Opisi algoritama

Zadatke, test primjere i rješenja pripremili: Ivan Katanić, Tonko Sabolčec, Marin Tomić, Mislav Balunović i Ante Đerek. Primjeri implementiranih rješenja su dani u priloženim izvornim kodovima koji ne odgovaraju uvijek u svim detaljima ovdje opisanim algoritmima.

Zadatak: Nalog

Predložio: Marin Tomić
Potrebno znanje: pohlepni algoritam

Zadatak rješavamo pohlepnim algoritmom: putne naloge razmatramo jedan za drugim i svakom pridružimo najkasniji mogući datum (31. prosinac) sve dok ne "potrošimo" ukupni iznos dnevnica. Kako bi osigurali da svaki putni nalog traje barem jedan dan, prije ovog postupka moramo smanjiti ukupan iznos dnevnica za $n \cdot 100$ kuna i za svaki putni nalog postaviti da je datum povratka jednak datumu polaska.

Također je potrebno napraviti funkcije koje datum pretvaraju u broj proteklih dana u godini i obratno. Ove funkcije se jednostavno ostvare ako se broj dana za svaki mjesec spremi u konstantno polje.

Zadatak: Mlin

Predložio: Ante Đerek
Potrebno znanje: ad-hoc

U ovom zadatku nije potrebno osmišljavati algoritam već sam direktno implementirati logiku opisanu tekstu zadatka. Rješenje možemo podijeliti na dva dijela: najprije odredimo na kojima od 16 mogućih lokacija postavljen mlin. U drugom djelu generiramo i ispisujemo tablicu znakova koja opisuje poziciju.

Jedan jednostavni način implementacije prvog dijela je da se za svaki od 16 mogućih mlinova u konstantnom polju ručno zapišu koordinate (redak i stupac) prvog i zadnjeg polja u mlinu. Srednje polje se lagano izračuna na temelju ova dva polja, te također jednostavno iteriramo kroz sve ostale pozicije u tablici znakova između prvog i zadnjeg polja kada pronađemo mlin. Sada s `for` petljom prolazimo kroz svih 16 mogućnosti, provjeravamo postoji li mlin te na svaku poziciju u tablici zapisujemo odgovarajuću vrijednost ako smo pronašli mlin.

Zadatak: Rebus

Predložio: Tonko Sabolčec
Potrebno znanje: Teorija grafova

Konstruirajmo usmjereni graf čiji čvorovi su pozicije znakova iz ulaznog stringa, a čiji bridovi su dobiveni na sljedeći način: Promotrimo dvije uzastopne riječi između kojih se nalazi k apostrofa. Iz svakog čvora koji odgovara nekoj poziciji i koja se nalazi u prvoj riječi dodajmo brid u čvor j koji se nalazi u sljedećoj riječi (ako takav postoji) i za koji vrijedi $j = i + 2k$. Također, povežimo čvorove i i $i + 1$ ako se obje te pozicije nalaze unutar iste riječi.

Primijetimo da je svako rješenje rebusa prikazano kao neki put od čvora 1 do čvora n u navedenom grafu. Dakle, želimo pronaći leksikografski najmanji put u tom grafu. Odmah možemo iz grafa obrisati čvorove iz kojih ne postoji niti jedan put do čvora sa oznakom n.

Prvo ubacimo čvor 1 u strukturu red. U svakoj iteraciji radimo sljedeće: Promotrimo sve čvorove do kojih postoji brid iz barem jednog elementa koji se trenutno nalazi u redu. Od svih takvih čvorova promotrimo samo one u kojima se nalazi leksikografski najmanji znak. U pomoćni red ubacimo sve takve čvorove. Sada glavni red zamijenimo pomoćnim redom i radimo sljedeću iteraciju.

Jasno je kako navedeni algoritam u i-toj iteraciji ima u redu skup čvorova koji su kandidati za i-ti znak rješenja. Dakle, nakon n-te iteracije znamo svaki znak rješenja.

