Sravnjivanje niski

Hashiranje

1. Hash Function (en.wikipedia.org)
2. MD5 message-digest algorithm (en.wikipedia.org)
3. Birthday Paradox (en.wikipedia.org)
4. Hash Functions (www.cs.hmc.edu)
5. Good hash functions for integers (www.stackoverflow.com)
6. Anti-hash test (www.codeforces.com)

Heširanje niski

a) **Gde se primenjuje?**

a1. Algoritmi heširanja mogu biti korisni u rešavanju različitih problema nad tekstom, kao što je recimo pronalaženje uzorka u tekstu ili pronalaženje najduže niske koji se javlja bar dva puta kao segment date niske ili prilikom kompresije niske i sl.

a2. Heširanje je korisno i prilikom verifikacije lozinki: prilikom logovanja na neki sistem, računa se heš vrednost lozinke i šalje se serveru na proveru. Lozinke se na serveru čuvaju u heširanom obliku, iz bezbedonosnih razloga.

b) Prednosti heširanja niski kod string matching (sravnjivanje niske)

Korišćenje heširanja često služi da se algoritmi grube sile učine efikasnijim.
Razmotrimo problem upoređivanja dve niske karaktera.
Algoritam grube sile poredi karakter po karakter datih niski i složenosti je O(min{n1, n2}), gde su n1 i n2 dužine ulaznih niski. Da li postoji efikasniji algoritam?
PODSEĆANJE: http://poincare.matf.bg.ac.rs/~jelenagr/ASP/cas11.html

Svaku nisku možemo konvertovati u ceo broj i umesto niski karaktera uporediti dobijene brojeve. Konverziju vršimo pomoću heš funkcije, a dobijene cele brojeve nazivamo heš vrednostima (ili skraćeno heševima) niski karaktera.

Potrebno je da važi sledeći uslov: ako su dve niske s i t jednake, i njihove heš vrednosti h(s) i h(t) moraju biti jednake. Obratimo pažnju da obratno ne mora da važi: 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 azbuke, heš vrednosti niski ne bi stale u opseg 64-bitnih celih brojeva (a svakako nam nije cilj da upoređujemo cele brojeve sa proizvoljno mnogo cifara jer bi takvo poređenje bilo složenosti O(n), gde je n broj cifara).
Ovakvo upoređivanje je složenosti O(1), međutim samo heširanje je složenosti O(n1 + n2).

c) Šta znači pojam hash?

Hash u engleskom jeziku može biti imenica i glagol. Imenica označava pulpu, kašu, nered, a glagol se prevodi kao “narezati”. Na taj način se i dobijaju slike ulaznih objekata - oni se na neki način režu i mešaju, a (mali deo) pulpe koja se proizvodi je takozvani hash. Važno je napomenuti da nije moguće vratiti ulazni objekat iz "pulpe" - samo je poznato da je iz nje dobijen.

3. Dakle, naš cilj je preslikati skup niski u vrednosti iz fiksiranog opsega [0, m).

```
unsigned simpleHash(const string& str) {  // proizvodi integer overflow
    unsigned ret = 0;
    for (int i = 0; i < (int)str.size(); i++) {
        ret = ret * 256u + (unsigned)str[i];
    }
    return ret;
}
```

4. Šta je KOLIZIJA i kako se izbegava?

Pritom je, naravno, poželjno da za dve različite niske verovatno¢a da njihove heš vrednosti budu jednake, odnosno da dođe do kolizije, bude veoma mala. U velikom broju slučajeva mogu¢nost kolizije se ignoriše, potencijalno pritom narušavaju¢i korektnost algoritma. Druga mogu¢nost je da se u slučajevima kada se dobiju jednake heš vrednosti dve niske izvrši provera jednakosti ove dve niske karakter po karakter, čime se potencijalno žrtvuje efikasnost algoritma. Odabir jedne od ove dve opcije zavisi od konkretne primene.

```
const int BASE = 257;
const int MOD = 1000000007;
int stringHash(const string& str) {
    int ret = 1;
    for (int i = 0; i < (int)str.size(); i++) {
        ret = ((long long)ret * BASE + str[i]) % MOD;
    }
    return ret;
}
```

Prosti brojevi 1.000.000,007 i 1.000.000.009 su čest izbor pri odabiru odgovarajućih modula za mnoge zadatke.

Imajte na umu da je modul u ovoj implementaciji veliki prost broj, jer to je način na koji se štitimo od kolizija ulaznih stringova. Dajte primer! Nešto kasnije ćemo videti da je važno da zbog efikasnog izračunavanja modularnog multiplikativnog inverza broja $BASE^i$ po modulu MOD oba broja BASE i MOD budu prosti. Dakle, BASE i MOD biramo tako da budu dovoljno veliki prosti brojevi.

I zašto smo izabrali modul od 1.000.000.007 (samo oko 1.000.000.000), a ne, recimo, 2.000.000.011 (samo oko 2.000.000.000)?

Često ćemo morati ne samo imati brojeve, već i neke operacije s njima (npr. Rolling Hash i Rabin-Karp ispod).

S jedne strane, moraćemo biti u mogućnosti da izdvojimo heševe (koji potencijalno primaju negativne vrednosti) - što nam ne dozvoljava da koristimo tip neoznačenih celih brojeva, a sa druge ćemo morati da SABIRAMO heševe (potencijalno primajući duple brojeve) tj. u kod 32 bitnih označenih brojeva, to nas ograničava da za modul ne koristimo brojeve veće od 1.073.741.823.

