

Opisi algoritama

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

Zadatak: Baza

autor: Paula Vidas

Algoritam za pretvorbu broja u neku negativnu bazu zapravo je skoro isti kao “običan” algoritam za pretvorbu u pozitivnu bazu, s kojim ste se možda susreli npr. za pretvorbu u bazu 2.

Zadnja znamenka zapisa od x u bazi -10 jednaka je (nenegativnom) ostatku pri dijeljenju broja x s -10 . Broj x zatim cjelobrojno podijelimo s -10 , pri čemu zaokružujemo “na više”, te ponavljamo postupak dok ne dobijemo $x = 0$.

Za 16 bodova, ovaj algoritam se može izravno implementirati, koristeći operacije na brojevima. Za sve bodove potrebno je analizirati što se događa u koracima ovog algoritma, kako bismo ga prilagodili tome da x može imati do 10^6 znamenki.

Ako je trenutni x pozitivan (što se događa u prvom, trećem, petom, itd. koraku), onda je nova znamenka jednaka zadnjoj znamenici od x . Dijeljenje s -10 zapravo znači da “odbacujemo” zadnju znamenku od x (te stavljamo negativni predznak).

Ako je trenutni x negativan (što se događa u drugom, četvrtom, itd. koraku), razmatramo dva slučaja. Ako je zadnja znamenka od x jednaka 0, onda je nova znamenka isto 0, a dijeljenje s -10 je samo odbacivanje zadnje znamenke. Ako je zadnja znamenka d različita od 0, onda je nova znamenka jednaka $10 - d$. Dijeljenje s -10 u ovom slučaju znači da odbacujemo zadnju znamenku, te na preostalo dodajemo 1.

Za bolje razumijevanje čitatelju preporučujemo da raspiše na papiru potrebne detalje, a za implementaciju pogleda službeni kod.

Alternativno, mogli smo razmišljati ovako: vrijedi $10^i = (-10)^i$ ako je i paran, odnosno $10^i = -(-10)^i$ ako je i neparan. Stoga ako znamenke (u bazi 10) od x na neparnim pozicijama pomnožimo s -1 , a na parnim ostavimo iste, dobili smo nekakav zapis od x u bazi -10 . Taj zapis nije nužno “ispravan”, neke znamenke su možda negativne. Zato idemo “od kraja” (tj. od najmanje značajne znamenke, one uz $(-10)^0$), i “popravljamo” trenutnu znamenku.

Neka je znamenka d na poziciji i negativna. Primjetimo da vrijedi

$$d \cdot (-10)^i = (d + 10 - 10) \cdot (-10)^i = (d + 10) \cdot (-10)^i + (-10)^{i+1}.$$

Dakle, na znamenku na poziciji i dodajemo 10, a na poziciji $i + 1$ dodajemo 1. Tako će znamenka na poziciji i postati nenegativna.

Pritom se moglo dogoditi da neke znamenke postanu veće ili jednake 10. Slično kao prije, vrijedi

$$d \cdot (-10)^i = (d - 10 - 90 + 100) \cdot (-10)^i = (d - 10) \cdot (-10)^i + 9 \cdot (-10)^{i+1} + (-10)^{i+2}.$$

Dakle, znamenku na poziciji i možemo smanjiti za 10, a znamenke na pozicijama $i + 1$ i $i + 2$ povećati za 9 odnosno 1.

Zadatak: Likovi

autor: Adrian Satja Kurdija

Možemo pretpostaviti da dio C nećemo rotirati i pomicati, nego ćemo ga postaviti fiksno u sredinu neke dovoljno velike matrice znakova, a onda ćemo dijelove A i B na sve moguće načine pokušati položiti s obzirom na taj fiksni dio C .

Preciznije: C ćemo "upisati" u srednji $m \times m$ dio velike matrice od $(n+m+n) \times (n+m+n)$ točaka. Onda ćemo za svaku moguću (od četiri) rotacije dijela A , za svaku moguću (od četiri) rotacije dijela B , za svaki mogući položaj gornje-lijeve točke od A unutar velike matrice, te za svaki mogući položaj gornje-lijeve točke od B unutar velike matrice, položiti A i B na odabrani način, provjeriti ima li preklapanja između A i B te ako nema, izbrojiti koliko se njihovih puzzli ne podudara s puzzlama dijela C , pamteći najbolji rezultat. Za implementacijske detalje pogledajte priloženi izvorni kod.

