

Sisukord

1. Hoovi pühkimine	2
2. Vale kuju 1	3
3. Kummalised kapivargad	4
4. Kõrgeim torn	7
5. Vale kuju 2	8
6. Mugandatud ahel	10

1. Hoovi pühkimine (1uud)

1 sek

20 punkti

Idee: Ahto Truu, teostus ja lahenduse selgitus: Targo Tennisberg

Leida, kuidas katta $M \times N$ ruudustik optimaalselt 1×3 tükkidega.

Siin on kolm kvalitatiivselt erinevat varianti: ruudustik laiussega 1, ruudustik laiussega 2 ja laiema ruudustikud.

Kui laius on 1, saab pühkida algusest ruute kolmekaupana. Ruutude arv võib kas täpselt jaguda või siis võib 1 või 2 ruutu üle jääda. Vastus on seega $\lceil M \times N \div 3 \rceil$.

Kui laius on 2, saab pühkida mõlemat rida algusest kolmekaupana. Ruutude arv võib kas täpselt jaguda või siis võib 1×2 või 2×2 ruutu üle jääda. Esimene juhtum on siin ülesandes ainuke tõeline erand, selle allesjäänud 1×2 ribakese katmiseks on vaja kaks korda luuda liigutada, sest nii kitsas kohas saab pühkida ainult pikkupidi.

Kui laius on 3 või rohkem, saab pühkida kolme ruudu kaupa enamvähem igal viisil, kuni lõpuks jääb alles kas 1×1 , 1×2 või 2×2 ala. Kuna aga hoovi laius on vähemalt 3, on alati piisavalt ruumi see allesjäänud tükike vastavalt ühe või kahe luuavibutusega üle käia.

Seega on vastus alati $\lceil M \times N \div 3 \rceil$, välja arvatud juhul, kui kogu ruudustik on mõõtmetega $2 \times (3K + 1)$, sellisel juhul on vastus ühe võrra suurem.

2. Vale kuju 1 (kuju1)

1 sek / 3 sek

30 punkti

Idee: Phil Labuschagne, teostus ja lahenduse selgitus: Ahto Truu

Antud $A_1 < A_2 < \dots < A_N$ ja $B_1 \leq B_2 \leq \dots \leq B_M$, kus $M, N \leq 200\,000$. Leida iga i jaoks vähim j , mille korral $B_i \leq A_j$.

Esimene võimalus selle ülesande lahendamiseks on püstitusest lähtuv naiivne algoritm: võrdleme iga i korral B_i väärtust järjest A_1, A_2, \dots väärtustega, kuni leiame esimese A_j , mis pole väiksem kui B_i . Nii on tehtud failides `sol_naive.py` ja `sol_naive.cpp` toodud lahendustes, mis vaatavad iga i väärtuse jaoks läbi keskmiselt pooled (ja halvimal juhul isegi kõik) j väärtused ning kulutavad selleks kokku $O(M \cdot N)$ operatsiooni. See on piisav, et lahendada esimene alamülesanne, aga teises alamülesandes võivad M ja N mõlemad olla kuni 200 000 ja siis võib selle algoritmiga kuluda kuni 40 000 000 000 operatsiooni, mida on ajalimiiti mahtumiseks selgelt liiga palju.

Üks võimalus parema lahenduse saamiseks on kasutada ära, et jada A on sorteeritud ja leida iga i jaoks sobiv j väärtus kahendotsinguga. Kahendotsing kulutab N -elemendilises jadas õige väärtuse leidmiseks $O(\log(N))$ võrdlemist ja seega kulub kokku $O(M \cdot \log(N))$ operatsiooni. Failides `sol_bin_man_ok.py` ja `sol_bin_man_ok.cpp` on kahendotsing programmeeritud ise, failides `sol_bin_std_ok.py` ja `sol_bin_std_ok.cpp` toodud lahendused kasutavad standardteegi funktsioone. Mõlemad variandid on piisavalt efektiivsed, et teenida maksimumpunktid.