Ono što nije odmah jasno je kolika je složenost navedenog algoritma. Ključno je uočiti kako će se svaki čvor u redu nalaziti točno jednom. Naime, ako se taj čvor pojavljuje u rješenju onda je jedinstveno određena pozicija na kojoj će se on pojaviti. Pozicija na kojoj će se čvor i pojaviti je upravo $i - 2s$ gdje je s ukupan broj apostrofa prije pozicije i. Kako je pozicija jedinstveno određena, vidimo da se čvor u strukturi red nalazi maksimalno jednom pa je složenost navedenog algoritma $O(N)$.

Zadatak: Dnevnice

Predložio: Marin Tomić
Potrebno znanje: Pohlepni algoritam

Rješenje je identično onome za zadatak "Nalog" osim što moramo voditi računa da najkasniji datum povratka je najviše 99 dana nakon datuma polaska.

Zadatak: Potencijal

Predložio: Ante Đerek
Potrebno znanje: rekurzivno ispitivanje svih kombinacija, meet in the middle

Najprije je potrebno implementirati funkciju koja za zadanu poziciju prebrojava mlinove. Ova funkcija se, kao kod zadatka "Mlin" može jednostavno implementirati ako u konstantnom polju zapišemo parametre svih 16 mogućih mlinova (primjerice prvo i zadnje polje u mlinu).

Ako se traži puni potencijal, postoji najviše 2^{24} pozicija koje je potrebno provjeriti te je dovoljno brzo rješenje koje rekurzivno generira sve takve pozicije, te za svaku od njih provjeri broj mlinova.

Ovaj pristup se ne može direktno primijeniti na slučaj kada se traži djelomični potencijal jer je 3^{24} kombinacija puno previše kako bi rješenje radilo u zadanim vremenskim ograničenjima. Postoji puno različitih ideja koje vode do dovoljno brzog rješenja za računanje djelomičnog potencijala, ovdje opisujemo takozvani *meet in the middle* algoritam. Označimo s D glavnu dijagonalu ploče za mlin (polja `a1`, `b2`, `c3`, `e5`, `f6` i `g7`), a neka su A sva polja na ili iznad dijagonale D, a B sva polja na ili ispod dijagonale D. A i B se oba dakle sastoje od 15 polja. Ključno je primjetiti da svaki mlin potpuno leži ili u A ili u B pa je u svakoj poziciji P' ukupan broj mlinova jednak $P'_A + P'_B$ gdje je prvi pribrojnik broj mlinova u A, a drugi broj mlinova u B. Rješenje sada, za zadanu poziciju P, radi u dva koraka:

1. Rekurzijom, na sve moguće načine odaberemo vrijednosti za polja iz A, za svako polje razmatramo slučaj kada polje ostane prazno, kada stavimo bijelu figuru te kada stavimo crnu figuru (naravno uvijek pazimo da smo konzistentni s zadanom početnom pozicijom P). Za dobivenu polu-poziciju Q:
   (a) Izračunamo k_Q — broj mlinova u A.
   (b) Izračunamo d_Q — broj između 0 i $3^6 - 1$ koji u potpunosti opisuje stanje polja na dijagonali D.
   (c) Uvećamo $S(d_Q, k_Q)$ za jedan. S je dvodimenzionalno polje veličine $3^6 - 1 \times 9$ u kojem za svako stanje dijagonale d i svaki mogući broj mlinova k čuva broj polu-pozicija Q koje imaju to k mlinova i stanje dijagonale d.

2. Rekurzijom, na sve moguće načine odaberemo vrijednosti za polja iz B, za svako polje razmatramo slučaj kada polje ostane prazno, kada stavimo bijelu figuru te kada stavimo crnu figuru (naravno uvijek pazimo da smo konzistentni s zadanom početnom pozicijom P). Za dobivenu polu-poziciju R:
   (a) Izračunamo k_R — broj mlinova u B.
   (b) Izračunamo d_R — broj između 0 i $3^6 - 1$ koji u potpunosti opisuje stanje polja na dijagonali D.
   (c) Za svaki k između 0 i 8, ukupnom rezultatu pribrojimo $S(d_R, k)$ pozicija s $k + k_R$ mlinova.

Svaki od koraka razmatra 3^{15} polu-pozicija pa je ovaj algoritam moguće implementirati tako da radi u zadanim vremenskim ograničenjima.

