Sisukord

1. Topsimäng (`topsid`)

1 sek / 3 sek 20 punkti

Idee: Tähvend Uustalu; teostus ja lahenduse selgitus: Targo Tennisberg

Laual on N topsi, mille seas korraldatakse hulk paarikaupa vahetusi.

Leida algselt etteantud topsi asukoht pärast kõiki vahetusi.

Lahenduse idee

Esimene mõte on võib-olla, et loeks kõik topsid massiivi või loendisse ja seejärel hakkaks seal ümbertõstmisi tegema. Tegelikult piisab aga täielikult sellest, kui lihtsalt kogu aeg meeles pidada, millise topsi all pall on. Teised topsid ei ole üldse tähtsad, samuti pole oluline ka arv N. Efektiivne lahendus on tsükkel, mis vaatab kõik M vahetust läbi ning kontrollib iga vahetuse kohta, kas vahetus mõjutab topsi, mille all pall parajasti on.

Arvestada tuleb kõigi võimalustega: pall võib olla esimese topsi all, teise topsi all või mitte kummagi all, mõned lahendused jätsid viimase variandiga arvestamata.

Implementatsioon

Üks lahendusvõimalus on näiteks selline:

```
for _ in range(m):
    # loeme vahetuse andmed
    a, b = [int(v) for v in input().split()]

    if p == a:
        # kui enne on pall esimese vahetatava topsi all
        # siis pärast on see teise vahetatava topsi all
        p = b
    elif p == b:
        # kui enne on pall teise vahetatava topsi all
        # siis pärast on see esimese vahetatava topsi all
        p = a
```

Tasub tähele panna, et teises tingimuses `elif` asemel lihtsalt `if` kasutamine oleks vale: esimese tingimuse kehtivuse korral omistamise `p = b` järel kehtib kindlasti ka teine tingimus ja siis tehtav omistamine `p = a` nullib eelmise omistamise efekti!

2. Edetabel (`tabel`)

1 sek / 3 sek 30 punkti

Idee: Tähvend Uustalu; teostus: Sandra Schumann; lahenduse selgitus: Ahto Truu

> Antud N võistleja nimed ja punktisummad. Koostada pingerida, kus on näidatud iga võistleja koht, jagamise korral jagatavad kohad.

Naiivne viis selle ülesande lahendamiseks on järgmine:

1. Leiame kõigepealt maksimaalse nimekirjas esineva skoori.
2. Siis leiame kõik selle skoori saanud osalejad; olgu nende arv M.
3. Kui $M = 1$, siis on see osaleja üksi 1. kohal.
   Kui $M > 1$, siis jagavad need osalejad 1. kuni M. kohta.
   Väljastame leitud grupi ja eemaldame need osalejad nimekirjast.
4. Kui nimekiri sai tühjaks, on kogu pingerida väljastatud ja võime lõpetada.
   Kui nimekirja jäi veel osalejaid, siis alustame otsast (ainult järgmine grupp algab 1. koha asemel kohast $M + 1$).

Sellised lahendused on toodud failides `sol_naive_loop.py` ja `sol_naive_loop.cpp`.

Koodiridade vähendamiseks võib neis lahendustes käsitsi kirjutatud korduste asemel kasutada ka keele standardvahendeid. Näiteks Pythonis võib

```
skoor = -1 # väiksem kui ükski sisendis lubatud punktisumma
for p, s in tabel:
  if p > skoor:
    skoor = p
```

asemel kirjutada lühemalt

```
skoor = max(p for p, s in tabel)
```

ja C++ lahenduses

```
int skoor = -1; // väiksem kui ükski sisendis olev punktisumma
for (const auto & [p, s] : tabel)
  if (p > skoor)
    skoor = p;
```

asemel

```
auto skoor = ranges::max(tabel
  | views::transform([](const auto & x) { return x.first; }));
```

Analoogiliselt saab ka teised kordused suuremalt jaolt asendada standardvahenditega ja tulemuseks on failides `sol_naive_std.py` ja `sol_naive_std.cpp` toodud lahendused.

