Studentsko natjecanje 2015
25. listopada 2015.
opisi algoritama

Zadatak BITOVNI

Promatrajmo podnizove uzastopnih brojeva. Fiksirajmo lijevu granicu podniza, a desnu pomičimo udesno i promatrajmo kako se se mijenja bitovni OR brojeva u podnizu. U tom slučaju (za neku fiksnu lijevu granicu) imat ćemo samo O(log M) različitih vrijednosti za bitovni OR, gdje M predstavlja najveći broj u nizu. Zašto? Pa kako se pomičemo udesno OR se može samo povećavati i to samo tako da se neka nula u njegovoj binarnoj reprezentaciiji pretvori u jedinicu. To nam daje samo O(N log M) zanimljivih podnizova koje lako možemo pronaći ako za svaku poziciju u nizu i svaki bit izračunamo poziciju prvog sljedećeg elementa u nizu koji ima taj bit postavljen na 1. Sada najveći bitovni OR nekog podniza duljine K izračunamo tako da nađemo zanimljivi podniz duljine do K (jer svaki podniz možemo proizvoljno proširiti) koji ima najveći OR.

Zadatak KRATKI

Sortirajmo niz a i zamislimo njegove brojeve kao plave točke na brojevnom pravcu. Zamislimo i niz b na tom brojevnom pravcu, njegove brojeve kao crvene točke, tada je izraz S jednak zbroju udaljenosti svake plave točke do najbliže crvene točke. Primjetimo sad da će svaka crvena točka "pokrivati" neki interval uzastopnih plavih točaka (ona će im biti najbliža crvena točka).
Dakle niz a moramo podijeliti na K grupa i svakoj grupi pridjeliti jednu točku iz niza b. Točnije pridjelit ćemo im broj koji minimizira sumu udaljenosti točaka iz grupe do te točke. Poznato je da je optimalna točka u ovom slučaju median točaka iz grupe (ako ima dva medijana onda bilo koji). Neka je calc(x, y) funkcija (konstantne složenosti) koja nam vraća optimalnu sumu udaljenosti grupe $a_x,..,a_{y-1}$.

Zadatak sada rješavamo dinamičkim programiranjem (niz a indeksiramo od 0 do N-1):
f[j][i] ⇔ najmanja suma udaljenosti ako točke $a_i,..,a_{N-1}$ dijelimo u j grupa.
f[1][i] = calc(i, N)
f[j][N] = 0
f[j][i] = min (po ni od i do N) (calc(i, ni) + f[j-1][ni]).

U prijelazu dinamike fiksiramo prvu grupu, znamo da je njen početak i, pa biramo kraj s for petljom po ni. Iz tog razloga složenost ovog rješenja je $O(N^3)$.

Ključna opservacija je primjetiti sljedeće:
g[j][i] ⇔ onaj ni za koji f[j][i] postiže minimum,
vrijedi: g[j][i-1] <= g[j][i] <= g[j-1][i].

U prijelazu dinamike je dovoljno provjeravati samo ni iz intervala [g[j][i-1], g[j-1][i]].
Da bismo izračunali složenost ovakvog rješenja moramo pozbrojiti g[j-1][i]-g[j][i-1] po svim parovima (i, j). Taj zbroj je otprilike jednak g[n][0] + g[n][1] + .. + g[n][n] + g[n-1][n] + … + g[0][n] - (g[n][0] + … + g[0][0] + g[0][1] + … + g[0][n]) što je naravno $O(N^2)$.

Dakle složenost rješenja je $O(N^2)$.

Zadatak LUHN

Budući da je u tekstu zadatka opisan algoritam provjere ispravnosti broja kartice, a algoritam ima složenost duljine broja, dovoljno je bilo samo ga implementirati. Ključna stvar na koju je trebalo je paziti s koje se strane krene iterirati kada se udvostručuju brojevi.

Zadatak LUTRIJA

Brojeve promatrajmo kao vrhove u grafu. A dopuštene parove kao bridove između odgovarajućih vrhova. Tada su nizovi iz teksta zadatka zapravo jednostavni putevi u grafu. Uvjet o nepostojanju trijumvirata je zapravo ekvivalentan tvrdnji da je graf kaktus (više o kaktus grafovima: https://en.wikipedia.org/wiki/Cactus_graph).
Za početak je potrebno parsirati ovaj graf i prikazati ga kao uniju ciklusa, radi jednostavnosti bridove koji nisu niti u jednom u ciklusu možemo udvostručiti i prikazati ih kao cikluse duljine 2 (samo ćemo na njih morati pripaziti kasnije).

Sada je šetnja po grafu zapravo šetnja po ciklusima s tim da uvijek prelazimo na novi ciklus (neki na kojemu nismo do sada bili).
Neka je solve(x, dad) funkcija koja izračunava sve puteve koji počinju u x i nikada ne idu u ciklus s oznakom dad. Neka dad = -1 kao argument funkcije označava da nema ciklusa u kojeg ne smijemo ići.
Funkcija vraća niz brojeva takav da na i-tom mjestu stoji broj puteva duljine i. Pretpostavimo da je posljednji element vraćenog niza uvijek različit od nule.