Zadatak: Dretve

Predložio: Tonko Sabolčec
Potrebno znanje: ad-hoc, simuliranje, stablo intervala

Prvi podzadatak može se riješiti jednostavnim simuliranjem raspoređivanja dretvi opisanim u zadatku.

Rješenje se može ubrzati osmišljavanjem strukture podataka koja podržava sljedeće funkcije:

1. $nextIn()$ — koja dretva sljedeća ulazi u listu (i kada)
2. $nextOut()$ — koja dretva sljedeća izlazi iz liste (i kada)
3. $work(t)$ — simulira rad sustava za t ciklusa

Sve 3 funkcije rade pod pretpostavkom da se lista dretvi u sustavu ne mijenja. Upiti $nextIn$ i $nextOut$ vraćaju par (D, t) gdje je D oznaka dretve, a t broj potrebnih ciklusa koje sustav mora izvršiti do trenutka pojave sljedeće dretve D, odnosno brisanja sljedeće dretve D iz liste. Pomoću ove strukture podataka simulaciju rada operacijskog sustava možemo izvesti tako da ponavljamo sljedeće korake dok sve dretve nisu završile s radom:

1. $(D_{in}, t_{in}) := nextIn()$
2. $(D_{out}, t_{out}) := nextOut()$
3. Ako je $t_{in} \leq t_{out}$:
   (a) $work(t_{in})$
   (b) ubaci dretvu D_{out} na kraj liste
4. Inače:
   (a) $work(t_{out})$
   (b) izbaci dretvu D_{out} iz liste

Napravimo jednostavnu verziju takve strukture podataka! Pamtit ćemo globalne varijable C (trenutni ciklus), P (pokazivač na dretvu koja se sljedeća izvodi), L (lista dretvi koje su u sustavu).

Na upite $nextIn$ jednostavno vratimo sljedeću dretvu na ulazu i broj ciklusa (d_i - C) gdje je d_i oznaka ciklusa u kojem se dretva i raspoređuje u sustav. Složenost poziva ovog upita je $O(1)$.

Odgovor na upite $nextOut$ odredit ćemo tako da za svaku dretvu koja se nalazi u sustavu izračunamo koliko ciklusa još moramo izvrtiti tako da ona završi s radom i vratiti dretvu za koju je taj broj ciklusa najmanji. Za neku dretvu L_i u listi L taj broj ciklusa iznosi:

- $(a_i - 1) \cdot duljina(L) + i - P + 1$, ako je $i \geq P$
- $a_i \cdot duljina(L) + i - P + 1$, ako je $i < P$

gdje smo s a_i označili broj ciklusa koje je potrebno izvršiti na dretvi L_i. Složenost poziva ovog upita je $O(n)$.

Za proceduru $work(t)$ bitno je primijetiti da će se na svakoj dretvi izvršiti barem $\lfloor \frac{t}{duljina(L)} \rfloor$ instrukcija. Stoga za svaku dretvu L_i možemo odmah smanjiti broj preostalih instrukcija za tu vrijednost. Preostalih (t mod $duljina(L)$) ciklusa možemo odsimulirati rješavajući dretvu po dretvu, pri čemu mijenjamo pokazivač P. Složenost poziva ove procedure iznosi $O(n)$. Budući da ćemo n puta pozvati funkcije $nextIn$, $nextOut$ i $work(t)$, ukupna složenost iznosi $O(n^2)$.

Funkcije $nextOut$ i $work(t)$ možemo ubrzati pomoću stabla intervala (*tournament* stabla). U svakom čvoru stabla pamtit ćemo trojku (i, c, cnt) u kojoj je i oznaka dretve koja će najprije biti gotova s radom, c broj instrukcija potrebnih za izvršavanje na toj dretvi i cnt broj aktivnih dretvi (one koje su trenutno u sustavu) na nekom intervalu.

Odgovori na upite *nextOut* svode se na traženje trojke s najmanjom vrijednošću c u stablu intervala. Kada nađemo takvu trojku, znamo i indeks i te dretve pa mozemo odrediti i vrijeme kada ce ona izaći iz sustava koristeći ranije spomenuti način. Složenost funkcije *nextOut* iznosi $O(\log n)$.