4. **Polinomijalna heš funkcija**

Jasno je da heš funkcija treba da zavisi od dužine niske i od redosleda karaktera u niski. Dobar i široko rasprostranjen način definisanja heš funkcije niske s dužine n j je

preslikavanje (tzv. **polynomial rolling hash**):

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

PRIMER HASH-iranja

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

long long izracunajHes(const string &s)
{
    int p = 31;
    int m = 1e9 + 9;
    long long heshVrednost = 0;

    for (int i=s.size()-1; i>=0; i--)
    {
        heshVrednost = (heshVrednost*p + (s[i]-'a'+1)) % m;
    }

    return heshVrednost;
}
int main()
{
    vector<string> reci = {"ana", "a", "ananas","b","algoritamsravnjivanjaniski"  };

    for (string s : reci)
    {
        cout << "Hesh vrednost niske " << s << " je: "<< izracunajHes(s) << endl;
    }
    return 0;
}
```

```
Hesh vrednost niske ana je: 1396
Hesh vrednost niske a je: 1
Hesh vrednost niske ananas je: 545295860
Hesh vrednost niske b je: 2
Hesh vrednost niske algoritamsravnjivanjaniski je: 278481167
```

Da bi se račun sveo na manje brojeve, setimo se Hornereove sheme.
Dakle, u funkciji za računanje heš vrednosti niske s, mi smo računali međurezultate po modulu m, odnosno razmatrali smo niz međuvrednosti:

$heshVrednost_n=0$

$heshVrednost_i=(heshVrednost_{i+1}*p+s[i]) \bmod m$

Podsetimo se da je ovaj postupak matematički ispravan, jer važi:

$(a+b) \bmod m= ((a \bmod m) + (b \bmod m)) \bmod m$

$(a*b) \bmod m= ((a \bmod m) * (b \bmod m)) \bmod m$

1. Pretraga duplikata u nizu niski

Dat je spisak sa n niski si, tako da svaka niska nije duža od m karaktera. Konstruisati algoriitam složenosti O(nm+nlogn) koji pronalazi sve stringove dulikate i razdvaja polazni spisak u grupe.

Ideja 1: Sortiramo stringove i vršimo poređenje po 2 stringa. Ukupna složenost je O(nmlogn), jer sortiranje zahteva O(n log n) poređenja i svako poređenje zahteva O(m) operacija, (mada je očekivano da ono često bude efikasnije, da se razlika pronađe na prvih nekoliko karaktera), te bi ukupna složenost bila O(nm log n).

Ideja 2: Ali, korišćenjem heširanja (preslikavanje niski u broj), smanjujemo vremensku složenost poređenja na O(1), tako da ukupna složenost algoritma je O(nm+nlogn), , gde složenost O(nm) potiče od računanja heš vrednosti svih niski.

Računamo hash za svaku nisku, sortiramo zajedno i hash-eve i indekse iz polaznog spiska, i potom grupišemo indekse prema identičnim hash-evima.

Pre implementacije, da se podsetimo:
https://www.geeksforgeeks.org/emplace-vs-insert-c-stl/

```
vector<vector<int>> group_identical_strings(vector<string> const& s) {
    // broj niski
  int n = s.size();

/* vektor parova hesh vrednosti i pozicije niske u polaznom nizu niski */
   vector<pair<long long, int>> hashes(n);

// izracunavamo hesh vrednost svake niske
  for (int i = 0; i < n; i++)
      hashes[i] = {compute_hash(s[i]), i};