Teine võimalus on kasutada ära, et mõlemad jadad on sorteeritud. See tähendab, et kui me oleme mingi indeksi i jaoks juba leidnud indeksi j , mille korral $A_{j-1} < B_i \leq A_j$, siis teame, et $i + 1$ jaoks sobiv j väärtus ei saa olla väiksem. Kuigi failides `sol_merge_ok.py` ja `sol_merge_ok.cpp` toodud lahendustes on kaks kordust üksteise sees, tasub tähele panna, et sisemine `while`-kordus alustab iga kord sealt, kus eelmine `while`-kordus lõpetas. Kui `for`-kordus käib läbi M erinevat i väärtust, käivad kõik `while`-korduse käivitused kokku läbi N erinevat j väärtust ja seega kulutab selline kahe jada paralleelse läbimise algoritm $O(M + N)$ operatsiooni.

Kas efektiivsem on kahendotsinguga või jadade paralleelse läbimisega lahendus, sõltub sisendandmetest. Kui M ja N on samas suurusjärgus, on efektiivsem paralleelse läbimisega lahendus. Kui M on palju väiksem kui N , siis otseselt vastuse arvutamine on efektiivsem kahendotsinguga. Aga samas tuleb tähele panna, et andmete sisse lugemiseks tuleb igal juhul teha $O(M + N)$ operatsiooni, seega siin ülesandes ei saa programmi kogutööaeg isegi M ja N ebavõrdsuse korral sellest väiksem olla.

3. Kummalised kapivargad (kapp)

1 sek / 3 sek

40 punkti

Idee: Rocco Liiva, teostus ja lahenduse selgitus: Sandra Schumann

Antud on numbritest 0 kuni 9 koosnev jada pikkusega M ja arv N . Kui palju osajadasid pikkusega N saab koostada esialgsest jadast, kui igast järjestikuste võrdsete numbrite plokist peab osajadas olema vähemalt üks number? Vastuse võib anda mooduli 2^{31} järgi.

Sellele ülesandele on võimalik läheneda järk-järgult. Vaatame esimest alamülesannet:

Vastus on alati 0 või 1.

See tähendab, et peame leidma, kas osajada pikkusega N on võimalik antud tingimustel moodustada. See on võimatu kahel juhul. Üks on see, et N on suurem kui M . Teine on see, et jadas on numbriplokke rohkem kui N . Ehk siis esimese alamülesande jaoks piisab lahendusest, mis kontrollib, et N pole suurem kui jada pikkus ega väiksem kui numbriplokkide arvu.

Jätkame teise alamülesandega:

Jada koosneb ainult numbritest 0 ja 1, kusjuures nullid on enne ühtesid.

Nüüd peame mõtlema, kui palju erinevaid osajadasid on võimalik moodustada jadast kujul $0\dots01\dots1$. Kindlasti peab lõplikus jadas olema 0, kui alguses on vähemalt üks 0, ja 1, kui alguses on vähemalt üks 1. Võime seda võtta nii, et kummastki plokist on vaja võtta vähemalt üks number. Ülejäänud osajadas olevad numbrid saab võtta nullide ja ühtede seast vastavalt nende olemasolule. Võime alustuseks võtta osajadasse kõikideks ülejäänud numbriteks 0 (kui nulle piisavalt palju on) ja hakata nullide arvu järjest vähendama, kuniks neid jääb alles vaid üks või algselt jadast number 1 otsa saab.

Mõtleme nüüd kolmanda alamülesande peale:

$$N, M \leq 20$$

Siin alamülesandes on arvud N ja M piisavalt väikesed, et oleks võimalik kõik juhud läbi proovida. Üks võimalus seda teha on nii: igast numbriplokist on osajadasse vaja võtta vähemalt üks number. Võime luua rekursiivse funktsiooni, mis igast plokist võtab osajadasse esimese numbri ja iga järgmise numbri puhul vaatab rekursiivselt kaht varianti: sellist, kus number on osajadasse võetud, ja sellist, kus ei ole. Seejuures tuleb jälgida, et ka viimasest plokist jõuaks osajadasse vähemalt üks number. Niimoodi saadud osajadad võime duplikaatide eemaldamiseks kõik lisada ühte hulka (nt Pythoni `set` või C++ `unordered_set`) ja lõpuks vaadata selle hulga suurust. Nii toimib näiteks lahendus `sol_brute.py`.

Järgmine alamülesanne enam naiivse lähenemisega läbi ei lähe:

Vastus on väiksem kui 2^{31} .

