



## Opisi algoritama

Zadatke, testne primjere i rješenja pripremili: Fabijan Bošnjak, Nikola Dmitrović, Marin Kišić, Josip Klepec, Daniel Paleka, Ivan Paljak, Tonko Sabolčec i Paula Vidas. Primjeri implementiranih rješenja su dani u priloženim izvornim kodovima.

### Zadatak: Koeficijent

Pripremio: Nikola Dmitrović

Potrebno znanje: naredba učitavanja i ispisivanja

Nekoliko je različitih načina na koje možemo riješiti ovaj zadatak. Najjednostavniji način je ispis izraza oblika  $(N - 1) + 1$ . Uočite da ispis razmaka, koji Python obavezno dodaje, nije dozvoljen što nas dovodi do činjenice da u naredbi ispisa trebamo koristiti svojstva separatora `sep`.

*Programski kod (pisan u Python 3):*

```
N = int(input())
print(N-1, '+', 1, sep = '')
# može i ovako.. print(N-1, 1, sep='+')
# ili ovako... print(str(N-1) + '+' + str(1))
```

Drugačije rješenje moglo je ići u smjeru kreiranja stringa oblika  $1 + 1 + \dots + 1$  s ukupno  $N$  jedinica.

*Programski kod (pisan u Python 3):*

```
N = int(input())
s = '1'
for i in range(N-1):
    s += '+1'
print(s)
```

### Zadatak: Hajduk

Pripremio: Fabijan Bošnjak

Potrebno znanje: naredba ponavljanja, naredba odlučivanja

Odredimo najprije koliko je glasova dobio trener s oznakom 1, a koliko su glasova dobili ostali treneri. Ovo bismo mogli napraviti tako da, prilikom učitavanja, varijablu  $A$  povećamo za 1 ako smo učitali broj 1, a varijablu  $B$  povećavamo za 1 ako smo učitali broj različit od 1.

Sve što nam preostaje je odrediti je li trener s oznakom 1 osvojio barem polovicu glasova. Ovaj uvjet elegantno možemo provjeriti uspoređivanjem varijabli  $A$  i  $B$ . Naime, ako je  $A \geq B$ , tada je trener s oznakom 1 sigurno osvojio barem polovicu glasova.

Naravno, isti uvjet mogli smo provjeriti uspoređivanjem varijable  $A$  s varijablom  $N$  (ukupnim brojem glasova), međutim, naivna implementacija ovog smjera razmišljanja mogla nas je koštati bodova. Matematički gledano, uvjet "trener 1 osvojio je barem polovicu glasova" odgovara izrazu  $A \geq \frac{N}{2}$  kojeg bismo naivno u naš program mogli ukomponirati u obliku  $A \geq N/2$ . Greška u dijelu podržanih programskih jezika se krije u činjenici da znak `'/'` označava tzv. cjelobrojno dijeljenje. Probajte sami pronaći primjer u kojem ovakva implementacija nije ispravna.

Problem cjelobrojnog dijeljenja rješavamo jednostavnom algebarskom manipulacijom pa umjesto  $A \geq N/2$  pišemo  $2 * A \geq N$ . Na taj smo način izbjegli cjelobrojno dijeljenje.

Na kraju još valja upozoriti na rješenja koja se oslanjaju na "pravo dijeljenje", odnosno, na rad s decimalnim brojevima. U ovom zadatku to uglavnom neće biti problem, no načelno vam savjetujemo da izbjegavate rad s decimalnim brojevima ako je to moguće. Više o ovoj temi možete pročitati [ovdje](#).



## Zadatak: Preokret

Pripremio: Nikola Dmitrović

Potrebno znanje: naredba ponavljanja, rad s nizovima

Zadatak se sastoji od tri dijela. Za svakog natjecatelja po nešto. Krenimo redom. Da bi odrediti koliko je koji tim postigao golova dovoljno je učitavati zadane vrijednosti i brojati koliko se puta pojavio broj 1, a koliko broj 2.

*Programski kod (pisan u Python 3):*

```
N = int(input())
city = protivnik = 0
for i in range(N):
    gol = int(input())
    if gol == 1:
        city += 1
    else:
        protivnik += 1
print(city, protivnik)
```

Nastavimo dalje. Da bi odredili i koliko se puta dogodio neriješen rezultat trebamo pratiti kada je broj postignutih golova jednak, tj. kada će razlika postignutih golova biti jednaka nuli.