Zadatak: Scamazon

autor: Ivan Paljak

Opisat ćemo najprije kako pristupiti varijantama zadatka za koje vrijede dodatna ograničenja opisana u poglavlju o bodovanju.

U najlakšoj varijanti (vrijednoj 5 bodova), tekst zadatka nam garantira da vrijedi $c_i = c_j$ za sve $1 \leq i, j \leq k$, dakle sve su novčanice međusobno jednake. U tom je slučaju optimalno *trpati* artikle u košaricu sve dok ukupna cijena košarice ne premaši vrijednost jedne novčanice. Kada se to dogodi, potrebno je potrošiti jednu novčanicu i primijeniti isti pohlepni algoritam na preostale artikle. Ovaj jednostavan algoritam moguće je implementirati u vremenskoj složenosti $\mathcal{O}(n)$, a izvorni kod jedne takve implementacije možete pronaći u datoteci `scamazon_jednak_c.cpp`.

Situacija je nešto složenija u varijanti (vrijednoj 10 bodova) gdje vrijedi $a_i = a_j$ za sve $1 \leq i, j \leq n$. Odnosno, u ovoj su varijanti svi artikli jednako vrijedni, ali vrijednosti novčanica koje imamo na raspolaganju mogu biti različite. Dakako, sada je važno koje ćemo novčanice iskoristiti, ali nije važno kojim ćemo ih redoslijedom koristiti (dokaz ostavljamo čitatelju, hint: [exchange argument](#)). Budući da je dovoljno odrediti samo podskup novčanica, a cijeli skup ima najviše 10 elemenata, zaključujemo da možemo iscrpnom pretragom pronaći taj optimalan podskup. Jednom kada fiksiramo skup novčanica, jednostavno je provjeriti (sličnom metodom kao iz prethodnog odlomka) je li moguće kupiti sve željene artikle. Ograničenja dopuštaju i manje efikasne algoritme, ali vješti natjecatelji će ovaj algoritam najvjerojatnije implementirati u vremenskoj složenosti $\mathcal{O}(2^k)$, a izvorni kod jedne takve implementacije možete pronaći u datoteci `scamazon_jednak_a.cpp`.

Za osvajanje dodatnih bodova više se nismo mogli osloniti na ograničenja nad vrijednostima nizova a i c , ali za 30 bodova vrijedila su nešto opuštenija ograničenja na njihove veličine. Iskusnije natjecatelje će sam pogled na te vrijednosti ($n \leq 1\,000$ i $k \leq 10$) odmah navesti na pravi put – dinamičko programiranje i bitmaske. Dobar način razmišljanja u ovakvim situacijama je zamisliti da uspješno gradite optimalno rješenje te razmislite koje vam informacije trebaju da biste dalje nastavili graditi to optimalno rješenje. Primijenimo li taj način razmišljanja na ovaj zadatak, brzo ćemo zaključiti da bi bilo zgodno "zaustaviti se" u trenutku kada smo već potrošili neke novčanice i imamo praznu košaricu, a u tom trenutku bilo bi korisno znati broj artikala koje smo kupili i koje smo novčanice potrošili. Stoga, neka nam $dp(i, m)$ govori koliko nam najviše novaca može ostati ako smo kupili prvih i artikala koristeći novčanice iz [bitmaske](#) m . Da bismo odredili tu vrijednost, potrebno je na sve moguće načine pretpostaviti koju ćemo sljedeću novčanicu iskoristiti, a u obzir dolaze samo one novčanice koje na odgovarajućem mjestu u bitmaski nemaju postavljen bit. Nakon što fiksiramo novčanicu, kupit ćemo najveći mogući broj artikala počevši od $(i+1)$ -og i tako preći u novo stanje rekurzije. Formalnije,

$$dp(i, m) = \max_{1 \leq j \leq k} \{dp(i', m + 2^j) \mid bit(j, m) = 0\}$$

gdje i' označava indeks artikla do kojeg je moguće kupovati koristeći j -tu novčanicu, a $bit(x, y)$ označava vrijednost x -tog bita u binarnoj reprezentaciji broja y . Budući da imamo ukupno $n \cdot 2^k$ različitih stanja te da trošimo $\mathcal{O}(k)$ vremena na prijelaz, ukupna složenost ovog pristupa, koristeći dinamičko programiranje, jest $\mathcal{O}(nk2^k)$. Izvorni kod ove implementacije možete pronaći u datoteci `scamazon_nk_puta_2_na_k.cpp`.