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

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

// 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 || hashes[i].first != hashes[i-1].first)
            groups.emplace_back();
/* u poslednju (tekucu) grupu dodajemo novu hesh vrednost na kraj */
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
```

TESTIRAJTE PROGRAM ZA ULAZ
vector<string> reci={"ana", "a", "dvorana", "ana", "banana", "ana", "kopakabana", "banana"};

Da se podsetimo: **compute_hash(s[i])**

Izračunajmo hash niske s, koja je sačinjena samo od malih slova engleske abecede. Preslikajmo svaki karakter niske s u ceo broj.

Možemo da koristimo preslikavanje: a→1, b→2, ……, z→26. Preslikavanje a→0 nije validno, jer bi se onda hash-evi niski a, aa, aaa, …… evaluirali u 0.

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Uobičejeno hash-iranje niske s dužine n je preslikavanje (tzv. **polynomial rolling hash**):

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

gde p i m su pažljivo odabrani prirodni brojevi.

Ima smisla da p bude prost broj koji je reda veličine broja karaktera u ulaznom alfabetu nad kojim se formira niska s.
Na primer, ako niska s je sačinjena samo od malih slova engleske abecede, onda vrednost p=31 je dobar izbor.
Ako niska s je sačinjena i od malih i od velikih slova engleske abecede, onda vrednost p=53 je dobar izbor.

Dalje, vrednost broja m mora biti veliki prirodan broj, jer verovatnoća kolizije dve nasumične niske je ≈1/m.
Ponekad se bira $m=2^{64}$, jer celobrojna pekoracenja 64 bita se ponašaju kao u prstenu po modulu 64 što nam je i potrebno. ALI; KAKO je moguće konstruisati metod koji generiše koliziju nisku koja postoji nezavisno od izbora broja p, TO SE IZBEGAVA izbor $m=2^{64}$.

U gornjem primeru koristi se strategija po kojoj je broj m neki veliki prost broj, npr. $m=10^9+9$. Ovaj veliki b roj je još uvek dovoljno mali da može da se obavi množenje dve vrednosti koristeći registar 64 bitni za cele brojeve.

2. Prebroj različite podniske u nisci

Zadatak: Data je niska s dužine n, koja se sastoji samo od malih slova engleske abecede. Ispišite broj različituh podniski niske s. Vremenska složenost $O(n^2 \log n)$

PODZADATAK: Za datu nisku s i indekse i i j, pronaći heš vrednosti segmenata s[i..j](podniske susednih karaktera polazne niske od pozicije i zaključno sa pozicijom j).

Prema definiciji važi:

$$h(s[i..j]) = (\sum_{k=i}^{j} s[k] \cdot p^{k-i}) \mod m$$

Ukoliko obe strane jednakosti pomnožimo sa p^i dobijamo:

$$h(s[i..j]) \cdot p^i = (\sum_{k=i}^{j} s[k] \cdot p^k) \mod m = (h(s[0..j]) - h(s[0..i-1])) \mod m$$

Stoga ako znamo heš vrednost svakog prefiksa date niske, možemo korišćenjem ove formule izračunati heš vrednost proizvoljnog segmenta date niske. Ova ideja odgovara računanju prefiksnih suma, sa kojima smo se upoznali ranije u kontekstu nekih drugih zanimljivih problema (segmentno stablo,…).

Važan problem koji se javlja jeste izvršiti deljenje izraza h(s[0..j]) − h(s[0..i − 1]) vrednošću p^i. Za to je potrebno odrediti modularni multiplikativni inverz broja p^i po modulu m (odnosno broj x tako da važi $p^i \cdot x = 1 \pmod m$) i onda izvesti množenje ovim inverzom. Može se unapred izraćunati modularni inverz za svako p^i (na primer, proširenim Euklidovim algoritmom, te je važno da broj p bude prost), čime je omogućeno izračunavanje heš vrednosti proizvoljnog segmenta u vremenu O(1).
Ali, postoji i drugi način, jednostavniji za implementaciju. Često, umesto da izračunamo tačne heš vrednosti dva segmenta, dovoljno je izračunati heš vrednosti pomnožene nekim stepenom broja p. Pretpostavimo da imamo izračunate heš vrednosti dva segmenta, jednog pomnoženog sa p^i, a drugog sa p^j . Ako je i < j pomnožićemo prvi heš sa p^{j-i}, inače množimo drugi sa p^{i-j} . Na ovaj način dobijamo oba heša pomnožena istim stepenom broja p i ove dve vrednosti možemo uporediti bez potrebe za deljenjem. Dakle, na ovaj način nakon

preprocesiranja koje je složenosti O(n), možemo u vremenu O(1) izračunati heš vrednost proizvoljnog segmenta.

Vratimo se na zadatak

IDEJA: Iterirajmo preko svih podniski dužine l=1…n.
Za svaki podstring dužine l, konstruišimo niz hash-eva svih podstringova dužine l koje ćemo pomnožiti sa istim eksponentom broj p.
Potrebno je uporediti samo segmente istih dužina jer oni mogu biti međusobno jednaki.
Prolazimo redom kroz sve moguće dužine l = 1, 2, . . . , n segmenata i konstruišemo 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.
Na kraju iteracije, taj broj se dodaje konačnom odgovoru.

Koristimo u implementaciji h[i] kao hash prefiksa podniske sa i karaktera, i uvodimo da praznu podnisku h[0]=0

```
int count_unique_substrings(string const& s) {
    int n = s.size();

    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
// racunamo stepene broja p
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

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

// broj razlicitih segmenata
    int cnt = 0;
// za svaku mogucu duzinu segmenta
    for (int l = 1; l <= n; l++) {
    // skup hesh vrednosti svih segmenata duzine l
        set<long long> hs;
   // prolazimo kroz sve segmente duzine l
        for (int i = 0; i <= n - l; i++) {
    //racunamo vrednost hesha segmenta pomnozenog sa p^{n-1}
            long long cur_h = (h[i + l] + m - h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
         // dodajemo u skup, isti segmenti imace isti hes
         // pa se nece dva puta racunati
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
```