*Programski kod (pisan u Python 3):*

```
N = int(input())
city = protivnik = 0
nerijeseno = 1 # zbog 0:0
for i in range(N):
    gol = int(input())
    if gol == 1:
        city += 1
    else:
        protivnik += 1
    nerijeseno += (city == protivnik)
print(city, protivnik)
print(nerijeseno)
```

Za treći dio zadatka uočimo da se preokret sastoji od tri dijela. Prvo, jedan od timova gubi i počne postizati golove. Drugo, nakon niza postignutih golova dođe do neriješenog rezultata. Treće, tu ne stane već nastavi davati golove. Znači, razlika postignutih golova prvo pada do nule i onda nastavi rasti. Ili obrnuto, ovisno jel li City pravi preokret ili njegov protivnik. Jednu od implementacija rješenja možete pronaći u priloženim izvornim kodovima.



## Zadatak: Grudanje

Pripremio: Marin Kišić

Potrebno znanje: prefiks sume, binarna pretraga

Za 14 bodova bilo je potrebno samo simulirati ono što piše u zadatku. Jednom for-petljom ćemo slovo po slovo označavati s ‘\*’. Nakon toga ćemo drugom for-petljom proći po svim intervalima te za svaki interval trećom for-petljom provjeriti je li savršen.

Vremenska složenost ovog dijela je  $\mathcal{O}(NQN) = \mathcal{O}(QN^2)$ .

Za dodatnih 14 bodova bilo je potrebno poznavati ideju prefiks suma. Recimo da za svako slovo imamo niz *pref* u kojem će na poziciji *i* pisati broj tog slova od početka riječi do *i*-tog slova. Primjerice, za riječ **abcaab** i slovo ‘a’, niz *pref* bio bi [1, 1, 1, 2, 3, 3]. Niz *pref* nam je koristan jer sada možemo u  $\mathcal{O}(1)$  odrediti koliko puta se to slovo pojavljuje u intervalu od *L*-tog slova do *R*-tog slova u riječi. Formula je  $pref[R] - pref[L - 1]$ .

Sad je ideja da prije prolaska po intervalima izračunamo niz *pref* za svako slovo. Zatim, kada prolazimo po intervalima ćemo, uz pomoć niza *pref*, za svako slovo odrediti koliko ga ima u trenutnom intervalu.

Vremenska složenost ovog dijela je  $\mathcal{O}(N(N\Sigma * Q\Sigma)) = \mathcal{O}((N^2 + Q)\Sigma)$ , gdje  $\Sigma$  označava veličinu abecede (u našem slučaju  $\Sigma = 26$ ).

Za sve bodove bilo je potrebno primijetiti sljedeće: ako je riječ savršena nakon *i*-te grude, tada će onda biti savršena i nakon svake grude poslije *i*-te grude. To svojstvo nam omogućava da binarnom pretragom pronađemo prvu grudu nakon koje riječ postane savršena. Provjeru u binarnoj pretrazi radit ćemo kao i u prethodnoj parcijali (uz pomoć prefiks suma).

Vremenska složenost:  $\mathcal{O}((N\Sigma + Q\Sigma) \log N) = \mathcal{O}((N \log N + Q)\Sigma)$ .

## Zadatak: Drvca

Pripremili: Marin Kišić i Josip Klepec

Potrebno znanje: ad-hoc, ugrađene strukture podataka

Formalnim rječnikom, zadatak je bio razdvojiti zadanih *N* brojeva u 2 aritmetička niza.

Za 20 bodova bilo je dovoljno u  $\mathcal{O}(2^N)$  fiksirati neku bitmasku u kojoj vrijednost *i*-tog bita govori u kojem se redu nalazi *i*-ti broj. Nakon tog trebalo je još provjeriti jesu li redovi aritmetički nizovi.

Za dodatnih 30 bodova bilo je potrebno primijetiti da će najmanji broj iz niza biti i najmanji broj u jednom od redova. Sada, najmanji broj stavimo kao prvi broj u prvi red, fiksiramo neki broj iz niza koji ćemo staviti na drugo mjesto u prvom redu. Sada, kada znamo prvi i drugi broj prvog reda, znamo i razliku između svaka dva susjedna broja u prvom redu pa možemo dodavati broj po broj u prvi red tako dugo dok broj koji želimo dodati postoji u originalnom nizu te nakon svakog dodavanja provjerimo tvore li svi brojevi koje nismo stavili u prvi red aritmetički niz. Ako je tako, onda smo našli rješenje. Ako nakon isprobavanja svih mogućih brojeva za drugi broj u prvom redu nismo našli rješenje, onda ono ni ne postoji.

