

Opisi algoritama

Primjeri implementiranih rješenja dani su u priloženim izvornim kodovima koji ne odgovaraju nužno u svim detaljima ovdje opisanim algoritmima.

Eventualna pitanja, mišljenja ili prijedloge vezane uz zadatke uputite na adrian.kurdija@fer.hr.

Zadatak: Mjere

U ovom zadatku dovoljno je unutar zadane matrice znakova izbrojiti pravokutnike sljedećeg oblika:

```
....  
.##.  
....
```

To možemo ostvariti dvostrukom for petljom koja bira početno polje (i, j) tog pravokutnika u matrici te provjerava čemu je jednak dio i -tog retka od polja (i, j) do polja $(i, j + 3)$, dio $(i + 1)$. retka od polja $(i + 1, j)$ do polja $(i + 1, j + 3)$, te dio $(i + 2)$. retka od polja $(i + 2, j)$ do polja $(i + 2, j + 3)$.

Zadatak: Konj

Dvostrukom for petljom prolazimo po zadanoj ploči. Ako je polje slobodno, iz njega puštamo rekursiju (DFS algoritam) koja obilazi i broji sva polja koja je moguće doseći skakačem iz tog početnog polja, te dohvaćena polja postavlja na 0 da ih ne bismo beskonačno obilazili. Za implementacijske detalje pogledajte priloženi kod.

Zadatak: Anagram

Slova zadane riječi sortiramo po abecedi. Zatim punimo traženi anagram od početnog do završnog slova na sljedeći način: u anagram stavljamo prvo *dopušteno* slovo od još neiskorištenih slova zadane riječi. Slovo je dopušteno ako nije jednako prethodno stavljenom slovu i ako, nakon njegovog stavljanja u anagram, najčešće preostalo slovo možemo rasporediti u ostatak anagrama na nesusjedna mjesta, tj. ako se svako preostalo slovo javlja najviše $\lfloor (k + 1)/2 \rfloor$ puta gdje je k broj preostalih znakova koje treba staviti u anagram. Ako nema dopuštenog slova koje možemo staviti, ispisujemo -1 .

Primjerice, u drugom probnom primjeru (aabbcc) u anagram stavljamo redom: slovo **a**, slovo **b** (jer **a** je jednak prethodnom slovu), slovo **a**. Anagram je sada **aba...** i preostala su slova **bcc**. Ne možemo staviti **b** jer bi onda preostalih slova **c** bilo previše, tj. ne bismo ih mogli rasporediti na dva preostala mjesta u anagramu bez susjedstva; zato biramo dalje i stavljamo slovo **c**, nakon čega redom stavljamo **b** i **c**.

Dokaz točnosti algoritma ide ovako. Ako rješenje ne postoji, očito ga ni naš algoritam neće pronaći jer on ne gradi nedopušteni anagram. Pretpostavimo sada da je točno rješenje anagram A . Tvrdimo da naše rješenje pronalazi isti anagram. Promatrajmo i -to slovo rješenja ($A_i = x$) pretpostavljajući da nam se prethodna slova podudaraju. Naš algoritam neće na i -to mjesto staviti neko slovo manje od x jer bi u tom slučaju postojao abecedno manji anagram od A (primjerice, onaj koji dobijemo daljnjim stavljanjem uvijek najčešćeg slova koje možemo staviti – ono će tako i dalje uvijek biti dopušteno). Također je x sigurno *dopušteno* slovo u kontekstu našeg algoritma, jer inače anagram A ne bismo mogli dovršiti. Dakle, naš će algoritam staviti slovo x na i -to mjesto.

Zadatak: Kruno

Dvostrukom for petljom biramo polje (i, j) u matrici te brojimo koliko susjednih sjedala (u osam smjerova) imaju polja (i, j) i $(i + 1, j)$, ili u drugom slučaju polja (i, j) i $(i, j + 1)$. Ako je svako od tih dvaju polja sjedalo koje ima točno jedno susjedno sjedalo (a taj susjed je onda upravo ovo drugo polje i više ih nema), riječ je o ispravnom paru sjedala pa povećavamo rješenje za jedan.

Zadatak: Skok

Dvostrukom for petljom prolazimo po zadanoj ploči. Ako je polje slobodno, iz njega puštamo rekursiju (DFS algoritam) koja obilazi i zbraja bombone na svim poljima koja je moguće doseći skakačem iz tog početnog polja, te dohvaćena polja postavlja na 0 da ih ne bismo više puta brojili. Za implementacijske detalje pogledajte priloženi kod.

Zadatak: Flip

Nešto jednostavnija inačica ovog zadatka (bez desnog flipa) pojavila se na HONI natjecanju 2012. godine kao zadatak DNA ([link](#)). Čitateljici predlažemo da najprije prouči opis algoritma za taj zadatak ([link](#)).

Zadatak rješavamo dinamičkim programiranjem. Neka $dp(i, r, o)$ predstavlja minimalan broj instrukcija potreban da prvih i bitova postavimo sve na $r \in \{0, 1\}$, pri čemu nam parametar $o \in \{0, 1\}$ govori jesmo li pritom učinili neku instrukciju desnog flipa (točnije, neparan broj tih instrukcija) koja će mijenjati iduće bitove nakon i -tog.

Početne vrijednosti su $dp(0, r, o) = o$ jer su svi od prvih nula bitova već jednaki r te samo činimo ili ne činimo početnu instrukciju desni flip 0.

Pretpostavimo da je i -ti bit jednak x i želimo izračunati $dp(i, r, o)$.

Najprije, ako je $o = 1$, imamo "otvoreni" desni flip koji mijenja $x := 1 - x$.

Potom, ako je $x = r$, on je "dobar" pa nas samo zanima prethodni dio, tj. $dp(i, r, o) = dp(i - 1, r, o)$.

Ako je pak $x \neq r$, imamo tri mogućnosti:

- single flip i -tog znaka, što daje $1 + dp(i - 1, r, o)$ instrukcija;
- lijevi flip do i -tog znaka, što daje $1 + dp(i - 1, 1 - r, o)$ instrukcija jer prethodni dio onda mora biti postavljen na $1 - r$;
- novi desni flip od i -tog znaka, što mijenja parametar o i daje $1 + dp(i - 1, r, 1 - o)$ instrukcija;

od čega minimum postaje $dp(i, r, o)$.

Na kraju ispisujemo manju od vrijednosti $dp(n, 0, 0)$ i $dp(n, 0, 1)$.