Nii käsitsi kirjutatud korduste kui ka standardvahendite kasutamisel on selle lahenduse puudus, et mõnel juhul tuleb osalejate nimekirja väga palju kordi läbi vaadata. Maksimaalselt on selles ülesandes ühes testis 100 000 osalejat. Kui neil on kõigil erinevad skoorid, siis vaatame võitja leidmiseks läbi kõik 100 000 osalejat, teiseks tulnu leidmiseks kõik ülejäänud 99 999 osalejat, kolmandaks jäänu leidmiseks ülejäänud 99 998 osalejat jne. Kokku peame seega tegema $100\,000 + 99\,999 + 99\,998 + \ldots + 1 = 5\,000\,050\,000$ operatsiooni, mis kindlasti ajalimiidi sisse ei mahu.

Lahenduse kiirendamiseks on põhiline idee grupeerida samade skooridega osalejad kokku. Üks võimalik viis selle saavutamiseks on nimekiri sisse lugemise järel skooride järgi ära sorteerida. Siis on võrdse skooriga osalejad nimekirjas järjest.

Lihtne viis iga osaleja koha või kohtade leidmiseks on sellest osalejast alustades vaadata esiteks nimekirja alguse poole, et leida kõige varasem sama skooriga osaleja, ning teiseks nimekirja lõpu poole, et leida kõige hilisem sama skooriga osaleja, nagu on tehtud failides `sol_sort_slow.py` ja `sol_sort_slow.cpp`.

Kahjuks jäävad need lahendused liiga aeglaseks testides, kus on palju kohtade jagamisi. Oletame, et ühes testis on kõigil osalejatel sama skoor. Siis tuleb iga osaleja jagatavate kohtade leidmiseks vaadata läbi kõik temast nimekirjas eespool ja kõik temast nimekirjas tagapool olevad võistlejad, ehk kogu nimekiri! N osalejaga testis tuleb seega teha iga osaleja kohta N operatsiooni. Maksimaalse lubatud $N = 100\,000$ korral on see $100\,000 \cdot 100\,000 = 10\,000\,000\,000$ operatsiooni, mis on isegi halvem kui eelmise naiivse lahendus tulemus!

Sorteerimisest tegelikult kasu saamiseks tuleks iga grupp leida ainult üks kord ja siis väljastada järjest kõik selle grupi liikmed. Üks viis seda teha on näha failides `sol_sort_ok.py` ja `sol_sort_ok.cpp`. Nendes lahendustes leitakse sorteeritud nimekirjas esikohta jagavad osalejad põhimõtteliselt samamoodi kui eelmises lahenduses. Seejärel aga kasutatakse teadmist, et need M osalejat jagavad kõik 1. kuni M. kohta ja väljastatakse kohe M rida, kus on samad kohad, aga erinevad nimed. Seejärel jätkatakse järgmise grupi suuruse tuvastamisega, alustades osalejast $M + 1$. Nii vaatame iga osaleja andmeid ainult kaks korda: ühe korra grupi suuruse leidmisel ja teise korra selle osaleja rea väljastamisel.

Teine viis sama skooriga osalejate kokku kogumiseks on kasutada sõnastiku andmetüüpi (Pythonis `dict`, C++ standardteegis `map`). Kui teeme sõnastiku, milles võtmed (ingl *key*) on skoorid ja neile vastavad väärtused (ingl *value*) selle skoori saanud osalejate nimed, on vastuse väljastamine imelihtne: käime sõnastiku skooride järjekorras läbi; iga skoori kohta teame, et seal jagavad kohti just nii palju osalejaid, kui on sellele skoorile vastava nimekirja pikkus, ja nimekirja ennast läbi käies saame väljastda igale osalejale vastava rea edetabelis. Jälle tegeleme iga osalejaga täpselt kaks korda: ühe korra teda sõnastikus õigesse nimekirja lisades ja teise korra tema nime väljastades. Sellised lahendused on toodud failides `sol_dict_ok.py` ja `sol_dict_ok.cpp`.