Kako napisati funkciju solve? Koristimo dinamičko programiranje, u prijelazu biramo novi ciklus u koji ulazimo. To su svi ciklusi (osim dad) na kojima se x nalazi. Također odabiremo i vrh na tom ciklusu u koji idemo. Uvedimo sljedeće oznake:
ndad - novi ciklus
nx - novi vrh na ciklusu ndad
d - broj bridova između x i nx na ciklusu ndad (nije bitno u kojem smjeru).
len(ndad) - duljina ciklusa ndad

Nakon prvog koraka putevi se mogu nastaviti na bilo koji put kojeg izračunava solve(nx, ndad). Svom rješenju pribrojimo sve puteve koje vrati solve(nx, ndad) ali dva puta. Jednom proširene za d, a drugi put za len(ndad) - d jer postoje dva puta od x do nx (osim u slučaju kad smo udvostručili brid).

Rješenja funkcije možemo memoizirati, parova (x, dad) ima samo O(N).
Onda i u prijelazu pri odabiru novog para (nx, ndad) također imamo O(N) opcija, a za svaki od njih prolazimo po cijelom nizu koji vrati poziv solve(nx, ndad).
Zbroj veličina svih nizova kroz koje ćemo proći u jednom pozivu funkcije solve se amortizira u O(N) jer svi potpozivi rješavaju disjunktne dijelove grafa, a veličina vraćenog niza je sigurno manja ili jednaka broju vrhova u dijelu grafa koji je dostižan.
Činjenica koja pri zaključivanju navedenih složenosti može pomoći jest da je kaktus planaran graf i kao takav ima O(N) bridova.

Za konačno rješenje potrebno je samo pozbrojiti sve nizove koje vraćaju pozivi solve(i, -1) za svaki i. Ukupna vremenska i memorijska složenost je $O(N^2)$. Za detalje pogledati priloženi kod.

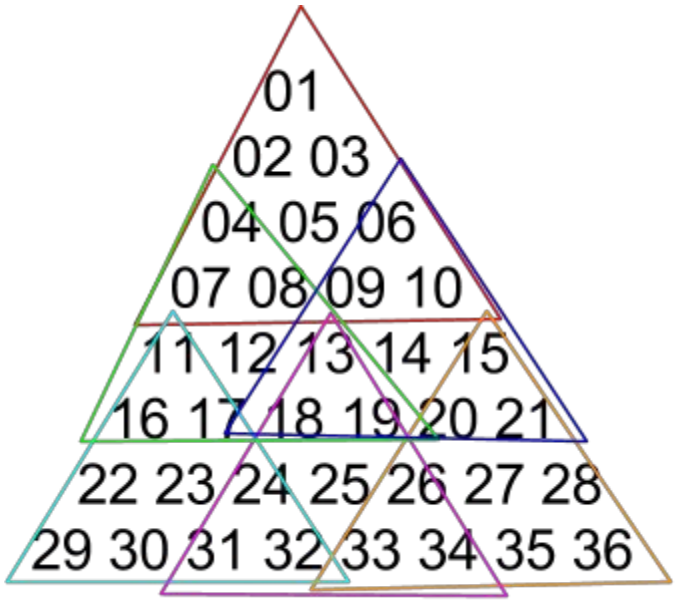
Zadatak PARADA

Prva stvar koju je trebalo primijetiti u zadatku jest da je funkcija cijene popravka monotona ali nije linearna. Kako funkcija cijene nije ujedno i linearna, nije nužno moguće pronaći jedinstevni put koji će uvijek biti najjeftiniji za popravak.
Međutim, kako je funkcija monotona, znamo da ako s danim budžetom možemo platiti popravak prolaska T tenkova, onda sigurno možemo platiti i popravak prolaska T-1 tenkova po istoj ruti. To nas navodi da rješenje možemo pronaći binarnim traženjem.
Ostaje još pronaći način koliko košta popravak prolaska T tenkova. Odgovor na to pitanje može se dobiti provođenjem Dijkstrinog algoritma na grafu na kojem su cijene bridova jednake cijeni popravka prometnica po kojima je prošlo točno T tenkova.
Ako Dijkstru implementiramo pomoću priritetnog reda (ili seta), ukupna složenost rješenja jednaka je O(M * log N * log R), gdje je N broj vrhova, M broj bridova a R gornja granica na rješenje.