See alamülesanne on lisatud puhtalt selleks, et kui lahendajale on võõras sageli raskemates ülesannetes kasutatav idee, et arve võiks väljastada mingi mooduli järgi, siis oleks võimalik ikka

märkimisväärne osa punkte kätte saada. Muus osas on tegemist sisuliselt täislahendust nõudva alamülesandega. Kuidas seda lahendada?

Eelmise alamülesande lahenduse suurim viga on see, et paljusid võimalikke osajadasid vaadatakse mitu korda. Lisaks ei ole meil mingit vajadust kõiki võimalikke jadasid mees pidada: võime selle asemel lugeda kohe erinevate jadade arvu. Siinkohal tuleb appi dünaamiline plaanimine (DP).

Esimese asjana võime ignoreerida seda, mis konkreetset sümboolid esinevad kogu jadas. Meid huvitab ainult see, kui mitu korda on sama numbrit järjest igas plokis. Kuna eri plokkides on eri numbrid ja igast plokist on lõplikus osajadas vähemalt üks number, siis pole vahet, kas vahelduvad plokid nullidest ja ühtedest või mingitest muudest numbritest — lõplike osajadade arv sellest ei muutu. Seega loeme kokku lihtsalt plokkides olevate numbrite arvud.

Selle käigus saame ka kohe kontrollida, ega plokkide pole rohkem kui N .

Mõned elemendid jadas on kohustuslikud: igast plokist vähemalt üks number. See jätab meile N miinus plokkide arv vabu kohti, mida täita eri viisidel.

Edasi teeme järgmist: vaatame i -ndat plokki ja tahame teada, kui mitmel viisil saame me osajada alustada nii, et j vaba kohta algusest loetuna oleks täidetud selles ja eelmistes plokkides olevate numbritega. i -ndast plokist on meil võimalik vabadele kohtadele panna minimaalselt 0 numbrit (lisaks siis ühele kohustuslikule). Maksimaalselt võime aga täita kõik j vaba kohta või siis nii palju, kui palju selles plokis (lisaks ühele kohustuslikule) veel numbreid on — oleneb, kumb piir enne ette tuleb.

Tähistame võimalikku osajadade arvu, kus on täidetud esimesed j kohta ning selleks kasutatud kuni i -ndat plokki kui $DP(j, i)$. Siis:

$$\begin{aligned} DP(j, i) = & \\ DP(j, i-1) + & \quad (\text{st } i\text{-ndast plokist on lisatud 0 numbrit}) \\ DP(j-1, i-1) + & \quad (\text{st } i\text{-ndast plokist on lisatud 1 number}) \\ DP(j-2, i-1) + & \quad (\text{st } i\text{-ndast plokist on lisatud 2 numbrit}) \\ \dots & \end{aligned}$$

Nii saame jätkata, kuni täidame kõik j vaba kohta i -nda ploki numbritega või plokis olevad numbrid otsa saavad.

Valemist on näha, et DP tabeli iga veeru elemendid sõltuvad ainult eelmise veeru elementidest. Seega saame DP tabeli veergude kaupa täita ja siis väljastada vastusena tabeli viimase veeru viimases reas oleva elemendi. Nõnda toimib näiteks lahendus `sol_dp.py`, mis lahendab tegelikult ära ka viimase alamülesande.

Mõnes muus keeles võib aga komistada täisarvude ületäitumise otsa ning sellest ka viimane alamülesanne:

Lisapiirangud puuduvad.

Järjestikuste liitmiste tulemusena võivad meie tabelis olevad arvud minna väga suureks ning seetõttu ei pruugi nad enam ära mahtuda C++ ja teise sarnaste keelte andmetüüpide `int`, `unsigned int`, `long long int` ega ka `unsigned long long int` piiresse. Seetõttu tuleks iga kord kaht arvu kokku liites tulemus leida mooduli 2^{31} järgi. Paneme tähele, et selleks, et liitmise tulemus juba kohe ületäitumist ei tekitaks, võiks tabeli arvude andmetüübiks olla vähemalt `unsigned int`. Sellist lahendust võib lugeda näiteks failist `sol_dp.cpp`.