Pythonis programmeerijad peavad `dict` kasutamisel tähele panema, et vakimisi pole selle elementide läbimise järjekord määratud. Nii saaksime korduste

```
for p in tabel:
  for s in tabel[p]:
    ...
```

korral osalejad väljundisse küll nii, et võrdse skoori saanud on kõrvuti, aga grupid ei tarvitse omavahel punktide järjekorras järjestatud olla. Õige tulemuse saamiseks on oluline läbida sõnastikku sorteeritud järjekorras:

```
for p in sorted(tabel):
  for s in tabel[p]:
    ...
```

C++ kasutajatel selle pärast eraldi muretsede vaja ei ole, sest `map` on juba vaikimisi järjestatud ja kordused

```
for (auto const & [p, list] : tabel)
  for (auto const & s : list)
    ...
```

annavad tulemuseks skooride järgi järjestatud grupid.

3. Sentide ümardamine (`sendid`)

1 sek / 1 sek 40 punkti

Idee: Tähvend Uustalu; teostus ja lahenduse selgitus: Birgit Veldi

> Antud N toodet ning nende hinnad eurodes ja sentides.
>
> Vaja on jagada tooted ostudeks nii, et pärast ostude summade lähima viie sendini ümardamist kuluks kokku võimalikult vähe raha.

Lahenduse idee

Mõtleme, kui palju on üldse võimalik ühe ostuga säästa (s.t. maksta vähem kui päris hind) või siis juurde maksta. Kuna summa ümardatakse lähima viie sendini, saab Juta säästa 2 või 1 senti, jääda tasa või kaotada 1 või 2 senti ning seda sõltuvalt sentide arvu jäägist 5-ga jagamisel.

Seega ühe ostu pealt saab säästa ülimalt 2 senti ning selleks peab toodete sentide summa andma 5-ga jagades jäägi 2. Ideaalis tahaks Juta teha võimalikult palju selliseid oste, millega säästab palju. Et teha võimalikult palju kasulikke oste, peaks ühes ostus olema pigem vähe tooteid.

Paneme ka tähele, et 5-ga jaguvate sentide arvuga toodete puhul pole vahet, kas osta neid mingite toodetega koos või üksi, kuna need ei muuda ühegi summa ümardamist ega anna ka ise kasu ega kahju juurde.

Vaatame järgnevalt, kuidas üldse on võimalik väheste toodetega 2 senti säästa.

Ühe toote puhul on see võimalik, kui toote sentide arv annab 5-ga jagamisel jäägi 2.

Paremuselt järgmine säästmisvõimalus oleks säästa 2 tootega 2 senti või 1 tootega 1 sent. Kasutades tooteid, mille sendid annavad jäägi $1, 3$ või 4, on see võimalik, kui ostu sentide jääkide summa on kas $3 + 4$, $1 + 1$ või 1.

Selle põhjal saaks tooted jääkidega $0, 1$ ja 2 ära jaotada. Mida aga teha, kui tooted jääkidega 3 ja 4 ei jagu paaridesse?

Vaatame, kuidas nende abil veel väheste toodete abil säästa saab. Kolme tootega saab 2 senti säästa, ostes kolm toodet jääkidega 4 (s.t. ost $4 + 4 + 4$). Nelja tootega saab 2 senti säästa, ostes neli toodet jäägiga 3, mis on sama hea, kui osta kask korda kaks toodet jäägiga 3 (s.t. kumbki ost $3 + 3$).

Kuidas aga otsustada ostude $3 + 4$, $4 + 4 + 4$ ja $3 + 3$ täpne jaotus? Paneme tähele, et ostud $3 + 3$ ja $4 + 4 + 4$ võime ümber jaotada ostudeks $3 + 4$, $3 + 4$ ja 4. Esimesel juhul säästame $1 + 2 = 3$ senti. Teisel juhul säästame samuti kokku 3 senti: säästame $2 + 2 = 4$ ning maksame juurde 1. Seega võiks teha nii palju paare $3 + 4$, kui saab.

Lahenduse algoritm: Juta võiks osta tooted jääkidega $0, 1$ ja 2 eraldi. Siis võiks ta moodustada võimalikult palju paare jääkidest 3 ja 4 (s.t., kui kolmesid ja neljasid pole võrdselt, jäävad üle vaid ühed neist jääkidest). Kui üle jäävad neljad, moodutaks neist võimalikult palju kolmikuid. Kui üle jäävad kolmed, moodustaks neist võimalikult palju paare. Ning ülejäänud tooted ostaks jälle üksinda.