```
    return cnt;
}
```

Diskusija (rođendanski paradoks):
Često je gore definisana polinomijalna heš funkcija dovoljno dobra i prilikom testiranja ne¢e doći do kolizije.
Na primer, za odabir $m = 10^9 + 9$ verovatnoća kolizije je samo $1/m = 10^{-9}$, pod pretpostavkom da su sve heš vrednosti jednako verovatne.
Međutim ukoliko nisku s poredimo recimo sa 10^6 različitih niski verovatno¢a da se desi bar jedna kolizija je oko 10^{-3}, a ako poredimo 10^6 niski međusobno verovatnoća bar jedne kolizije je 1.

Poslednja pomenuta pojava poznata je pod nazivom rođendanski paradoks i odnosi se na sledeći kontekst: ako se u jednoj sobi nalazi n osoba, verovatno¢a da neke dve imaju rođendan istog dana iznosi oko 50% za n = 23, dok za n = 70 ona iznosi čak 99.9%.

Postoji jednostavni trik kojim se može smanjiti verovatno¢a pojave kolizije – računanjem dve različite heš vrednosti (npr koriš¢enjem iste funkcije za različite vrednosti parametara p i/ili m).
Ako m ima vredbost oko 10^9 za obe funkcije, onda je ovo uporedivo sa koriš¢enjem jedne heš funkcije za vrednost m približno jednako 10^{18}.
Sada, ako poredimo 10^6 niski međusobno, verovatno¢a kolizije smanji¢e se na 10^{-6}.

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

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

KMP, Manacher, Z-algorithm, Rabin-Karp

3. **Najkraća dopuna do palindroma**

vremensko ograničenje	memorijsko ograničenje
0.1 s	64 MB

https://arena.petlja.org/sr-Latn-RS/competition/27#tab_97511
Niska abaca nije palindrom (ne čita se isto sleva i sdesna), ali ako joj na početak dopišemo karaktere ac, dobijamo nisku acabaca koja jeste palindrom (čita se isto i sleva i sdesna).
Napiši program koji za datu nisku određuje dužinu najkraćeg palindroma koji se može dobiti dopisivanjem karaktera s leve strane date niske.

Ulaz
Sa standardnog ulaza se unosi niska sastavljena samo od N ($1 \leq N \leq 50000$) malih slova engleske abecede.

Izlaz
Na standardni izlaz se ispisuje tražena dužina najkraće proširene niske koja je palindrom.
Primer 1
Ulaz
abaca
Izlaz

7
Primer 2
Ulaz
abcdefg
Izlaz
13
Rešenje je gfedcbabcdefg.
Primer 3
Ulaz
anavolimilovana
Izlaz
15
Reč je već palindrom, pa se ne dopisuje ni jedan karakter i rešenje je anavolimilovana.
Primer 4
Ulaz
anavolimilovanakapak
Izlaz
25
Rešenje je kapakanavolimilovanakapak.

Rešenje

Sporije rešenje (pokriva vremensko ograničenje za neke test primere)

Neka je data ulazna niska s="abbac"

Dovoljno je da postavimo poslednje slovo c ispred sadržaja niske s i dobićemo nisku "cabbac", koje jeste palindrom s minimalnom dužinom, i zadovoljava uslove zadatka.

Sličan postupak se može primeniti i u opštem slučaju, kada ulazna niska *s* sadrži podniz na početku niske koji je palindrom.

Predstavimo ulaznu nisku u obliku s = s1 + s2, gde s1 je najduži prefix od s, koji je palindrom.
Tada možemo da sastavimo niz t = s3 + s1 + s2, gde niska s3 je dobijena obrtanjem niske s2 (pročitane kao u ogledalu) .

Kako niska s1 je najduži palindrom, onda niska t je palindrom, napravljen po uslovima zadatka i to najkraći moguć.

Dužina niske t je:

L = n2+n1+n2=2*n2 + n1,

gde n1 je dužina za s1, n2 je dužina za s2 (naravno i za s3). Polazna dužina niske s je n tj. n = n1 + n2. Dakle, L = 2*(n-n1) + n1 = 2*n – n1 i program mora da ispiše L.

Ako želino da odredimo n1, onda redom proveravamo sve prefikse niske s i tražimo najduži palindrom među njima. Njegova dužina je n1.

```
#include <iostream>
#include <string>

using namespace std;

bool jePalindrom(const string& s, int n) {
for (int i = 0, j = n - 1; i < j; i++, j--)
    if (s[i] != s[j])  return false;
return true;
}

int duzinaNajduzegPalindromskogPrefiksa(const string& s) {
for (int n = s.size(); n >= 1; n--)
   if (jePalindrom(s, n))  return n;
}

int main() {
string s;
cin >> s;
// s razlazemo na prefiks + sufiks tako da je prefiks sto duzi
// palindrom. Tada je trazeni palindrom (palindrom koji se dobija sa
// sto manje dopisivanja slova na pocetak niske s) jednak:
// obrnut_sufiks + prefiks + sufiks
int duzinaPrefiksa = duzinaNajduzegPalindromskogPrefiksa(s);
int duzinaSufiksa = s.size() - duzinaPrefiksa;
int duzinaNajkracegPalindroma = duzinaSufiksa + duzinaPrefiksa + duzinaSufiksa;
cout << duzinaNajkracegPalindroma << endl;
return 0;
}
```

Algoritam složenosti O(n)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int duzinaNajduzegPalindromskogPrefiksa(const string& s) {
string sObratno(s.rbegin(), s.rend());
string str = s + '.' + sObratno;
vector<int> kmp(str.size() + 1, 0);
 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;
}


return min(kmp[kmp.size() - 1], (int)s.size());
```

}

```
int main() {
string s;
cin >> s;
// s razlazemo na prefiks + sufiks tako da je prefiks sto duzi
// palindrom. Tada je trazeni palindrom dobijen sa sto manje
// dopisivanja slova na pocetak jednak:
// obrni(sufiks) + prefiks + sufiks
int duzinaPrefiksa = duzinaNajduzegPalindromskogPrefiksa(s);
int duzinaSufiksa = s.size() - duzinaPrefiksa;
int duzinaNajkracegPalindroma = duzinaSufiksa + duzinaPrefiksa + duzinaSufiksa;
cout << duzinaNajkracegPalindroma << endl;
return 0;
}
```