Proceduru $work(t)$ možemo ubrzati ako stablo intervala podržava operaciju za smanjivanje svih vrijednosti u nekom intervalu za određeni iznos. Jedan način na koji možemo riješiti taj problem je korištenje tehnike lijenog propagiranja (*lazy propagation*). Drugi način je da prilikom dodavanja nove dretve u stablo intervala umjesto broja instrukcija te dretve stavimo oznaku *punog kruga* u kojem će dretva završiti s radom. *Puni krug* je broj punih krugova koje je napravio pokazivač tijekom rada (broj punih krugova se poveća kada se pokazivač nakon rada na zadnjoj dretvi u listi postavi na prvu). Tada dodatno moramo pamtiti globalnu varijablu koja predstavlja broj punih krugova od početka rada sustava. Korištenjem drugog načina, možemo ostvariti proceduru $work(t)$ u složenosti $O(1)$.

Ukupna složenost iznosi $O(n \log n)$.

Zadatak: Blokada

Predložili: Ante Đerek, Ivan Katanić
Potrebno znanje: DFS ili dinamičko programiranje, pohlepni algoritmi

Označimo polje $(1, 1)$ sa s i polje (n, m) sa t. Promotrimo najgornje travnato polje u stupcu. Označimo to polje s x, a polje desno od njega sa y. Razmatramo sljedeće slučaje:

1. Postoji travnati put od s do x i travnati put od y do t.

   Ovo polje nužno moramo betonirati jer betoniranjem polja trenutnog stupca koja se nalaze ispod njega ne možemo blokirati travnati put koji ide od s preko x i y do t.

2. Ne postoji travnati put od s do x ili ne postoji travnati put od y do t.

   Pokazat ćemo da ni u kojem slučaju ne moramo betonirati ovo polje. Zaista, ako ne postoji travnati put od s do x onda ne postoji travnati put od s do t koji ide preko x pa njegovo betoniranje nema efekta. Ako postoji put od s do x, a ne postoji put od y do t, onda svaki travnati put od s do t koji prolazi kroz x nužno prolazi i kroz polje ispod njega. Slijedi da umjesto polja x možemo betonirati polje ispod njega (ako je to potrebno).

Sada se nameće slijedeći pohlepni algoritam: idemo redom po travnatim poljima od najgornjeg prema najdoljnjem i blokiramo polja kada god je to nužno (kada se nađemo u uvjetima prvog slučaja).

Za implementaciju ovog algoritma potrebno je samo znati izračunati postoji li put od s do x i od x do t za svako polje x. To se može učiniti jednostavno pomoću dinamičkog programiranja ili DFS-a. Složenost rješenja je $O(nm)$.

Zadatak: Spojka

Predložio: Ivan Katanić
Potrebno znanje: stringovi, trie

Za početak ćemo zadane riječi ubaciti u strukturu podataka *trie*. To nam omogućava brze odgovore na upite oblika: "Koliko postoji riječi w_i kojima je neka riječ s prefiks?". Umjesto ove strukture podataka, zadatak se mogao i efikasno riješiti korištenjem tipa `dict` u Python-u ili `set` odnosno `unordered_set` u jeziku C++.

Fokusirajmo se na jednostavniji potproblem, u kojemu je prva riječ w_i fiksirana. Potrebno je, dakle, za neki i, prebrojati koliko postoji j takvih da je w_j spojka od w_i. Ako taj potproblem znamo riješiti, onda je rješenje cijelog zadatka jednostavno zbroj rješenja tog potproblema za svaki i.

Prva ideja bila bi za svaki sufiks riječi w_i izbrojati koliko postoji riječi w_j kojima je on prefiks. No, tada bismo neke riječi w_j brojali više puta, a to ne želimo. Potrebno je primjetiti da ako sufikse obrađujemo od

kraćih prema dužima, tada će za svaki sufiks vrijediti jedno od sljedećeg:

- a) sve riječi w_j kojima je on prefiks smo već prebrojali (kod nekog kraćeg sufiksa, koji je zapravo prefiks trenutnog sufiksa)
- b) niti jednu riječ w_j kojoj je on prefiks nismo još prebrojali.