Tõestus, et see annab minimaalse summa

Näitame nüüd, et nii saab tõepoolest alati vähima võimaliku summaga ostud.

Olgu meil mingisugune ostudeks jaotus, mille korral saavutatakse vähim võimalik summa. Näitame, kuidas see ilma summat suurendamata eelnevalt kirjeldatud jaotuseks ümber grupeerida.

Ostu $X = \{x_1, \dots x_m\}$ (s.t. ost X koosneb toodetest $x_1, \dots, x_m$) tegelikku summat (sentides) tähistame $S_T(X)$ ning sularahaga makstavat summat $S_R(X)$.

> Kui meil on ost $X = \{a_1, \dots, a_\ell, b_1, \dots, b_m\}$ ning toodete $a_1, \dots, a_\ell$ sentide jääkide summa annab 5-ga jagades jäägi $0, 1$ või 2, siis võime selle ostu jagada kaheks ostuks: $A = \{a_1, \dots, a_\ell\}$ ja $B = \{b_1, \dots, b_m\}$ ning siis $S_R(X) \geq S_R(A) + S_R(B)$.

Tõestus. Kui $S_T(A)$ jagub 5-ga, siis nii koos kui eraldi makstes määrab säästetud või suure makstud summa $S_T(B)$ sentide arvu jääk 5-ga jagamisel.

Kui $S_T(A)$ annab 5-ga jagades jäägi 1 või 2, siis vaatame kõiki võimalusi:

$S_T(B)$ jääk	$S_R(A) + S_R(B)$	$S_T(X)$ jääk	$S_R(X)$
0	$S_T(A) + S_T(B) - 1 = S_T(X) - 1$	1	$S_T(X) - 1$
1	$S_T(A) - 1 + S_T(B) - 1 = S_T(X) - 2$	2	$S_T(X) - 2$
2	$S_T(A) - 2 + S_T(B) - 1 = S_T(X) - 3$	3	$S_T(X) + 2$
3	$S_T(A) + 2 + S_T(B) - 1 = S_T(X) + 1$	4	$S_T(X) + 1$
4	$S_T(A) + 1 + S_T(B) - 1 = S_T(X)$	0	$S_T(X)$

$S_T(B)$ jääk	$S_R(A) + S_R(B)$	$S_T(X)$ jääk	$S_R(X)$
0	$S_T(A) + S_T(B) - 2 = S_T(X) - 2$	2	$S_T(X) - 2$
1	$S_T(A) - 1 + S_T(B) - 2 = S_T(X) - 3$	3	$S_T(X) + 2$
2	$S_T(A) - 2 + S_T(B) - 2 = S_T(X) - 4$	4	$S_T(X) + 1$
3	$S_T(A) + 2 + S_T(B) - 2 = S_T(X)$	0	$S_T(X)$
4	$S_T(A) + 1 + S_T(B) - 2 = S_T(X) - 1$	1	$S_T(X) - 1$

□

Vaatame seda toodete jaotust, kus saavutatakse vähim võimalik kogusumma. Eelmise väite põhjal võime nüüd igast ostust eemaldada kõigepealt kõik tooted jääkidega $0, 1, 2$ eraldi ning paarid $3 + 4$, kuniks seda on võimalik teha. Seejuures me ei kaota raha (kuna eelnev summa oli ka juba minimaalne, siis ei võida ka, aga see pole siinkohal veel oluline).

Mis sai üldse alles jääda? Ainult tooted jääkidega 3 ja/või 4. Seejuures pole enam üheski allesjäänud ostus jääke 3 ja 4 mõlemat.

Eraldame võimalikult palju kolmikuid $4+4+4$ ning paare $3+3$. Nüüd saab alles saab jääda mingi hulk üksikuid kolmesid ja üksikuid või kahekaupa neljasid. Endiselt pole summa selle tegevuse käigus vähenenud.

