



## Opisi algoritama

Zadatke, testne primjere i rješenja pripremili: Vito Anić, Fran Babić, Toni Brajko, Jakov Celin, Nikola Dmitrović, Josip Klepec i Iva Tadić.

Primjeri implementiranih rješenja dani su u priloženim izvornim kodovima.

### Zadatak SAD

Pripremio: Nikola Dmitrović

Potrebno znanje: naredba odlučivanja

Nakon što učitamo četiri zadana podatka s ulaza, provjerimo tko je pobijedio u državi C. Ovisno tko je pobijedio, povećamo broj osvojenih glasova D ili K za CG te ispišemo odgovor na prvo pitanje. Završno, provjerimo tko je imao više glasova na kraju te ispišemo odgovor na drugo pitanje.

Programski kod:

```
AG = int(input())
BG = int(input())
tko = int(input())
CG = int(input())

D = AG
K = BG

if tko == 1:
    print("DONALD")
    D += CG
else:
    print("KAMALA")
    K += CG

if D > K:
    print("DONALD")
else:
    print("KAMALA")
```

### Zadatak Akcija

Pripremio: Fran Babić

Potrebno znanje: naredbe ponavljanja (petlje), naredbe odlučivanja

Kako će Vlatko dobiti svaki 5. dan besplatan proizvod, njegovu cijenu ne trebamo uključiti u zbroj cijena koje smo platili. Potrebno je odrediti koje ćemo proizvode platiti tj. njihovu cijenu zbrojiti u konačan odgovor, a koje nećemo.

Za to možemo koristiti pomoćnu varijablu koju ćemo inkrementirati nakon svakog učitanoj broja, ali nakon što je peti put povećamo znamo da taj broj ne zbrajamo i *resetiramo* pomoćnu varijablu. Jedna od mogućih implementacija tog algoritma je:



```
n = int(input())

cijena = 0
brojac = 0

for i in range(n):
    x = int(input())
    brojac += 1
    if brojac == 5:
        brojac = 0
    else:
        cijena += x

print(cijena)
```

## Zadatak Paradoks

Pripremla: Iva Tadić

Potrebno znanje: polja, for petlja

Cilj ovog zadatka je uredno i pametno spremiti podatke zadane u zadatku tako da se s njima što lakše manipulira.

Kako nam je općenito lakše raditi s brojevima nego stringovima, označimo naše igrače s brojevima 1 do 5, redom kojim bi igrali kad bi Sonja započinjala rundu (dakle Sonju označimo s 1, Viktora s 2, itd.) Također označimo i boje s brojevima 1 do 4 (gdje je crvena boja „boja 1“, plava „boja 2“, žuta „boja 3“ i zelena „boja 4“).

Paradoksi mogu biti uzrokovani na 2 načina - karta se već pojavila u igri i/ili igrač nema tu boju u ruci. Pratit ćemo ta stanja kroz 2 polja:

- *se\_pojavila\_karta*[*x*][*y*], koje će u slučaju da je već odigrana karta boje *x* s brojem *y* iznositi 1, a u suprotnom 0.
- *ne\_smije\_koristiti\_boju*[*x*][*y*], koje će u slučaju da igrač *x* ne smije koristiti boju *y* do kraja igre iznositi 1, a u suprotnom 0.

Primijetimo da će nam na početku igre sve vrijednosti oba polja biti 0 (nijedna karta nije odigrana i svim igračima su sve boje dozvoljene).

Sada dalje simuliramo igru, red po red unosa (rundu po rundu igre).

Pratiti ćemo: boju runde, koji je trenutno igrač na potezu, tko zasad pobjeđuje rundu, i koja je boja i broj karte koja zasad nosi pobjedu u toj rundi. Paradokse ćemo spremiti u vektor sa strane, kako nađemo na njih - kronološki.

Na početku runde, kao pobjedničku kartu postavljamo prvu kartu iz runde, te kao trenutnog pobjednika runde igrača koji je započeo rundu.

Za svaku iduću kartu iz runde, prvo provjeravamo radi li se o paradoksu koristeći već gore navedena polja:

- Ako se radi o paradoksu, cijelo to bacanje ignoriramo (ne spremamo nikakve nove informacije) a informacije o ovom paradoksu šaljemo u naš vektor paradoksa.  
(Napomena: preskačemo i ažuriranje pobjednika te krećemo analizirati iduće bacanje u rundi).
- Ako se pak ne radi o paradoksu, spremamo informaciju da se ta karta pojavila (update polja *se\_pojavila\_karta* ).  
Ako bačena karta nije boje runde, zaključujemo da igrač na potezu nema kartu boje te runde u ruci te postavljamo *ne\_smije\_koristiti\_boju*[*igrac\_na\_potezu*][*boja\_runde*]=1.