Dakle sve što je u orginalnom algoritmu potrebno izmjeniti je dodati "preskakanje" određenih sufiksa (onih za koje vrijedi tvrdnja a)). Takve sufikse je jednostavno detektirati, to su točno oni sufiksi za koje postoji neki kraći sufiks koji im je prefiks.

Ukupna složenost rješenja je $O(\sum_{i=1}^{n} w_i^3)$.

Za one koji žele više: riješite zadatak u složenosti $O(\sum_{i=1}^{n} w_i^2)$.

Za one koji žele još više: riješite zadatak u složenosti $O(\sum_{i=1}^{n} w_i)$.

Zadatak: Dos

Predložio: Ante Đerek
Potrebno znanje: isprobavanje svih kombinacija, grafovi

Potrebno je zaključiti nekoliko bitnih svojstava niza karata koje možemo baciti prije nego što pristupimo rješavanju zadatka. Kao prvo, ako smo bacili jednu kartu neke boje X, onda možemo i baciti sve ostale karte te boje tako da ih ili ubacimo između dvije uzastopne karte boje X ili (ako je X boja zadnje bačene karte) dodamo na kraj niza bačenih karata. Zadatak se, dakle, svodi na to da vidimo koje je sve podskupove boja moguće odbaciti te odabrati onaj podskup koji sadrži ukupno najviše karata.

Razmatrajmo niz bacanja te pokušajmo eliminirati neke nizove koje nije potrebno razmatrati jer postoje drugi nizovi koji će nam dati isti podskup boja.

- Ako se u nizu S ista boja pojavljuje u 3 uzastopna bacanja onda ga nije potrebno razmatrati jer srednju od te tri karte možemo izbaciti i dobiti niz koji sadrži iste boje. Dakle, razmatramo samo nizove koji u svakom bacanju (osim možda u prvom) mijenjaju boju.
- Ako se u nizu S dva puta mijenja boja iz A u B onda ga nije potrebno razmatrati. Neka je niz S oblika $xA_iB_iyA_jB_jz$ gdje su x, y i y nizovi karata, obrtanjem srednjeg dijela niza dobivamo $S' = xA_iA_jy^RB_iB_j$ koji je također ispravan te sadrži iste boje kao S.
- Ako se u nizu S zadnja boja pojavljuje još negdje onda S nije potrebno razmatrati uz argumentaciju sličnu prethodnom slučaju. Dakle, razmatramo samo one S u kojima se zadnja boja pojavljuje samo na tom zadnjem mjestu.

Dakle, dovoljno razmatrati samo one nizove S koji u svakom bacanju (osim možda u prvom) mijenjaju boju, nikad ne mijenjaju boju iz nekog A u neki B više od jednom te u kojima se zadnja boja ne pojavljuje više od jednom. Dodatnim zaključivanjem možemo se i uvjeriti da je dovoljno razmatrati samo one S koji mijenjaju boju *najviše 5 puta*. Dokaz ove tvrdnje prepuštamo čitatelju.

Rješenje sada rekurzivno ispituje sve moguće opisane nizove S te traži onaj koji sadrži najpovoljniji podskup boja. Kada smo pronašli optimalni S sve ostale karte jednostavno ubacujemo u njega.

Za one koji žele više: Dokažite nedokazane tvrdnje iz teksta zadatka.

Zadatak: Lopoči

Predložio: Ivan Katanić
Potrebno znanje: Dinamičko programiranje

Kada bi svi lopoči bili različitih visina onda bi postojao samo jedan način da ih sve posjetimo pa zapravo jedina odluka koju moramo donijeti je kojim redoslijedom posjetiti lopoče istih visina. Lagano se uvjeriti

da možemo samo razmatrati strategije u kojima uvijek najprije skočimo na krajnje lijevi ili desni lopoč neke visine h pa onda redom posjetimo sve lopoče do krajnje desnog ili lijevog lopoča iste visine.