Märkus. Tähelepanelik vaatleja avastab, et 2^{31} ei ole valitud juhuslikult — kuna tegemist peaks olema lihtsama ülesandega, siis on antud moodul ja ülesande paljusid eri väljundeid lubav sõnastus valitud selliselt, et nii 32-bitise märgiga täisarvu kui ka suuremate 2-süsteemil põhinevate arvutüüpide loomuliku ületäitumise korral loetakse väljastatud vastus siiski õigeks. Kui moodul oleks midagi muud (näiteks rahvusvahelistel võistlustel populaarne $10^9 + 7$), oleks `sol_dp.cpp` lahenduse ilmutatud kujul jääkide arvutamine hädavajalik.

4. Kõrgeim torn (torn)

1 sek / 3 sek

60 punkti

Idee, teostus ja lahenduse selgitus: Semjon/Sona Kravtšenko

Antud risttahukakujuliste klotside jada pikkusega $N \leq 1\,000$. Leida maksimaalse kindlalt püsiva ortogonaalselt pööratud klotside osajadast (samas järjekorras) torni kõrgus. Alamülesanded: $N \leq 12$ ja/või kõik klotsid on kuubid.

Naiivne viis selle ülesande lahendamiseks on vaadata läbi kõik klotside osajadad koos kõikide klotside orientatsioonidega, jättes vahele need kombinatsioonid, mis ei anna kindlalt püsivat torni. Läbivaatuseks sobib hästi rekursioon, kuna siis pole vaja kirjutada koodi, mis genereerib kõiki osajadasid ja orientatsioone, vaid piisab kirjeldamisest, milliseid tehteid saab ühe klotsiga teha. Rekursiooni tehes tuleb pidada meeles praeguse toetuspinna mõõtmeid (alguses laua mõõdud) ja praegust torni kõrgust (alguses 0). Olgu käesoleva klotsi mõõdud A_i , B_i ja C_i , praeguse toetuspinna mõõdud (X, Y) ning senini ehitatud torni kõrgus H . Osutub, et saab teha neli tehet:

1. Kasutada klotsi nii, et uus torni kõrgus oleks $H + A_i$. Siis on toetuspind (B_i, C_i) . See on lubatud vaid siis, kui tahk mõõtudega (B_i, C_i) mahub alale (X, Y) . Kuna mitte-ortogonaalne keeramine pole lubatud, mahub tahk alale üksnes juhul, kui ala pikem serv pole väiksem tahu pikemast servast ning ka ala laua lühem serv pole väiksem tahu lühemast servast.
2. Uus torni kõrgus on $H + B_i$ ja uus toetuspind (A_i, C_i) . See on lubatud vaid siis, kui (A_i, C_i) mahub (X, Y) -sse.
3. Uus kõrgus on $H + C_i$ ja uus toetuspind (A_i, B_i) . Lubatud, kui (A_i, B_i) mahub (X, Y) -sse.
4. Klots jäetakse vahele. Torni kõrgus ega toetuspind ei muutu.

See lahendusviis on piisav, alamülesannetes 1 ja 2, kuna (sõltuvalt detailidest) võib selle keerukus olla näiteks $O(4^N)$. Üks võimalik implementatsioon on toodud failis `sol_comb.cpp`.

Kuna paigutamise lubatavuse kontroll on keeruline, on ülesandes alamülesannetes 1 ja 3 sätestatud, et $A_i = B_i = C_i$. See tähendab, et kõik klotsid on kuubid. Siis mahub üks klots teise peale, kui alumise klotsi mõõt pole väiksem kui ülemise klotsi mõõt. Esimese klotsi puhul tuleb selle mõõtu võrrelda laua lühema servaga, nagu on tehtud failis `sol_comb_cube.cpp`.

Täislahendus eeldab, et klotsi paigutamist tuleb katsetada vaid üks kord. Selleks tuleb määrata, millised on maksimaalsed võimalikud tornide kõrgused, kui klots on asetatud tahule (A_i, B_i) , (A_i, C_i) või (B_i, C_i) . Selleks võib vaadata läbi kõik juba saavutatud olekud (H, X, Y) (mida peab jooksvalt salvestama). Esimene salvestatud olek on $(0, W, L)$, kus W ja L on laua mõõdud. Kui iga klotsi orientatsiooni kohta salvestada vaid parema kõrgusega olekuid, tekitab iga klots ülimalt 3 uut salvestatud olekut, mida tuleb iga järgmise klotsi paigutamiseks läbi vaadata. Seega selle lahenduse keerukus on $O(N^2)$. Lisaks tuleb hoolikalt jälgida, et mitte salvestada uut olekut juhul, kus klotsi ei õnnestunudki ühelegi varasemale olekule paigutada, ning ka seda, et klotsi ei lisataks olekule, mis on saavutatud sama klotsi endaga (ehk klotsi ei tohi panna iseenda peale). Täislahenduse üks võimalik implementatsioon on toodud failis `sol.cpp` ning samal ideel põhinev lahendus, mis töötab vaid kuupide jaoks, on failis `sol_cube.cpp`.