Nadalje, uspoređujemo trenutnu kartu s zasad najjačom kartom koja je bačena u rundi. Ako je trenutna jača, ažuriramo pobjedničkog igrača i postavljamo ovu kartu za zasad najjaču bačenu kartu u rundi. Sada promatramo iduće bacanje u toj rundi...

Nakon svake runde resetiramo varijable pobjedničkog igrača i najjače karte. Ovaj postupak ponavljamo dok nismo obradili i zadnju rundu.

Na kraju gledamo koliko smo paradoksa našli (koliko elemenata nam sadrži vektor u koji smo spremali paradokse) te paradokse ispišemo redom.

Za detalje implementacije pogledati priloženi kod.

## Zadatak Igre

Pripremio: Vito Anić

Potrebno znanje: dinamičko programiranje - problem naptrnjače (*Knapsack problem*)

Riješimo prvo treću parcijalu  $p_i = 0$ . Zadatak možemo onda pojednostaviti: Imamo ruksak (naptrnjaču) u koju stane  $d$  kilograma stvari. Imamo  $n$  stvari, svaka ima svoju težinu i vrijednost. Želimo da je ukupna vrijednost spremljenih stvari u ruksak što veća. Svaku stvar možemo uzeti i više od jedanput u ruksak. To je klasičan primjer iz dinamičkog programiranja koji se poznato zove *Knapsack problem*. Opišimo rješenje ukratko.

Stanje dinamike će biti  $dp(x, k)$  gdje  $x$  označuje koliko stvari smo dosada razmotrili za uzimanje a  $k$  koliko ruksaka smo dosada popunili. Prijelaz dinamike će biti:

$$dp(x, k) = \max(dp(x, k-1), dp(x-1, k), dp(x, k-t_x) + o_x)$$

U prijevodu, ili smo  $k$ -to mjesto u ruksaku ostavili prazno (možda ga želimo iskoristiti za neki kasniji predmet) ili ćemo svih  $k$  mjesta popuniti s jednim predmetom manje ili ćemo uzeti  $x$ -ti predmet (doajemo na ukupnu vrijednost  $o_x$ ) i s  $k-t_x$  mjesta popuniti ruksak s nekim ostalim predmetima (moguće opet i istim). Prolaskom po svim  $x$  od 1 do  $n$  i po svim  $k$  od 1 do  $d$  nam daje rješenje za ovu parcijalu. Za one koje žele znati više, detaljnije objašnjen knapsack i dodatno neke varijacije njega mogu se pronaći na [poveznici](#).

Riješimo sada cijeli zadatak. Problem do kojega dolazimo jest da nama prije prvog igranja igre potrebno je uložiti vrijeme za čitanje pravila. Trenutna dinamika nikako ne može odvojiti prvi put igranja igre i ostale. Kako to riješiti? Dodati ćemo još jednu varijablu u stanje. novo stanje će biti  $dp(x, k, 0$  ili  $1)$  gdje  $x$  i  $k$  imaju isto značenje kao prije ( $x$  - koje igre dosad razmatramo,  $k$  - koliko vremena smo dosad uložili), dok ovaj zadnji bit će biti 1 ako smo naučili pravila te igre a 0 ako nismo. Prijelazi su tada:

$$dp(x, k, 0) = \max(dp(x-1, k, 0), dp(x-1, k, 1))$$

$$dp(x, k, 1) = \max(dp(x, k-p_i, 0), dp(x, k-t_i, 1) + o_x)$$

Ukratko objašnjeno, ako nismo još naučili pravila, ne možemo ništa osim trenutno vrijeme potrošiti na prethodne igre, a ako jesmo, mogli smo ih naučiti u zadnjih  $p_x$  minuta ili, pod uvjetom da smo prije  $t_x$  minuta znali pravila, možemo igrati igru. Vremenska složenost algoritma:  $O(nd)$ .