Stoga, zadatak možemo riješiti dinamičkim programiranjem. Za svaku visinu h, od manjih visina prema većima računamo dva broja $L(h)$ i $R(h)$ — najmanji broj kalorija koje je potrebno potrošiti kako bi obišli sve lopoče čije su visine najviše h i završili na krajnje lijevom odnosno krajnje desnom lopoču visine h. Rekurzivnu formulu te rješenje tehnikom dinamičkog programiranja jednostavno napravimo tako da razmatramo obje mogućnosti za prethodnu visinu.

Zadatak: Daske

Predložio: Ivan Katanić
Potrebno znanje: algoritmi skeniranja linijom, stablo intervala

Zamislimo graf u kojemu su segmenti (daske) iz zadatka vrhovi. A neusmjereni brid između dva vrha postoji ako se odgovarajući segmenti sijeku. Identificirajmo povezane komponente u tom grafu. Jasno je da se šetnjom može doći od bilo koje točke jednog segmenta do bilo koje točke drugog segmenta ako i samo ako su oba segmenta u istoj komponenti. Upite, dakle, rješavamo tako da pronađemo odgovarajuće segmente za obje točke i provjerimo jesu li u istoj komponenti. Odgovarajući segment (npr. horizontalni) za neku točku možemo brzo pronaći binarnim pretraživanjem po svim horizontalnim segmentima s istom y koordinatom, sortirane prema desnoj x koordinati.

Kako identificirati komponente? Glavni izazov u ovom problemu je taj što broj bridova može biti kvadratan u odnosu na broj vrhova tako da si prolazak kroz sve bridove naivnim algoritmom ne možemo priuštiti. Na našu sreću, za identifikaciju komponenti nikada ne treba više od $n - 1$ bridova.

Bitne bridove pronalazimo *sweep-line* algoritmom. Krećemo s grafom bez bridova, tj. svaki segment je u svojoj vlastitoj komponenti, a komponente označavamo različitim cijelim brojevima. Označimo s C_s oznaku komponente segmenta s. Skeniramo ravninu vertikalnom linijom s lijeva na desno.

Događaji prilikom skeniranja su:

- početak horizontalnog segmenta – proglašavamo ga aktivnim
- kraj horizontalnog segmenta – proglašavamo ga neaktivnim
- vertikalni segment – registriramo sve bitne bridove između njega i horizontalnih segmenata
  - Što čini bitan brid između v i nekog horizontalnog segmenta h?
    - v i h se sijeku (tj. h je aktivan)
    - v i h trenutno nisu u istoj komponenti (tj. $C_v \neq C_h$)

Brzo pronaći bitan brid za vertikalni segment v (koji se proteže od v_{y1} do v_{y2} na x koordinati v_x), možemo na više načina. Predlažemo sljedeći algoritam:

1. pitamo se koja je najmanja i najveća oznaka komponente koju ima neki aktivni horizontalni segment s y koordinatom u $[v_{y1}, v_{y2}]$
2. ako su obje oznake jednake C_v, tada smo gotovi (pronašli smo već sve bitne bridove za v) i prekidamo algoritam
3. inače, uzimamo jednu od te dvije oznake (onu koja je različita od C_v) i spajamo odgovarajuću komponentu sa C_v (time smo implicitno dodali brid između v i nekog horizontalnog segmenta iz druge komponente)
4. ponavljamo sve od koraka 1.

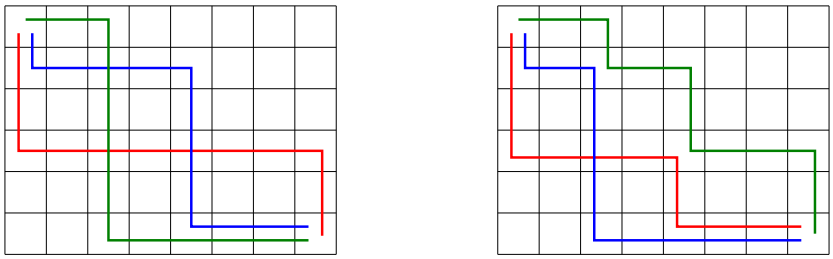
Spajanje komponenti možemo odraditi tako da prođemo kroz sve vrhove manje komponente i promijenimo im oznaku C.