Ulaz Izlaz
abaca 7

KMP tablica za abaca.acaba

index: 0 1 2 3 4 5 6 7 8 9 10
kmp[index]:-1 0 0 1 0 1 0 1 0 1 2

Ulaz Izlaz
anavolimilovana 15

KMP tablica za anavolimilovana.anavolimilovana

index: 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 29 30
kmp: -1 0 0 1 0 0 0 0 0 0 0 0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Ulaz Izlaz
anavolimilovanakapak 25

KMP tablica za anavolimilovanakapak.kapakanavolimilovana

ind: 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 29 30 31 32 33 34 35 36 37 38 39 40
kmp:-1 0 0 1 0 0 0 0 0 0 0 0 0 1 2 3 0 1 0 1 0 0 0 1 0 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

4. https://petlja.org/biblioteka/r/problemi/zbirka-napredni-nivo/maksimalan_palindrom

Dat je string s koji sadrži samo mala slova. Prikazati najduži segment stringa s (niz uzastopnih elemenata stringa) koji je palindrom u stringu s. Ako ima više najdužih segmenata prikazati segment čiji početak ima najmanji indeks.

Ulaz

Prva i jedina linija standardnog ulaza sadrži string sastavljen od malih slova.

Izlaz

Na standardnom izlazu prikazati prvi najduži segment datog stringa koji je palindrom.

Primer 1

Ulaz
cabbadcmmc
Izlaz
abba

Primer 2

Ulaz
babcbabcbaccba
Izlaz
abcbabcba

NEKOLIKO IDEJA REŠENJA

Provera svih segmenata

Zadatak je moguće rešiti analizom svih segmenata (pomoću ugnežđene petlje, kao u zadatku Sve podreči), proverom da li je tekući segment palindrom (kao u zadatku Niska palindrom) i određivanjem najdužeg pronađenog palindroma (algoritmom određivanja maksimuma serije, kao u zadatku Najmanja temperatura).
Pošto je provera palindroma složenosti $O(n)$, a segmenata ima $O(n^2)$, složenost ovog pristupa je $O(n^3)$.

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 (slično kao u zadatku Sve podreči po opadajućoj dužini). 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 (slično kao u slučaju algoritama pretrage, kao, na primer, u zadatku Negativan broj) ili naredbom `goto` (iako je potrebno izbegavati je, neki autori smatraju da je prekid ugnežđenih petlji jedina situacija u kojoj je upotreba `goto` opravdana). 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 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 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 na 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 $O(n)$ i ukupna složenost algoritma je $O(n^2)$.

Dopuna reči i pozicije

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 dopunjenog reči i dovoljno je analizirati samo palindrome neparne dužine u njemu. Ovo dopunjavanje je moguće realizovati i fizički (u programu kreirati dopunjeni string), što može malo da olakša implementaciju po cenu malo sporijeg programa (doduše ne asimptotski) i dodatnog zauzeća memorije. Ipak naredno razmatranje nam govori da ove pomoćne karaktere možemo samo razmatrati dok razmatramo algoritam, bez eksplicitnog pravljenja dopunjenog stringa u programu.

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 $k = \left\lfloor \frac{p}{2} \right\rfloor$.

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 $\left\lfloor \frac{i-d-1}{2} \right\rfloor$ i $\left\lfloor \frac{i+d+1}{2} \right\rfloor$) 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 izađe van opsega dopunjene reči.
Recimo još i da se i provera pripadnosti indeksa tj. pozicija opsegu reči može eliminisati ako se polazna reč proširi sa 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).

Manačerov algoritam

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.

```
   0   1   2   3   4   5   6   7   8   9   10  11  12  1     - indeksi i u reci s
|  b  |  a  |  b  |  c  |  b  |  a  |  b  |  c  |  b  |  a  |  c  |  c  |  b  |  a  |  - prosirena rec t
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  - pozicije p
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   - d[p]
```

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 tj $C - i' = i - C$, 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 to 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 polinoma). Moguće je dokazati da je složenost ovog algoritma $O(n)$ (intuitivno, pronalaženje kratkih palindroma zahteva mali broj izvršavanja unutrašnje petlje, dok je 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 izbegava njeno izvršavanje).