## Zadatak Različitost

Pripremio: Fran Babić

Potrebno znanje: ad-hoc, modularna aritmetika

Za prvi podzadatak dovoljno je bilo iterirati po prvim  $k$  parova  $a_i \oplus b_i$  jer je  $k$  dovoljno malen da algoritam složenosti  $O(k)$  prođe u zadanom vremenskom ograničenju.

Drugi i treći podzadatak rješavamo tako da svedemo problem na ciklički niz duljine  $m$ ,  $c_i = a_1 \oplus b_i$



$(c_i = a_i \oplus b_i)$  za svaki  $i = 1, 2, \dots, m$ . Primjećujemo da je traženi zbroj  $\sum_{i=1}^k c_i$ , što se može relativno jednostavno riješiti.

Za ostvariti sve bodove na zadatku, korisno je primijetiti da je ovaj zbroj nezavisan po znamenkama u binarnom zapisu tj.

$$\sum_{i=1}^k a_i \oplus b_i = \sum_{j=0}^{\infty} \left( \sum_{i=1}^k a_{i,j} \oplus b_{i,j} \right) 2^j$$

gdje je  $a_{i,j}$  oznaka za znamenku u binarnom zapisu koja pripada potenciji  $2^j$  i analogno za  $b_{i,j}$ . Ovime smo pojednostavili zadatak tako što ga svodimo na nizove koji se sastoje od 0 i 1.

Sada ćemo za svaki element iz prvog niza zbrojiti sve *bit-po-bit isključivo* ili parova u kojima se taj element nalazi, ali to se, zbog ograničenja na nove nizove, svodi na zbroj svih elemenata s kojima će on biti ili zbroj komplementa istih. Jedino što sad preostaje je odrediti te elemente i njihovu zbroj.

Za to nam je potrebna ključna opservacija. Neka je  $g = \gcd(n, m)$  najveći zajednički djelitelj  $n$  i  $m$ . Za jako velike brojeve  $k$ , traženi elementi drugog niza, za fiksirani element iz prvog niza, čine ciklus duljine  $\frac{m}{g}$ . Također, ukupan broj različitih ciklusa (do na cikličku rotaciju) će biti  $g$ . Spretnom implementacijom ovog algoritma u složenosti  $O((n + m) \log m)$  se ostvaruju svi bodovi na zadatku.

## Zadatak Blistavost

Pripremili: Toni Brajko i Jakov Celin

Potrebno znanje: dinamičko programiranje

**Opservacija 1** *Ako na brojevnom pravcu označimo sve točke  $l_i, r_i$  te točku 0, primjećujemo da postoji optimalni put kojim će čuvar ići, a da ga možemo rastaviti na dijelove čiji su rubovi označene točke. U opisu rješenja navedene točke ćemo nazivati **bitnima** te ćemo ih sortirati tj.  $i$ -tu bitnu točku u sortiranom poretku ćemo nazivati  $T_i$ .*

Zadatak rješavamo dinamičkim programiranjem. Kako je u prvom podzadatku  $n \leq 18$ , neka je  $dp[maska][j]$  najmanje vrijeme potrebno da ispunimo sve uvjete za koje su bitovi u maski upaljeni te da se na kraju puta nalazimo na točki tj. Fiksiranjem maske, *bitne* točke te promatranjem jednog od uvjeta lako dolazimo do rješenja čija je vremenska složenost  $O(2^n \cdot n^2)$ .

U drugom podzadatku svaki od raspona uvjeta započinje u točki 1. Može se pokazati da će u ovom podzadatku čuvar ići s lijeva nadesno ili će čuvar otići do najdesnije točke u među zadanim rasponima, po potrebi pričekati određeno vrijeme te ići nalijevo do točke 1. Dokaz ostavljamo čitatelju za vježbu. Vremenska složenost algoritma je  $O(n \log n)$  zbog sortiranja točaka.

**Opservacija 2** *Ako na svaku poziciju zapišemo zadnji trenutak u kojemu smo ju posjetili te popis pozicija zapamtimo kao niz  $P$ , zadatak čuvara je osmisliti put takav da za svaki od uvjeta stanovnika vrijedi da je  $\min(P[j]) \geq t_i$  za  $l_i \leq j \leq r_i$ .*

**Opservacija 3** *Ako čuvar put promatramo obrnuto ("odiza"), čuvar proširuje interval posjećenih točaka naizmjenično nalijevo i nadesno te se unutar intervala već posjećenih točaka može kretati bez da ponovo zapisuje kada ih je opet posjetio.*

Neka je  $dp(l, r, 0)$  najmanje vrijeme da su za niz  $P$  ispunjeni svi uvjeti stanovnika ako samo promatramo točke izvan intervala  $[T_l, T_r]$ . Neka je  $dp(l, r, 1)$  najmanje vrijeme da su za niz  $P$  ispunjeni svi uvjeti stanovnika ako samo promatramo točke izvan intervala  $[T_l, T_r]$ .