Ako oznake komponenti aktivnih horizontalnih segmenata držimo u stablu intervala (*tournament tree*) koje odgovara y koordinatama, tada brzo možemo pronaći najmanju odnosno najveću oznaku aktivnog horizontalnog segmenta iz nekog y intervala. Spomenuti način spajanja komponenti je u općenitom slučaju složenosti $O(n \log n)$, a kod nas je složenosti $O(n \log^2 n)$ budući da se pri promjeni oznake C nekog horizontalnog segmenta može dogoditi da ju moramo osvježiti i u stablu intervala.

Zadatak: Tragovi

Predložio: Tonko Sabolčec
Potrebno znanje: grafovi, stabla, pohlepni algoritmi

Pretpostavimo da je K optimalni broj putova koji prekrivaju sve tragove te da smo našli neki takav skup od K putova. Tada možemo zamijeniti neke dijelove putova tako da jedan od putova bude "najgornji" mogući put (koji dobijemo tako da počevši od polja $(1, 1)$ idemo desno koliko možemo, spustimo se za 1 polje prema dolje, pa opet idemo desno koliko možemo itd.). Brisanjem najgornjeg puta dobit ćemo neki novi izgled dvorišta u kojem je optimalni broj koraka $K - 1$. Prema tome, dovoljno je u svakom trenutku naći najgornji put, obrisati ga i to ponavljati dokle god postoji neki put.



Slika 1: Slika lijevo prikazuje neki optimalni skup putova koji pokrivaju sve tragove. Promjenom nekih putova moguće je dobiti novi skup optimalnih putova među kojima se nalazi i najgornji put prikazan zelenom bojom na slici desno.

Posebni slučaj kada je $n = 3$ može se riješiti tako da se odrede najgornji i najdonji putovi. Kada se obrišu svi tragovi na tim putovima ostat će samo skupine tragova u drugom retku. Pribrojimo rješenju još i broj tih skupina i riješili smo taj podzadatak.

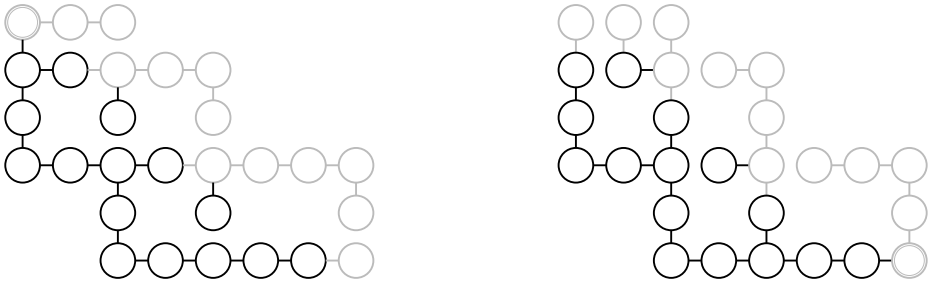
Neka je P skup svih tragova koji se u nekom trenutku nalaze na najgornjem putu, a S skup svih ostalih tragova. Trag na poziciji $(1, 1)$ zvat ćemo *početnim* tragom, a onaj na poziciji (n, m) *završnim* tragom. Općenito, kada nađemo neki najgornji put ne smijemo obrisati sve tragove na tom putu jer bi tada neki tragovi iz S ostali nepovezani s početnim i/ili završnim poljem. Za trag iz skupa P ćemo reći da je povoljan za brisanje ako su nakon njegovog brisanja svi tragovi iz skupa povezani i s početnim i završnim tragom.

Najjednostavniji način na kojim možemo provjeriti povezanost tragova iz S nakon brisanja nekog traga iz P jest pomoću *DFS* obilaska po tragovima iz početnog i završnog čvora. U najgorem slučaju postoji $O(nm)$ potrebnih putova. Za svaki trag na najgornjem putu P ćemo provjeriti povezanost tragova iz S nakon njegovog brisanja, pa je ukupna složenost ovakvog algoritma $O(n^2m^2(n + m))$ i to je rješenje koje prolazi na 2. podzadatku.