Rešenje 1 (efikasno)

```
#include <iostream>

#include <string>
#include <vector>

using namespace std;

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
  // to je ujedno i broj pozicija levo i desno od i na kojima se podaci
  // poklapaju – ili su obe pozicije izmedju slova ili su na njima jednaka slova
  vector<int> d(N);

  // Ilustrujmo d[i] na primeru reci babcbabcbaccba
  // 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  -
pozicije
  // |  b  |  a  |  b  |  c  |  b  |  a  |  b  |  c  |  b  |  a  |  c  |  c  |  b  |  a  |   - rec s
  //    0     1     2     3     4     5     6     7     8     9     10    11    12    1       - indeksi unutar s
  // 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   - d[i]


  // Primecujemo da je niz d[i] visoko simetrican i da se mnoge
  // njegove vrednosti mogu odrediti na osnovu prethodnih, bez potrebe
  // za ponovnim izracunavanjem. Npr. posto je d[11] = 9, rec
  // abcbabcba je palindrom. Zato je npr. d[10] = d[12] = 0 ili d[9] =
  // d[11] = 1. To mozemo da tvrdimo jer su ti palindromi duzine 1
  // zajedno sa karakterima koji ne dopustaju njihovo produzivanje (to
  // su s leve strane cba i sa desne strane abc) ukljuceni u palindrom
  // abcbabcba.
  // Ipak, d[7] = 7, sto je razlicito od d[15] = 5.  Uzrok tome je to
  // sto palindrom duzine 7 oko pozicije 7 babcbab nije ceo ukljucen u
  // palindrom abcbabcba. Ipak, Mozemo zakljuciti da je d[15] >= 5 (jer
  // nam to obezbedjuje deo abcba koji jeste ceo ukljucen u abcbabcba.
```

```
// Dalje od toga je moguce utvrditi samo proverom daljih karaktera.

// 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])
      d[i] += 2; // ukljucujemo dva slova u palindrom, pa mu se duzina uvecava za 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;

}
```

Uporedi rešenje 1 sa sledećih 5 rešenja:
Rešenje 2:

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// 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;

}
```

Rešenje 3:

```
#include <iostream>

#include <string>

using namespace std;

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;

}
```

Rešenje 4:

```
#include <iostream>

#include <string>
using namespace std;

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;
}
```

Rešenje 5:

```
#include <iostream>


using namespace std;

// 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--;
  }
  return i >= j;
}

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
  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;

}
```

Rešenje 6:

```
#include <iostream>

#include <string>
#include <vector>

using namespace std;

// 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--;
  }
  return i >= j;
}

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

  int maxDuzina = 0, maxPocetak = 0;
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      if (palindrom(s, i, j)) {
          int duzina = j - i + 1;
          if (duzina > maxDuzina) {
            maxDuzina = duzina;
            maxPocetak = i;
          }
      }

  cout << s.substr(maxPocetak, maxDuzina) << endl;

  return 0;

}
```

5.

Za string S veličine *N*, kažemo da *Ai* je veličina najdužeg prefiksa koji se može ponovo pronaći u *S počev* **od pozicije** *i*. Smatramo da *A*1 nije definisan.
S druge strane, neka *Bi* je veličine najdužeg prefiksa koji može da se nađe ponovo u *S* tako da se završava na poziciji *i*. *Bi* je strogo manje od *i*, i važi da *B*1 nije definisano.
Na primer, ako *S*=*ababacaba*, onda je A=[-,0, 3, 0,1,0,3,0,1], B=[-,0,1,2, 3,0,1,2,3]
Nije Vam poznato *S*, ali je dato *A* i *B*.

ULAZ: U prvoj liniji standardnog ulaza dat je ceo broj *N* ($2 \leq N \leq 10^5$). Druga linija sadrži *N*−1 celih brojeva tako da *iti* broj jeste *Ai*+1.

IZLAZ: Ispisati na standardni izlaz $N-1$ celih brojeva u prvoj liniji međusobno razdvojenih blanko karakterom tako da *iti* broj je B_i+1.

Možete pretpostaviti da su test primeri takvi da će uvek biti moguće da se generiše string S koji odgovara za ulaz A koristeći alfabet koji ima najviše 10^5. Garantuje se jedinstvenost rešenja.
Vremensko ograničenje: *1 s*
Memorijsko ograničenje: *128 MB*

Ulaz	Izlaz	Pojašnjenje
6 2 1 0 2 1	1 2 0 1 2	Moguće rešenje je: *aaabaa*
5 4 3 2 1	1 2 3 4	*aaaaa*
6 0 3 0 1 0	0 1 2 3 0	*ababac*
11 0 1 0 3 0 3 0 2 0 0	0 1 0 1 2 3 2 3 2 0	*abacababababc*

Pretpostavimo da znamo string S i želimo da nađemo string B.

Rešenje grubom silom: izaberemo svaki indeks $i \geq 2$ i nađimo najduži prefiks koji počinje na toj poziciji. Kada proširimo prefiks, kada dođemo do indeksa j moramo ažurirati B_j na vrednost $j-i+1$.

Ali, šta se dešava ako želimo da ažuriramo istu vrednost j dva puta? Zapravo, zanima nas samo prvi put da ažuriramo određen j, jer je vrednost $j-i+1$ veća što je vrednost pozicija i manja. Niz A govori koji je to najduži prefiks koji počinje na svakoj poziciji i, tako da možemo doći do jednostavnog rešenja koje uključuje red (queue).

Održavaćemo queue sa svim indeksima i koji mogu ažurirati sve sledeće vrednosti j. Kada dospemo do nove pozicije, onda postavimo poziciju u queue ako prefiks počinje na toj poziciji i završava se nakon poslednjeg elementa u queue. Inače, odbacujemo poziciju, jer se neće koristiti. Da bismo našli rešenje za tekući indeks, proveravamo početak reda (queue). Ako taj prefiks više ne ažurira tekuću poziciju, onda uzimamo element iz reda i ispitujemo narednog kandidata.

Rešenje 1

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_N = (int)1e5;

int n;
int A[MAX_N];

int main() {
    cin >> n;

    queue<int> q;
    for (int i = 2; i <= n; ++i) {
        cin >> A[i];
        if (A[i]) {
            if (!q.size() || A[q.back()] + q.back() < A[i] + i) {
                q.push(i);
            }
        }
        if (q.size() && A[q.front()] + q.front() <= i) {
            q.pop();
        }
        if (q.size()) {
            cout << i - q.front() + 1 << " ";
        } else {
            cout << "0 ";
        }
    }
}
```