Za osvajanje svih bodova na zadatku bilo je potrebno osmisliti dovoljno efikasan algoritam za nešto stroža ograničenja nad ulaznim podacima ($n \leq 200\,000$ i $k \leq 20$). Primijetimo da na neki način postoji *intimna*

veza između dvaju dimenzija u stanju rekurzije iz prošlog odlomka. Odnosno, slutimo da bismo na neki način mogli na temelju maske m jedinstveno odrediti jedini *smislen* i . Dakako, zapravo nas zanima koliko najviše artikala možemo kupiti koristeći neki podskup danih novčanica. Kada bismo znali tu vrijednost izračunati za svaki podskup, bilo bi dovoljno pronaći podskup najmanje ukupne vrijednosti pomoću kojeg je moguće kupiti sve artikle. Posegnimo ponovno za dinamičkim programiranjem, neka je $dp(m)$ najdulji prefiks artikala koje možemo kupiti koristeći novčanice iz maske m . Tada vrijedi:

$$dp(m) = \max_{1 \leq j \leq k} \{f(dp(m - 2^j), c_j) \mid \text{bit}(j, m) = 1\}$$

gdje $f(i, c)$ određuje najveći $j \geq i$ takav da je pomoću novčanice vrijednosti c moguće kupiti artikle s indeksima $(i + 1), (i + 2), \dots, j$. Naizgled nismo ništa poboljšali, sada imamo 2^k različitih stanja, a naivnom implementacijom funkcije f na njega trošimo $\mathcal{O}(nk)$ vremena.

Međutim, jednostavnim dosjetkama moguće je znatno efikasnije implementirati funkciju f . Izgradimo najprije niz prefiks suma p nad vrijednostima artikala, odnosno, neka vrijedi $p_i = \sum_{1 \leq j \leq i} a_j$. Funkcija $f(i, c)$ zapravo traži najveći j za koji vrijedi $p_j - p_i \leq c$. Budući da su c i p_i konstante, a niz p monotonno raste, moguće je pomoću binarnog pretraživanja pronaći vrijednost $f(i, c)$ u $\mathcal{O}(\log n)$ vremena. Ukupna vremenska složenost cijelog algoritma sada iznosi $\mathcal{O}(k2^k \log n)$.

Zadatak: Slaganje

autor: Adrian Satja Kurdija

Možemo pretpostaviti da dio B nećemo rotirati i pomicati, nego ćemo ga postaviti fiksno u sredinu neke dovoljno velike matrice znakova, a onda ćemo dio A na sve moguće načine pokušati položiti s obzirom na taj fiksni dio B .

Preciznije: B ćemo "upisati" u srednji $n \times n$ dio velike matrice od $3n \times 3n$ točaka. Onda ćemo za svaku moguću (od četiri) rotacije dijela A , za svaki mogući položaj gornje- lijeve točke od A unutar velike matrice, položiti A na odabrani način te izbrojiti koliko se njegovih puzzli ne podudara s puzzlama dijela B , pamteći najbolji rezultat. Za implementacijske detalje pogledajte priloženi izvorni kod.

Zadatak: Hulje

autor: Ivan Paljak

Čak petinu bodova na ovom zadatku bilo je moguće osvojiti algoritmom koji pretpostavlja da se u ulaznim podacima nalazi trokut ili četverokut. Trokut predstavlja trivijalan slučaj kojeg je optimalno pokriti samim sobom, dok je lagano uočiti da je četverokut optimalno pokriti koristeći neka 2 od 4 različita trokuta određenih ulaznim točkama. Odnosno, sve što je potrebno za rješavanje ovog zadatka jest prisjetiti se (ili izvesti) formule za računanje površine trokuta u koordinatnom sustavu. Dakako, radi se o:

$$P = \frac{1}{2} |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$$

Možemo li ovaj pristup generalizirati i za $n \geq 4$? Sastoji li se optimalno rješenje samo od pokrivajućih trokuta? Odgovor je da!