Vremenska složenost je  $\mathcal{O}(N^3)$ .

U stilu Boba Graditelja: “Mozemo li to popraviti? Naravno da mozemo!”

Potrebno je primijetiti da za prva dva broja u prvom redu ne trebamo isprobavati sve kombinacije (najmanji broj, neki broj iz niza), već samo tri kombinacije. Promotrimo tri najmanja broja iz niza, označimo ih s  $A \leq B \leq C$ . Sigurno znamo da, ako postoji rješenje, jedan od redova će početi s (*A*, *B*) ili (*A*, *C*) ili (*B*, *C*).

Vremenska složenost je  $\mathcal{O}(N^2)$ .

Za treći podzadatak možemo probati započeti jedan od redova na sva tri gore opisana načina, napraviti



niz od točno  $\frac{N}{2}$  brojeva i onda provjeriti tvori li ostalih  $\frac{N}{2}$  brojeva aritmetički niz.

Vremenska složenost je  $\mathcal{O}(N)$ .

Za sve bodove fiksirat ćemo na gore spomenuta tri načina početak jednog reda. Zatim ćemo dodavati broj po broj u taj red te provjeriti tvore li svi brojevi koji su ostali aritmetički niz. Tu provjeru moramo napraviti pametnije od naivnog rješenja for petljom. Održavat ćemo dva skupa. U jednom ćemo pamtit sve brojeve koji su ostali, a u drugom razlike između susjednih brojeva koji su u prvom skupu. Kada broj izbacimo iz prvog skupa, tj. dodamo ga u prvi red, onda moramo iz drugog skupa izbaciti razlike između njega i brojeva susjednih njemu, ali i ubaciti razliku između brojeva koji su mu bili susjedi jer su sada oni posatli susjedi. Kada imamo ta dva skupa i znamo kako ih održavati, možemo samo u skupu razlika provjeriti je li najveći broj jednak najmanjem. Ako jest, onda su svi brojevi u skupu jednaki, a to znači da je razlika između svaka dva susjedna broja koja su ostala jednaka, a to znači da ti brojevi tvore aritmetički niz, odnosno da smo pronašli rješenje. Sve operacije nad spomenutim skupovima može podržati kolekcija `std::set` u jeziku C++. Slične kolekcije postoje i u ostalim podržanim jezicima uz iznimku programskog jezika C.

Vremenska složenost je  $\mathcal{O}(N \log N)$ .

## Zadatak: Lampice

Pripremio: Tonko Sabolčec

Potrebno znanje: hashiranje, binarno pretraživanje, centroidna dekompozicija stabla

Prvi podzadatak moguće je riješiti na više načina. Opisat ćemo rješenje pomoću hashiranja koje će nam koristiti i za konačno rješenje. Ukorijenimo stablo u čvoru  $R$ . Zanimaju nas svi palindromski segmenti kojima je jedan kraj u čvoru  $R$ . Za svaki čvor  $X$  računamo dvije hash vrijednosti:

$$\begin{aligned}down_x &= x_0 B^{k-1} + x_1 B^{k-2} + \dots + x_{k-1} B^0 \\up_x &= x_0 B^0 + x_1 B^1 + \dots + x_{k-1} B^{k-1}\end{aligned}$$

Pri tome je  $x_0, x_1, \dots, x_{k-1}$  niz boja na putu od korijena  $R$  do čvora  $X$  ( $color(R) = x_0, color(X) = x_{k-1}$ ), a  $B$  vrijednost baze hashiranja. Ove dvije vrijednosti mogu se jednostavno odrediti jednim DFS obilaskom po stablu. Put od žaruljice  $R$  do žaruljice  $X$  je palindromski segment ako vrijedi  $down_x = up_x$ . Ako ovaj postupak ponovimo za svaki mogući korijen stabla, pokrili smo sve slučajeve. Složenost tog algoritma iznosi  $\mathcal{O}(N^2)$ .

Drugi podzadatak klasični je problem traženja najduljeg palindroma u nizu, koji se može riješiti pomoću [Manacherovog algoritma](#).

Treći podzadatak zapravo je jednak drugom podzadatku samo što Manacherov algoritam moramo primijeniti za svaki par listova u stablu, kojih zbog ograničenja ima dovoljno malo.