Rešenje 2

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


vector<int> solve(const vector<int> left) {
        int n = (int)left.size();
        vector<int> right(n);
        for (int i = 1; i < n; i += 1) {
                right[i + left[i] - 1] = max(right[i + left[i] - 1], left[i]);
        }

    for (int mx = 0, i = n - 1; i; i -= 1) {
        mx--;
        mx = max(mx, right[i]);
        right[i] = mx;
    }

    return right;
}
```

```
int main() {
    int n; cin >> n;
    vector<int> left(n);
    for (int i = 1; i < n; i += 1) {
        cin >> left[i];

    }

    auto right = solve(left);
    for (int i = 1; i < n; i += 1) {
        cout << right[i] << ' ';
    }
    return 0;
}
```

6. https://petlja.org/biblioteka/r/problemi/zbirka-napredni-nivo/prefiks_sufiks

Домине се слажу једна уз другу, тако што се поља на доминама постављеним једну уз другу морају поклапати. Домине обично имају само два поља, међутим, наше су домине специјалне и имају више различитих поља (означених словима). Ако све домине које слажемо имају исту шару, напиши програм који одређује како је домине могуће сложити тако да заузму што мање простора по дужини (свака наредна домина мора бити смакнута бар за једно поље). На пример, ако на доминама пише ababcabab, најмање простора заузимају ако се сложе на следећи начин:

```
ababcabab
  ababcabab
    ababcabab
```

Улаз: Први ред стандардног улаза садржи ниску малих слова енглеске абецеде које представљају шаре на доминама. Дужина ниске је између 2 и 100000 карактера. У наредном реду се налази цео број n (1 ≤ n ≤ 50000) који представља број домина.

Излаз: На стандардни излаз исписати цео број који представља укупну дужину сложених домина.

Primer 1
Ulaz
ababcabab
3
Izlaz
19
Primer 2
Ulaz
abc
5
Izlaz
15
Primer 3
Ulaz
aa

10
Izlaz
11
Rešenje

Ključ rešenja ovog zadatka je određivanje najdužeg pravog prefiksa niske koji je ujedno i njen sufiks (takav prefiks tj. sufiks zvaćemo prefiks-sufiks). Jedan način da se on pronađe je gruba sila (da se uporede svi pravi prefiksi sa odgovarajućim sufiksima i da se od onih koji se poklapaju odabere najduži). Ipak, mnogo efikasnije rešenje zasnovano je na Knut-Moris-Pratovom (KMP) algoritmu.

Rekurzivna formulacija

Zadatak ima rekurzivno rešenje. Neka je niska s dužine n. Ako je ta niska prazna ili jednoslovna, tada sigurno nema (pravi) prefiks-sufiks.

Zato pretpostavimo da je $|s|\geq 2$ i da je $s=s'a$.

Pretpostavimo da je d dužina najdužeg prefiks-sufiksa x reči ss′, tj. dela niske s koji se završava na njenoj pretposlednjoj poziciji (on se može izračunati rekurzivno, jer je reč s′ kraća od reči s). Tada je $s'=xu=vx$.

Neka je b prvo slovo niske u (ono postoji jer je x pravi prefiks-sufiks niske s′) i da je $u=bu'$. Tada je $s=xbu'a=vxa$.

Pošto je $|x|=d$, važi da je $b=sd$, dok je $a=s_{n-1}$.

Ako je $a=b$, tj. ako je $s_{n-1}=s_d$ tada je $xa=xb$ najduži prefiks-sufiks niske s i njegova dužina je d+1.

Ostaje pitanje šta raditi u slučaju kada je $a\neq b$. Ako je x prazan, tj. ako je d=0, tada ne postoji pravi prefiks-sufiks reči s.

U suprotnom, sledeći kandidat za prefiks-sufiks reči s je najduži pravi sufiks od x koji je ujedno pravi prefiks od x koji se možda može produžiti slovom a. On se može izračunati rekurzivno (jer je reč x kraća od reči s, pa se proces mora zaustaviti). U tom slučaju ažuriramo x tj. njegovu dužinu d i ponavljamo poređenje s_d i s_{n-1} i postupak nastavljamo sve dok se ne ispostavi da su oni jednaki ili da su različiti i da je dužina d=0.

Dinamičko programiranje

Možemo primetiti da će se u prethodno opisanom rekurzivnom postupku mnogi rekurzivni pozivi poklapati (više puta će se određivati dužina najdužeg prefiks-sufiksa za isti prefiks niske s). Stoga je potrebno primeniti memoizaciju, ili još bolje dinamičko programiranje. U toj situaciji održavamo niz d_i koji sadrži dužine najdužih prefiks-sufiksa za prefiks niske s dužine i. Tako će d_n sadržati traženu dužinu najdužeg prefiks-sufiksa cele niske s. Prva dva elementa niza možemo inicijalizovati na nulu (jer prazna i jednoslovna niska ne mogu imati pravi prefiks-sufiks). Nakon toga, popunjavamo ostale elemente niza, jedan po jedan.

Dužinu najdužeg prefiks-sufiksa reči $s_0\ldots s_i$ beležimo kao d_{i+1} (jer ta reč ima tačno i+1 slovo). Tu dužinu određujemo tako što poredimo s_i sa s_{di} (zaista, d_i je dužina najdužeg prefiks-

sufiksa za $s_0 \ldots s_{i-1}$, pa se prvi sledeći karakter iza njega nalazi na poziciji s_{di}). Ako su jednaki, važi da je $d_{i+1}=d_i+1$. U suprotnom, ako je d_i jednak 0, zaključujemo da je $d_{i+1}=0$, a ako nije, nastavljamo isti postupak tako što umesto d_i posmatramo d_{di} i poredimo s_i sa s_{ddi}. Zapravo, možemo promenljivu d inicijalizovati na d_i, a zatim porediti s_i sa s_d, ako su jednaki postavljati d_{i+1} na d+1 i prekidati postupak, a ako nisu, onda postavljati d_{i+1} na nulu i prekidati postupak Ako niz d realizujemo nizom kmp, a nisku s predstavimo sa str, možemo formulisati sledeći kod.