Naime, zamislimo neko optimalno rješenje u kojem neki od pokrivajućih poligona nije trokut, već konveksni poligon s proizvoljnim brojem stranica. Poznato je da je svaki takav poligon moguće [triangulirati](#), odnosno njegovu površinu podijeliti na skup trokuta koji se međusobno ne sijeku. Stoga, dovoljno je prilikom pokrivanja ulaznih točaka u obzir uzeti samo trokute koji su njima određeni.

Ukupan broj trokuta određen ulaznim točkama je $\frac{1}{6}n(n-1)(n-2)$, a direktna primjena gornje zamjedbe dovodi nas do tipične primjene dinamičkog programiranja. Idemo redom po trokutima, te svaki trokut probamo staviti ili ne staviti u naš konačan skup pokrivajućih trokuta. Usput ćemo pomoću [bitmaske](#) pamtit i koje smo točke do sad pokrili. Formalnije,

$$dp(i, m) = \max(dp(i + 1, m), p_i + dp(i + 1, m'))$$

gdje i označava indeks trenutnog trokuta kojeg promatramo, p_i označava njegovu površinu, a m' označava osvježenu bitmasku m nakon što smo u skup dodali i -ti trokut. Ovaj pristup možemo implementirati u vremenskoj složenosti $\mathcal{O}(n^3 2^n)$ što je dovoljno za osvajanje 30 bodova na ovom zadatku. Primjer implementacije možete pronaći u datoteci `hulje_2_na_n_puta_n_na_3.cpp`.

Konačno, iskristit ćemo trivijalnu zamjedbu da se nikad ne isplati u rješenje dodati neki trokut koji je određen točkama koje smo već pokrili. Očito ćemo micanjem takvog trokuta iz rješenja smanjiti ukupnu površinu pokrivajućih trokuta, a sve će točke i dalje biti pokrivene. Sada ćemo u stanju dinamike pamtit samo masku, a u prijelazu ćemo u obzir uzeti samo one trokute čiji jedan vrh leži u prvoj nepokrivenoj točki bitmaske. Budući da takvih trokuta ima $\mathcal{O}(n^2)$, cijeli je algoritam moguće implementirati u složenosti $\mathcal{O}(n^2 2^n)$ što je dovoljno efikasno za osvajanje svih bodova na ovom zadatku.

Zadatak: 365

autor: Marin Kišić

U zadatku se zapravo traži broj mogućih pravokutnika koji sadrže samo slova s ili a , puta broj mogućih pravokutnika koji sadrže samo slova t , i ili n , puta broj mogućih pravokutnika koji sadrže samo slova k , e , d , z ili o . Svaki od navedenih brojeva možemo tražiti neovisno, budući da među slovima nema preklapanja. Ako zamislimo da su odgovarajuća slova imena koje tražimo jedinice, a ostala slova nule, zadatak se svodi na brojenje pravokutnika u matrici koji sadrže samo jedinice.

Fiksirajmo donji redak pravokutnika i te promatrajmo dijelove stupaca uzastopnih jedinica koji završavaju u odabranom retku i , tvoreći histogram. Da bismo izbrojili tražene pravokutnike u tom histogramu, prolazimo po njegovim stupcima s lijeva na desno i pitamo se koliko pravokutnika završava u trenutačnom stupcu j (neka je on visine H jedinica), točnije u polju (i, j) . Da bismo to saznali, promotrimo prvi niži stupac histograma lijevo od j , nazovimo ga k i neka je on visine h jedinica. Možemo uzeti bilo koji pravokutnik s visinom do H i širinom do $j - k$, njih ima $H \cdot (j - k)$. Osim tih pravokutnika koji završavaju u polju (i, j) , do istog polja možemo produžiti i bilo koji pravokutnik koji završava u polju (i, k) . Ako smo broj takvih pravokutnika prethodno spremili kao $f(i, k)$, onda je ukupan broj pravokutnika koji završavaju u polju (i, j) jednak

$$f(i, j) = f(i, k) + H \cdot (j - k).$$

Za brz pronalazak prvog nižeg stupca k lijevo od j možemo koristiti npr. monotoni stog. Ako takav stupac ne postoji, onda je $f(i, j) = H \cdot (j - 0)$.