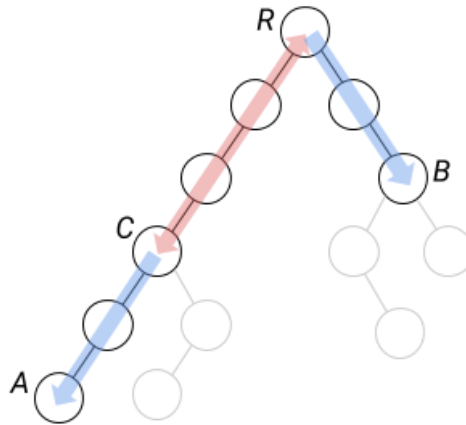
Potpuno rješenje započet ćemo sljedećom zamjedbom: Ako postoji palindromski segment duljine  $K > 2$ , tada postoji i palindromski segment duljine  $K - 2$ . Drugim riječima, traženu duljinu moguće je pronaći binarnim pretraživanjem za segmente parne i neparne duljine.

No, kako provjeriti postoji li palindrom određene duljine u nekom stablu? Kako bismo odgovorili na to pitanje, prvo ćemo riješiti lakšu verziju tog problema – provjerit ćemo postoji li palindrom zadane duljine koji prolazi korijenom (unaprijed određenim čvorom) stabla, čvorom  $R$ .

Na ukorijenjenom stablu, put između neka dva čvora  $A$  i  $B$  sastoji se od dvije grane na stablu, pri čemu je duljina jedne grane veća ili jednaka duljini druge grane. Pretpostavimo da se čvor  $A$  nalazi na duljoj grani. Put između čvorova  $A$  i  $B$  rastavit ćemo na 3 dijela: put od  $A$  do  $C$ , put od  $C$  do korijena stabla i put od korijena stabla do čvora  $B$ , pri čemu je  $C$  odabran tako da su duljine  $A - C$  i  $R - B$  jednake. Primijetite kako je  $C$  jedinstveno određen čvorom  $A$ , budući da nas zanimaju samo palindromi fiksne duljine. Kako bismo provjerili je li put od  $A$  do  $B$  palindromski segment, dovoljno je provjeriti sljedeće:



- (1) Put od  $R$  do  $C$  je palindrom (na slici označeno crveno)
- (2) Slijed boja na putu od  $C$  do  $A$  jednak je slijedu boja na putu od  $R$  do  $B$  (na slici označeno plavom).



Za svaki čvor stabla unaprijed odredimo hash vrijednosti  $down_X$  i  $up_X$  na isti način kao što je opisano u rješenju prvog podzadatka. Prvu provjeru jednostavno radimo uspoređivanjem vrijednosti  $down_C$  i  $up_C$ . Drugu provjeru moguće je napraviti usporedbom vrijednosti:

- $down_B$
- (3)  $down_A - down_{par(C)} B^{dep(A) - dep(C)}$ , gdje je  $par(C)$  roditelj čvora  $C$ , a  $dep(X)$  dubina čvora  $X$ .

Sada znamo efikasno provjeriti je li put od čvora  $A$  do čvora  $B$  palindromski segment, ali to i dalje nije dovoljno brzo jer postoji  $\mathcal{O}(N^2)$  mogućih parova  $(A, B)$ . Pokušajmo promatrati problem iz malo drugačijeg kuta: Ako nam je poznat čvor  $A$ , postoji li (barem jedan) čvor  $B$  koji zadovoljava navedena svojstva? Neka je  $S_B$  skup hasheva u koji ćemo staviti vrijednosti  $down_B$  svih čvorova  $B$  u promatranom stablu. Tada za neki čvor  $A$  možemo provjeriti postoji li odgovarajući čvor  $B$  tako da provjerimo uvjet (1) i postoji li hash vrijednost (3) u skupu  $S_B$ . Pretpostavlja se da je složenost operacija dodavanja vrijednosti u skup i provjeru postoji li neka vrijednost u skupu  $\mathcal{O}(1)$  korištenjem hash tablice (`std::unordered_set` u jeziku C++). Složenost cijele provjere iznosi  $\mathcal{O}(n)$ .

**Napomena:** prilikom implementacije potrebno je paziti da najviši zajednički predak čvorova iz skupa  $S_B$  i čvora  $A$  bude upravo korijen  $R$ .