Boonusülesanne: kirjutada $O(N \cdot \log(N))$ programm, mis lahendab sama ülesannet kuupide jaoks.

Lisaküsimus: kuidas tuleb lahendust muuta, et see lubaks klotse panna mitte-ortogonaalselt (st. mistahes nurga all)?

Veel üks lisaküsimus: kas on võimalik seda lahendust kohandada viiemõõtmelistest klotsidest torni kõrguse leidmiseks? Mitme mõõtmeline on sel juhul laua "pindala"?

5. Vale kuju 2 (kuju2)

2 sek / 3 sek

80 punkti

Idee: Phil Labuschagne ja Ahto Truu, teostus ja lahenduse selgitus: Ahto Truu

Antud $A_1 < A_2 < \dots < A_N$ ja $B_1 \leq B_2 \leq \dots \leq B_M$, kus $M, N \leq 25\,000$. Leida iga i jaoks vähim j , mille korral $B_i \leq A_j$, kasutades võimalikult vähe päringuid kujul “kas $B_i \leq A_j$?”

Esimene võimalus selle ülesande lahendamiseks on püstitusest lähtuv naiivne algoritm: proovime iga punni järjest aukudesse $1, 2, \dots$, kuni leiame esimese, mis pole liiga väike. Nii on tehtud failides `sol1_naive.py` ja `sol1_naive.cpp` toodud lahendustes, mis vaatavad iga i väärtuse jaoks läbi keskmiselt pooled (ja halvimal juhul isegi kõik) j väärtused ning kulutavad selleks kokku $O(M \cdot N)$ operatsiooni. See on piisav, et lahendada esimene alamülesanne, aga edasi võivad M ja N olla nii suured, et see lahendus ei mahu ajalimiidi piiridesse. 2. ja 3. alamülesande $1\,000 \cdot 25\,000 = 25\,000 \cdot 1\,000 = 25\,000\,000$ operatsiooni oleks ajalimiidi piires tehtavad, kui andmed oleks lahenduse enda mälus, aga hindamisprogrammiga suhtlemine on märksa aeglasem.

Üks võimalus parema lahenduse saamiseks on kasutada ära, et augud on kasvavalt sorteeritud, ja leida iga i jaoks sobiv j väärtus kahendotsinguga. Kahendotsing kulutab N -elemendilises jadas õige väärtuse leidmiseks $O(\log(N))$ võrdlemist ja seega kulub kokku $O(M \cdot \log(N))$ operatsiooni. Failides `sol2_bin.py` ja `sol2_bin.cpp` toodud lahendused seda teevadki ja teenivad punktid nii 1. kui ka 2. alamülesande eest.

Teine võimalus on kasutada ära, et punnid on sorteeritud, ja teha iga augu jaoks kahendotsing punnide hulgas. Selle juhu teeb natuke keerulisemaks asjaolu, et punnide diameetrid ei tarvitse unikaalsed olla ja siis peab olema hoolikas, et leida iga augu jaoks kõik punnid, mis sinna auku käivad. Failides `sol3_bin.py` ja `sol3_bin.cpp` toodud lahendused lähenevad sellele nii, et leiavad iga augu jaoks esimese punni, mis sinna *ei mahu* ja teevad siis järelduse, et kõik väiksemad punnid peavad mahtuma. Eelmise juhuga sarnaselt kulutavad need lahendused $O(\log(M) \cdot N)$ operatsiooni ja teenivad punktid nii 1. kui ka 3. alamülesande eest.