Tragove iz P koje smijemo obrisati možemo pronaći i prolaskom po svim tragovima iz skupa P i promatranjem njihovih susjednih polja. Pretpostavimo da brišemo po redu sve tragove iz P od početnog prema završnom. Recimo da se trenutno nalazimo na tragu na poziciji (x, y) gdje je x oznaka retka, a y oznaka stupca. Ako se na poziciji $(x + 1, y)$ nalazi trag iz skupa S, a na poziciji $(x + 1, y - 1)$ nema traga, to znači da će brisanjem trenutnog traga iz P, trag $(x + 1, y)$ ostati nepovezan s početnim tragom. Tada moramo vraćati tragove koje smo brisali tako dugo dok trag $(x + 1, y)$ ne postane povezan s početnim tragom. Time smo osigurali da svi tragovi iz S nakon brisanja nekih tragova iz P budu povezani s početnim tragom. Analogno tome potrebno je još osigurati povezanost svih tragova iz S sa završnim tragom, što

možemo postići na sličan način iteriranjem po svim tragovima iz P od završnog prema početnom. Ovaj algoritam ima složenost $O(nm(n+m))$ i rješava 3. podzadatak.

Povezanost nekog čvora iz skupa S sa početnim i završnim tragom nakon brisanja nekog traga iz P možemo provjeriti i na drukčiji način: ako "najdonji" put kojim možemo doći od nekog traga s iz S do početnog ili završnog traga prolazi nekim tragom p iz P, trag p ne smijemo obrisati jer bi inače trag s ostao nepovezan s početnim i/ili završnim tragom. Kako bismo brzo mogli odgovoriti postoji li za p neki s čiji najdonji put prolazi tragom p, koristit ćemo 2 *stabla najdonjih putova*: jedno kojim ćemo moći provjeriti povezanost s početnim tragom i drugo s kojim ćemo provjeravati povezanost sa završnim tragom. Stablo najdonjih putova kojim provjeravamo povezanosti s početnim tragom dobijemo tako da promatramo tragove kao čvorove stabla i za svaki čvor tražimo susjedan čvor na koji bi se pomaknuli ako želimo doći do početnog čvora najdonjim putem te povežemo te čvorove.



Slika 2: Lijevo je prikazano stablo najdonjih putova kojim ćemo provjeravati povezanosti s početnim čvorom, a desno stablo koje će poslužiti pri provjeravanju povezanosti sa završnim čvorom. Čvorovi iz P označeni su sivom bojom, dok su čvorovi iz skupa S označeni crnom bojom.

Svaki čvor može biti u jednom od tri stanja:

- *aktivan* — nalazi se u skupu S
- *neaktivan* — nalazi se u skupu P
- *obrisan* — trag koji pripada čvoru je obrisan

Promotrimo podstablo nekog čvora p iz P na lijevom stablu sa slike. Ako se u podstablu nalazi neki aktivan čvor, tada čvor p ne smijemo obrisati. Isto vrijedi i za stablo desno na slici. Sada možemo pronaći skup B svih čvorova koje smijemo obrisati. Nakon što obrišemo čvorove iz B, moramo ažurirati najgornji put i pronaći novi skup cvorova za brisanje B'. Ažuriranje najgornjeg puta možemo jednostavno obaviti u složenosti $O(len(B))$. Za određivanje čvorova iz skupa B' bitno je primijetiti da se oni nalaze u okolini čvorova koje smo dodali u prethodnom koraku prilikom ažuriranja najgornjeg puta P pa je složenost ovog dijela $O(len(B) + len(B'))$.

Potrebno je još odgovoriti na pitanje kako brzo postići upite o broju aktivnih čvorova u nekom podstablu i postavljanjem nekog čvora u neaktivno stanje. Ako pustimo *DFS* iz korijena stabla i za svaki čvor pamtimo *discovery* i *finish time* (vremena u kojima smo ušli i izašli iz čvora), tada u logaritamskoj strukturi možemo za svaki čvor pamtiti vrijednost 1 ako je čvor aktivan ili 0 ako nije. Oba upita možemo postići u složenosti $O(\log(nm))$.

Ukupna složenost algoritma je $O(nm\log(nm))$.