Sada kada znamo postoji li palindromski segment određene duljine koji prolazi korijenom stabla  $R$ , možemo primijeniti centroidnu dekompoziciju. Ako za svako dekomponirano stablo napravimo provjeru s centroidom kao korijenom stabla, pokrit ćemo sve slučajeve, odnosno moći ćemo provjeriti postoji li palindrom određene duljine u cijelom stablu. Složenost takve provjere iznosi  $\mathcal{O}(n \log n)$ . Dodavanje razine binarnog pretraživanja dovodi do konačnog rješenja složenosti  $\mathcal{O}(n \log^2 n)$ .



## Zadatak: Sob

Pripremili: Paula Vidas i Daniel Paleka

Potrebno znanje: matematika, pohlepni algoritmi

Uređeni par  $(a, b)$  zvat ćemo *dobrim* ako vrijedi  $a \& b = a$ .

Prvi podzadatak možemo riješiti tako da  $a \in A$  uparimo sa onim  $b \in B$  za kojeg vrijedi  $b \bmod N = a$ .

Drugi podzadatak možemo riješiti sljedećim algoritmom: Neka su  $i_1 > i_2 > \dots > i_k$  pozicije jedinica u binarnom zapisu od  $N$ . Uparit ćemo najmanjih  $2^{i_1}$  elemenata skupova  $A$  i  $B$  tako da uparimo one  $a$  i  $b$  za koje vrijedi  $a \equiv b \bmod 2^{i_1}$ . Zatim uzmemo sljedećih  $2^{i_2}$  najmanjih elemenata i uparimo one koji su jednaki modulo  $2^{i_2}$ , itd. Dokaz da su odabrani parovi dobri ostavljamo čitatelju za vježbu.

Treći podzadatak mogao se riješiti na više načina. Jedan mogući način je da napravimo bipartitni graf sa čvorovima iz skupova  $A$  i  $B$  te dodamo bridove između svih dobrih parova. Na dobivenom grafu napravimo algoritam za uparivanje na bipartitnom grafu (*bipartite matching*), na primjer u složenosti  $\mathcal{O}(NE)$ , pri čemu je  $E$  broj bridova u grafu. Primjetimo da broj bridova možemo ograničiti sa  $E < 3^{10} = 59049$ .

Drugi način koristi sljedeći pohlepni algoritam: Prolazimo kroz elemente skupa  $A$  od većih prema manjima i trenutni element uparimo sa najmanjim još neuparenim elementom skupa  $B$  s kojim ga smijemo upariti.

Ako pokrenemo taj algoritam na nekoliko primjera, možemo uočiti sljedeću pravilnost: Neka se najveći element skupa  $A$ , tj.  $N - 1$ , upari sa  $b \in B$ . Tada se upare i  $N - 1 - t$  sa  $b - t$  za svaki  $t \in \{1, 2, \dots, b - M\}$ . Nakon što maknemo uparene elemente dobili smo isti zadatak, sada za skupove  $A' = \{0, 1, \dots, N - 1 - (b - M) - 1\}$  i  $B' = \{b + 1, b + 2, \dots, M + N - 1\}$ . Ovo rješenje možemo implementirati u složenosti  $\mathcal{O}(N)$ .

Dokaz prethodne tvrdnje:

Neka je  $a = N - 1$  (radi ljepših oznaka) i  $b$ , kao i prije, najmanji element skupa  $B$  za kojeg vrijedi  $a \& b = a$ . Indeksom  $i$  ćemo označavati znamenku težine  $2^i$ . Ako je  $b = M$  nemamo što za dokazivati, pa pretpostavimo da je  $b > M$  i označimo  $k = b - M$ . Neka je  $i$  pozicija najmanje značajne jedinice u  $b$ . Očito mora biti  $a_j = b_j = 0$  za  $j < i$ . Kada bi bilo  $a_i = 0$  onda bi vrijedilo  $a \& (b - 1) = a$  pa  $b$  ne bi bio najmanji element od  $B$  koji se može upariti sa  $a$ . Dakle,  $a_i = b_i = 1$ . Sada je očito da je  $(a - t, b - t)$  dobar par za  $t \in \{1, 2, \dots, 2^i\}$ . Ako je  $k \leq 2^i$  gotovi smo, inače promatramo sljedeću najmanje značajnu jedinicu u  $b$  i induktivno ponavljamo isti postupak. Preostaje još pokazati da uvijek postoji neki  $b \in B$  s kojim se  $a$  može upariti, no to ostavljamo čitateljici za vježbu.