Kolmas võimalus on kasutada ära, et mõlemad jadad on sorteeritud. See tähendab, et kui me oleme juba leidnud, et punn i peab minema auku j , siis teame, et $i + 1$ jaoks sobiv j väärtus ei saa olla väiksem. Kuigi failides `sol4_merge.py` ja `sol4_merge.cpp` toodud lahendustes on kaks kordust üksteise sees, tasub tähele panna, et sisemine `while`-kordus alustab iga kord sealt, kus eelmine `while`-kordus lõpetas. Kui `for`-kordus käib läbi M erinevat i väärtust, käivad kõik `while`-korduse käivitused kokku läbi N erinevat j väärtust ja seega kulutab selline kahe jada paralleelse läbimise algoritm $O(M + N)$ operatsiooni. Sellega teenivad need lahendused punktid nii 1. kui ka 4. alamülesande eest, aga 2. ja 3. alamülesandes nad maksimumpunkte ei saa, sest kui M on palju väiksem kui N , siis on $M \cdot \log(N)$ väiksem kui $M + N$, ja sarnaselt ka siis, kui N on palju väiksem kui M .

Üks võimalus maksimumskoori teenimiseks oleks kirjutada kolm eelkirjeldatud lahendust ja need kõik esitada. Siis saaks igas alamülesandes vähemalt üks lahendus täispunktid ja kolme lahenduse peale kogunekski ülesande eest kokku maksimumskoor.

Teine võimalus täispunktide saamiseks on kaks kahendotsingut kombineerida: kui teeme aukude hulgas esimese kahendotsingu mitte kõige väiksema ega kõige suurema, vaid keskmise punni jaoks, jagab leitud i - j paar kaheks nii punnid kui augud; seejärel saame eraldi leida vastavused väiksemate punnide ja väiksemate aukude ning suuremate punnide ja suuremate aukude vahel, nagu on tehtud failides `solx_binbin.py` ja `solx_binbin.cpp` toodud lahendustes. Kui eeldame, et keskmisele punnile $i = M/2$ vastav auk j jagab ühe kahendotsinguga ka aukude hulga umbes

pooleks ($j \approx N/2$), saame selle algoritmi tööaja $T(M, N)$ hindamiseks rekurrentse võrrandi $T(M, N) \approx \log_2(N) + 2 \cdot T(M/2, N/2)$. Selle võrrandi lahendamine oleks päris keeruline, aga eksperimendid näitavad, et kui M ja N on ligikaudu võrdsed, kulutab see lahendus ainult mõned päringud rohkem kui kahe kahe jada paralleelne läbimine, ja kui M ja N on oluliselt erinevad, siis umbes poole vähem päringuid kui ühekordne kahendotsing.

6. Mugandatud ahel (ahel)

1 sek / 3 sek

100 punkti

Idee, teostus ja lahenduse selgitus: Andres Alumets

On antud tippude ahel pikkusega N , millele on lisatud üks serv mingi kahe tipu vahele. Vaja on leida iga arvu $k \in 1, 2, \dots, N$ jaoks, mitu sellist tipupaari leidub, mille vahel lühima tee pikkus on k .

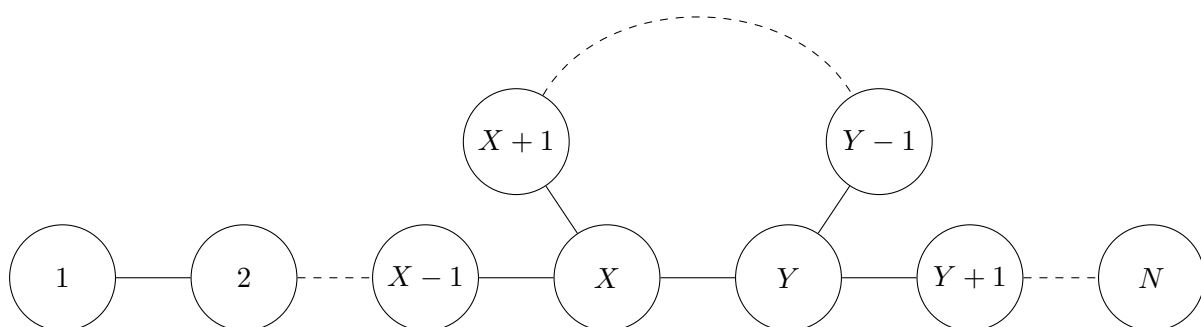
Alamülesanded: Ahela pikkus kuni 100, kuni 5 000, kuni 100 000.