Zadatak PIRAMIDA

Glavna ideja rješenja je sljedeća. Pretpostavimo da za zadanu piramidu na svakom polju imamo unaprijed izračunatu 3d tablicu $M(i, j, k)$ koja pohranjuje

maksimalan broj pod-piramide koja ima vrh u retku i i stupcu j, a duljina stranice joj je 2^k. U tom slučaju bismo na upit za maksimalni broj u pod-piramidi s vrhom u retku r, stupcu s i proizvoljne duljine stranice x mogli odgovoriti promatrajući najviše 6 elemenata. Slika prikazuje gdje bi otprilike trebalo postaviti manje pod-piramide:



Tablicu M možemo izgraditi tako da definiramo M(i, j, 1)=piramida(i, j). M(i, j, k) za k veći od jedan možemo izgraditi koristeći šest pod-piramida sa stranicama duljine 2^{k-1}. Njihovi maksimumi su pohranjeni u M(?, ?, k-1). Za točne koordinate tih pod-piramida moguće je konzultirati kodove službenih rješenja. Čuvanje cijele tablice u memoriji nije moguće zbog zadanih ograničenja. Međutim, ukoliko su upiti unaprijed učitani, moguće ih je odgovarati u uzlaznom redoslijedu po duljini stranice. Time smo postigli da je u memoriji samo potrebno čuvati dio tablice za dvije uzastopne vrijednosti od k.
Ukupna vremenska složenost opisanog rješenja je $O(N^2 log\, N)$, a memorijska $O(N^2)$.

Zadatak R9K

Za rješenje ovog je zadatka bilo je ključno pažljivo implementirati sve upute iz teksta zadatka. Kada se poruka podijeli na dijelove (tokenizira), bilo je potrebno izbaciti sve dijelove koji su ujedno i nadimci. Za prepoznavanje je li riječi nadimak, dovoljno je bilo koristiti bilo koji od poznatih algoritama za uspoređivanje stringova koji je dovoljno brz. Najčešći je odabir bio hashing, tj. Rabin-Karp algoritam, ali je bilo dovoljno koristiti i dobru strukturu podataka koja ograniči najveći broj stringova koje treba usporediti naivnim algoritmom. Primjer takve strukture je struktura set u programskom jeziku C++. Složenost

prvog algoritma je O(M+N+K), gdje je K ukupna količina ulaznih podataka, dok je složenost drugog O(K log N + K log M).

Za slučaj odabira izgradnje hash tablice uz "standardnu" valjajuću (engl. *rolling*) hash funkciju $f(S_{0,i}) = (f(S_{0,i-1}) * B + S_i) \% MOD$, gdje je S string, a B i MOD baze odnosno modulo, treba obratiti pozornost da se kao modulo ne koristi 2^k. Za taj slučaj postoji specifičan Thue-Morse niz koji ima svojstvo da stvara kolizije za gotovo sve odabrane baze. Više o tome možete pročitati u ovom radu Pachockija i Radoszewskog: http://www.mii.lt/olympiads_in_informatics/pdf/INFOL119.pdf

Zadatak SJECKANJE

Zadatak rješavamo u dva koraka: najprije za svaku točku odredimo njenu regiju te, zatim, odredimo najbrojniju regiju. Regija u kojoj se nalazi neka točka je jednoznačko određena s dva podatka: najgornjim rezom tipa A ispod točke i najlijevijim rezom tipa B desno od točke. Oba dva reza se mogu naći binarnim pretraživanjem nakon što sortiramo rezove tipa A i B odozgo prema dolje odnosno slijeva na desno. Određivanje da li je točka ispod ili iznad reza tipa A (odnosno lijevo ili desno reza tipa B) svodi se na jednostavnu analitičku geometriju pravca i točke. Drugi korak radimo tako da sortiramo sve pronađene regije i odredimo najdulji uzastopni niz istih.

Zadatak SLIKA

Glavna primjedba tijekom rješavanja zadatka je da su pikseli neovisni jedan o drugome. To znači da ako za neki intenzitet sive boje intenziteta I definiramo f(I) kao broj boja koje će konverzijom rezultirati tim intenzitetom, ukupno rješenje će biti jednako:

$$\prod_{i=1}^{R}\prod_{j=1}^{S} f(slika(i,j))$$

f(I) je moguće unaprijed izračunati i pohraniti u polju, prolazeći kroz 256^3 mogućih kombinacija svih boja.

Zadatak TABLICA

Zadatak rješavamo rekurzivnim algoritmom koji najprije odredi da li se u pojedinoj ćeliji nalazi formula ili broj. Ako se u ćeliji nalazi broj, jednostavno vraćamo njegovu vrijednost. Ako se nalazi formula, razlažemo je na pribrojnike,

pretvaramo adrese u broj retka i stupca, rekurzivno tražimo vrijednost pribrojnika koje na kraju zbrajamo kako bi dobili vrijednost tražene ćelije. Obzirom da nema cirkularnih zavisnosti garantiramo je da rekurzija uvijek završi. Kako bi rješenje bilo dovoljno efikasno potrebno je koristiti memoizaciju - ako smo već pozvali rekurzivnu funkciju za neku ćeliju onda vrijednost ne računamo ponovo nego samo vraćamo već izračunatu i spremljenu vrijednost.