ilustracija algoritma grube sile za traženje uzorka u tekstu.

U drugom slučaju uzorak se "pomera" za jedan karakter udesno, odnosno nastavlja se sa proverom jednakosti karaktera p_0 uzorka sa karakterom t_{k+1} teksta. Broj upoređivanja karaktera je u najgorem slučaju reda mn (mada će se u praksi često mnogo brže naići na neslaganje uzorka i teksta), pa je složenost ovog algoritma $O(mn)$ u najgorem slučaju.

```
// 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){
      // bar jednom je uzorak pronadjen u tekstu
      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;
}
```

Primer: Razmotrimo primer teksta `aaaaaaaaaaaaa` i uzorka `aaaaab`. Naivni algoritam bi uspešno poklopio prvih $m-1$ karaktera uzorka sa prvih $m-1$ karaktera teksta i onda detektovao neslaganje na poslednjem karakteru uzorka. Nakon toga bi se uzorak pomerio za jedno mesto udesno u odnosu na tekst i ponovo bi se uspešno poklopilo prvih $m-1$ karaktera uzorka sa odgovarajućim karakterima teksta, dok bi se na poslednjem karakteru uzorka detektovala razlika. Identičan scenario bi se ponavljao sve do kraja izvršavanja algoritma. U ovom slučaju broj poređenja karaktera je reda mn.

Rabin-Karpov algoritam

Rabin i Karp su 1987. godine predložili algoritam koji rešava problem traženja uzorka u tekstu korišćenjem tehnike heširanja niski. Ideja algoritma je sledeća: izračunati heš vrednost uzorka P dužine m i svih segmenata teksta T dužine m i uporediti njihove vrednosti. Heš vrednost proizvoljnog segmenta teksta može se izračunati u vremenu $O(1)$ na osnovu heš vrednosti prefiksa teksta T i stepena broja p po modulu m. Izračunavanje heš vrednosti uzorka je složenosti $O(m)$, izračunavanje heš vrednosti svih prefiksa date niske je složenosti $O(n)$, dok je ukupna složenost poređenja heš vrednosti svih segmenata teksta T dužine m sa heš vrednošću uzorka $O(n)$. Stoga je ukupna složenost Rabin-Karpovog algoritma $O(n+m)$.

Razmotrimo Rabin-Karpov algoritam u slučaju kada je heš funkcija zadata formulom (1).

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

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

  // vektor pocetnog indeksa pojavljivanja uzorka u tekstu
  vector<int> pojave;
  for (int i = 0; i <= N - M; i++){
    // racunamo hesh vrednosti segmenta duzine M
    // koji pocinje na poziciji i
    long long hTekuce = (heshPr[i+M]
        - heshPr[i] * pStepen[M] + m) % m;
    // ako se poklapaju hesh vrednosti segmenta teksta i uzorka
    // pomnozene istim stepenom broja p, pamtimo indeks i
    if (hTekuce == heshUzorka)
      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] << " ";
  }
  else
    cout << "Uzorak se ne javlja u tekstu" << endl;
}
```

```
int main(){
  string tekst = "banana";
  string uzorak = "ana";
  rabinKarp(uzorak, tekst);
  return 0;
}
```

Algoritam zasnovan na z-nizu

Nekada se prilikom obrade niski konstruiše jedna duža niska koja se sastoji od većeg broja niski razdvojenih specijalnim karakterima. To je slučaj i kod algoritma traženja uzorka u tekstu zasnovanog na z-nizu. Naime, u ovom slučaju možemo konstruisati nisku $P\#T$, gde su uzorak P i tekst T razdvojeni specijalnim karakterom # koji se ne javlja u niskama P i T. z-niz niske $P\#T$ nam može ukazati na to gde se u tekstu T pojavljuje uzorak P jer će takve pozicije imati z-vrednost jednaku dužini uzorka P. Dobijene vrednosti z-niza na pozicijama 0 do m, gde je m dužina uzorka, nam nisu bitne jer se odnose na pozicije unutar uzorka, ali ih je potrebno izračunati zbog efikasnog sprovođenja z-algoritma.

Primer: Razmotrimo slučaj kada je `P = ra`, a `T = abrakadabra`. Formiramo z-niz niske $P\#T$ (slika 11). Primetimo da su vrednosti z-niza na pozicijama 5 i 12 jednake 2, odnosno dužini uzorka, što nam govori da se počev od ovih pozicija u tekstu T javlja uzorak P.

```
|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|
```

Slika 11: Algoritam traženja uzorka u tekstu zasnovan na z-nizu.

Složenost ovog algoritma je $O(m+n)$, gde je m dužina uzorka, a n dužina teksta, jer se algoritam svodi na konstrukciju z-niza (što smo videli da je linearne složenosti u odnosu na dužinu niza) i prolazak kroz njegove vrednosti.

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

  for (int i = 1; i < n; i++) {
    // ako je tekuca pozicija unutar segmenta [l,d]
    // koristimo prethodno izracunatu vrednost za inicijalizaciju
    if (i <= d)
      z[i] = min(d-i+1,z[i-l]);
```

```
    // poredimo karakter po karakter u niski
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
      z[i]++;
    // ako je nova vrednost desnog kraja intervala preklapanja
    // 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 tekst = "banana";
  string uzorak = "ana";
  string zajedno = uzorak + "#" + tekst;
  int n = zajedno.size();
  int m = uzorak.size();

  vector<int> zNiz = izracunajZNiz(zajedno);
  for (int i = 0; i < n; i++){
    if (zNiz[i] == m)
      cout << "Uzorak se javlja u tekstu na pocev od pozicije "
       << i - m - 1 << endl;
  }
  return 0;
}
```