Vaatame võimalikke lahendusi alamülesannete kaupa.

Esimeses alamülesandes, kus $N \leq 100$, mahub ajalimiiti lahendus, mille keerukus on $O(N^3)$. Sellise keerukusega on näiteks Floyd-Warshalli algoritm. Saame teha $N \times N$ tabeli ja kõigi tipupaaride vahelised kaugused välja arvutada. Pärast saab sellest tabelist iga kauguse kordused kokku lugeda. Tuleb tähele panna, et iga tipupaari tuleb lugeda ainult ühe korra. Seega õige vastuse saamiseks tuleb tabelis esinemiste arv jagada 2-ga.

Teises alamülesandes, kus $N \leq 5\,000$, mahub ajalimiiti lahendus, mille keerukus on $O(N^2)$. Selline lahendus on näiteks kõikide tipupaaride vaatlemine. Näiteks vaatame tipupaari a, b , kus $a < b$. Kui mõlemad on väiksemad lisatud serva väiksemast otpunktist, siis nende vaheline kaugus on $|b - a|$. Samamoodi ka siis kui mõlemad on suuremad lisatud serva suuremast otpunktist. Kui a on väiksem kui lisatud serva väiksem otpunkt ja b on suurem kui lisatud serva suurem otpunkt, siis läbib lühim tee kindlasti lisatud serva ja on võimalik leida kaugus tipust a lisatud servani ja lisatud servast tipuni b . Ülejäänud juhud on sellised, kus vähemalt üks kahest tipust on graafi ainukesel tsükliks. Siis saab leida kaugused kui minna mööda üht tsükli poolt või mööda teist ja võtta nendest miinimum. Sellise lahendusega peaks kätte saama 50 punkti.

Kolmandas alamülesandes, kus $N \leq 100\,000$, on oodatav lahendus keerukusega $O(N)$. Jätkame ideed, kus jagame graafi osadeks. Kindlasti on selleks jagamiseks mitmeid võimalusi, kuid näidislahenduses jagame näiteks nii: ütleme et uus serv on tehtud tipust X tippu Y , kus $X < Y$. Siis esimene osa on tipud $1, 2, \dots, X$, teine Y, \dots, N ja kolmas $X + 1, X + 2, \dots, Y - 1$.



Paneme tähele et esimene ja teine osa moodustavad ühe ahela. Olgu selle ahela pikkuseks m . Paneme tähele et tipupaare kaugusega 1 on seal $m - 1$, paare kaugusega 2 on $m - 2$ jne. Seega saame $O(N)$ ajas leida iga kauguse kohta nende arvu.

Nüüd vaatame juhtu, kus üks tipp on esimeses osas ja teine kolmandas. Lihtsuse mõttes võib kolmandasse ossa lisada ka tipu Y . Siis igal esimese osa tipul on mingil kaugusel kolmandas osas alati kas 2 või 0 tippu (kui kaugus on kas liiga suur või liiga väike). Erijuht on siis, kui tsükli pikkus on paaritu. Siis on ühe kauguse puhul kolmandas osas ainult 1 tipp.

Seega idee on teha esimese osa peal libisev aken, kus kõigil aknas olevatel tippudel on mingil kaugusel vähemalt üks tipp kolmandas osas. Algul on aknas ainult tipp X ja kaugus 1. Iga järgmise kaugusega liigub akna väiksem ots tipu 1 poole ja ülemine ots jääb paigale. Ülemine ots hakkab vähenema siis, kui ülemise otsa juures oleval tipul kolmandas osas enam vaadeldaval kaugusel ühtegi tippu pole. Nii saame lihtsalt akna pikkuse järgi õige kaugusega paaride arvu kokku lugeda.

Sarnane lahendus on ka teise ja kolmanda osa kohta.

Kui mõlemad tipud on kolmandas osas, siis tsükli peal on igal tipul täpselt 2 paarilist iga kauguse kohta mis on väiksem kui pool tsükli pikkusest. Seega ka nende kokku lugemine on kiiresti tehtav.

Ülesandes on lisaks veel mõned tehnilised keerukused. Lohakalt lahendades võib juhtuda et mõnda paari loetakse mitmekordselt. Seega peab pidevalt ka seda jälgima ja vajadusel mõne tipu mitmekordselt loetud paarilised maha lahutama.