```
kmp[0] = kmp[1] = 0;
for (int i = 1; i < str.size(); i++) {
  int k = kmp[i];
  while (true) {
    if (str[i] == str[k]) {
      kmp[i+1] = k + 1; break;
    } else if (k == 0) {
      kmp[i+1] = 0; break;
    } else
      k = kmp[k];
  }
}
```

REŠENJE 1

```
#include <iostream>

#include <string>
#include <vector>

using namespace std;

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()] + n * (str.size() - kmp[str.size()]) << endl;

  return 0;

}
```

REŠENJE 1 (PYTHON)

```
str = input()

l = len(str)
```

```
n = int(input())
kmp = [0] * (l + 1)
kmp[0] = -1;
for i in range(0, l):
 k = i
 while k > 0 and (str[i] != str[kmp[k]]):
   k = kmp[k]
 kmp[i + 1] = kmp[k] + 1


print(kmp[l] + n * (l - kmp[l]))
```

REŠENJE 2

```
#include <iostream>

#include <string>
#include <vector>

using namespace std;

int main() {
  string str;
  cin >> str;
  int n;
  cin >> n;

  int maxD = 0;
  for (int d = 1; d < str.size(); d++) {
    bool prefiks_suffix = true;
    for (int i = 0, j = str.size() - d; i < d; i++, j++)
      if (str[i] != str[j]) {
            prefiks_suffix = false;
            break;
      }
    if (prefiks_suffix)
      maxD = d;
  }

  cout << maxD + n * (str.size() - maxD) << endl;
  return 0;

}
```

REŠENJE 3

```
#include <iostream>

#include <string>
#include <vector>

using namespace std;
```

```
// odredjuje najduzi pravi prefiks-sufiks reci str[0, ..., i]
int prefiks_sufiks(const string& str, int i);

int prefiks_sufiks_(const string& str, int k, int i) {
  int ps = prefiks_sufiks(str, k);
  if (str[ps] == str[i])
    return ps + 1;
  else if (ps == 0)
    return 0;
  return prefiks_sufiks_(str, ps - 1, i);
}

int prefiks_sufiks(const string& str, int i) {
  if (i == 0)
    return 0;
  return prefiks_sufiks_(str, i - 1, i);
}

int main() {
  string str;
  cin >> str;
  int n;
  cin >> n;

  int ps = prefiks_sufiks(str, str.size() - 1);
  cout << ps + n * (str.size() - ps) << endl;
  return 0;

}
```

RESENJE 4 (hashiranje)

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

const int MAXN=50010;
const int BASE=29;
const int MOD=1000000007;
#define velicina(ulaz) int((ulaz).size())
long long l, r, stepen[MAXN];
int main()
{
   string shara;
   int n;
   cin >> shara >> n;

   // pravimo potencije (stepene) broja BASE
   st[0] = 1;
   for(int i = 1; i < MAXN; i++)
```

```
    {
        stepen[i] = stepen[i-1]*BASE;
        if(st[i] >= MOD)
            st[i]%= MOD;
    }

    //otlrivamo najduzi prefiks niske shara koji je ujedno i sufiks niske shara
    int preklapanje = 0;
    for(int i = 0; i < velicina(shara)-1; i++)
    {
        l += stepen[i]*(shara[i]-'a');  //hashiramo pravi prefiks duzine i+1 (od 0. karakteta do i. karaktera)
        r = r*BASE + (shara[velicina(shara)-1-i]-'a');  //hashiramo pravi sufiks duzine i+1
        if(l >= MOD)  //racunanje u prstenu Zmod
            l %= MOD;
        if(r >= MOD)   //racunanje u prstenu Zmod
            r %= MOD;
        if(l == r)   //BINGO: pravi prefiks duzine i+1 se poklapa sa pravim sufiksom iste duzine
            preklapanje = i+1;
    }

    cout << velicina(shara) + (n-1)*(velicina(shara)-preklapanje);
    return 0;
}
```

7. Zadaci za samostalan rad

https://arena.petlja.org/competition/2019stringmatching