Pomoću svih opservacija lako je za pokazati da iz bilo kojeg stanja  $dp(l, r, g)$  postoje 4 stanja u koje ono direktno prelazi:  $dp(l, r - 1, 0)$ ,  $dp(l, r - 1, 1)$ ,  $dp(l + 1, r, 0)$ ,  $dp(l + 1, r, 1)$ . Ako intervale obilazimo od većih



duljina ka manjima, rješenje dobivamo uzimajući najmanju vrijednost  $dp(i, i, 0)$  za svaki indeks  $i$  bitne točke. Vremenska i memorijska složenost navedenog algoritma je  $O(n^2)$ . Izravna implementacija iznad objašnjenog algoritma donosila je 90 bodova zbog memorijskog ograničenja zadatka.

Za potpuno rješenje zadatka, bilo je potrebno primijetiti da svako stanje  $dp(l, r, 0/1)$  prelazi u stanja čiji su intervali indeksa za 1 kraći od intervala  $[l, r]$ . Pažljivom implementacijom postiže se memorijska složenost  $O(n)$ , a vremenska složenost ostaje nepromijenjena.

## Zadatak Trokuti

Pripremio: Josip Klepec

Potrebno znanje: pohlepni algoritam, ad-hoc

Zadaci ovog tipa mogu se riješiti [flow](#) algoritmom, pomoću kojeg ako na dobar način postavimo mrežu možemo pronaći i svih  $2 \cdot n$  trokuta.

Originalna zamišljena ideja rješenja ovog zadatka bila je pomoću pohlepnog algoritma koristeći argument zamjene.

U sljedećem odlomku opisat ćemo ideju. Naivno konstruiramo trokute jedan po jedan, koristeći samo čvorove koje do sad već nismo iskoristili. Ponavljajući ovaj postupak dobivamo određen broj trokuta koji može biti manji od  $n$ . Točnije dobit ćemo barem  $\lfloor \frac{2}{3} \cdot n \rfloor$  trokuta. U slučaju da nismo dobili dovoljan broj trokuta više od  $3 \cdot n$  (pola) čvorova je još uvijek neiskorišteno.

Promatrajući nekih  $2 \cdot n$  disjunktih trokuta na koje se graf može podijeliti zaključujemo da niti jedan od njih ne sadrži 3 do sada neiskorištena čvora, jer bi u suprotnom mogli pohlepno načiniti još jedan trokut. Nadalje, pošto imamo samo  $2 \cdot n$  trokuta, a  $3 \cdot n$  neiskorištenih čvorova, zaključujemo da postoji barem  $n$  disjunktih trokuta koji sadrže barem 2 do sada neiskorištena čvora.

Promatrajući tih  $n$  trokuta i njihov treći već iskorišteni čvor zaključujemo da postoje neka 2 trokuta čiji su iskorišteni čvorovi u istom već pohlepno izgrađenom trokutu. Ono što možemo napraviti je uništiti taj trokut i zamijeniti ga s ova nova 2 trokuta. Navedenu logiku možemo implementirati u  $O(n^4)$  vremenu. Algoritam je moguće ubrzati na  $O(n^3)$  pamteći parove neiskorištenih čvorova koji mogu formirati trokut za svaki već iskorišteni čvor, ovo možemo efikasno održavati ubacivanjem i zamjenama trokuta u svakom dijelu algoritma, no ovo na kraju nije bilo potrebno.

Međutim, ispostavlja se da vjerojatnost da ćemo u pohlepnom algoritmu dobiti barem  $n$  trokuta teži 1 kada  $n$  teži beskonačnosti. Također lako je pokazati da je donja ograda za očekivani broj trokuta koji ćemo dobiti veća od  $n$  promatrajući slučajne grafove s 3 disjunktne trokuta.

Zadatak se mogao riješiti tako da za male vrijednosti broja  $n$  napravimo brute-force algoritam ili ponovimo pohlepni algoritam mnogo puta, dok ćemo za veće  $n$  uvijek dobiti  $n$  trokuta.