Paneme tähele, et alles ei saa olla rohkem kui 1 toode jäägiga 3: kui neid oleks vähemalt 2, võiksime võtta need kaks koos ja säästa 1 sendi. Kuna praegu oleks need kaks eraldi, mis tähendaks $2 + 2 = 4$ sendi juurde maksmist, ning tegu oli vähima võimaliku summaga, pole selline olukord võimalik.

Samuti, alles ei saa olla nii kolm kui ka neljasid: ostud 3 ja 4 võiksime panna kokku ning säästa 2 senti varasemalt juurde makstud $2 + 1 = 3$ sendi asemel, kuid siis poleks olnud tegu enam vähima võimaliku jaotusega. Ostud 3 ja $4 + 4$ võiksime jaotada $3 + 4$ ja 4 säästes nii kokku 1 sendi varasema juurde makstud 3 sendi asemel. Seega, kui leidub 3, siis neljasid pole ning olemegi jõudnud soovitud jaotusesse.

Kui kolmesid alles ei jäänud, on alles ülimalt 2 nelja: kui leiduks rohkem, leiduks vähemalt üks komplekt oste:

- $4, 4, 4$
- $4, 4 + 4$
- $4 + 4, 4 + 4$

Neist iga komplekti puhul saaks need kokku pannes maksta tegelikult summast vähem, kuid praegu makstakse rohkem. Kuid eeldasime, et tegu oli vähima summaga jaotusega.

Seega on alles kas üks või kaks nelja, kas üksi või koos. Ning neist viimased võib eraldi tõsta, summa ei muutu ning nii jõudsimegi soovitud jaotuseni.

Implementatsioon

Tekitame endale kolm listi ning arvutame summat juba pooleldi paralleelselt sisendi lugemisega (et ei peaks eurode ja sentide täpset arvu enam hiljem kusagilt otsima): See tähendab, et kui lisame toote jäägiga 0, 1 või 2, siis arvutame jäägi jagu sente juba summast maha. Muude toodete puhul liidame esialgu algse summa.

```
yksikud = [] # tooted jääkidega 0, 1, 2
kolmed = [] #tooted jäägiga 3
neljad = [] #tooted jäägiga 4

for i in range(N):
    T, E, S = input().split()
    eurod += int(E)
    S = int(S)

    if S % 5 == 3:
        kolmed.append(T)
        sendid += S
    elif S % 5 == 4:
        neljad.append(T)
        sendid += S
    else:
        yksikud.append(T)
        sendid += S - (S % 5)
```

Järgmiseks arvutame paaride arvu ning igast paarist võtame 2 senti maha.

```
x, y = len(kolmed), len(neljad)
paarid = min(x, y)
sendid -= paarid * 2
```

Kui rohkem on jääke 3, siis vaatame, palju neist paare saab teha ning igast paari kohta võtame sendi maha. Kui üksik 3 jäi alles, tuleb 2 senti juurde liita.

```
if x > y:
    sendid -= (x - y) // 2
    if (x - y) % 2 == 1:
        sendid += 2
```

Kui rohkem on jääke 4, siis vaatame, palju neist kolmikuid saab teha ning igast kolmiku kohta võtame 2 senti maha. Kui mõni 4 jäi alles, tuleb nii mitu senti juurde liita.

```
if y > x:
    sendid -= ((y - x) // 3) * 2
    if (y - x) % 3 != 0:
        sendid += (y - x) % 3
```

Siis teisendame eurod ja sendid sobivale kujule ning prindime sama loogikaga ostud välja.

```
eurod += sendid // 100
sendid = sendid % 100
print(eurod, sendid)

for i in yksikud:
    print(i)
for i in range(paarid):
    print(kolmed[i], neljad[i])
if x > y:
    for i in range(paarid, x, 2):
        if i + 1 < x:
            print(kolmed[i], kolmed[i+1])
        else:
            print(kolmed[i])
elif y > x:
    for i in range(paarid, y, 3):
        if i + 2 < y:
            print(neljad[i], neljad[i+1], neljad[i+2])
        else:
            print(neljad[i])
            if i + 1 < y:
                print(neljad[i+1])
```

4. Seitse lammast (`lambad`)

1 sek / 1 sek 60 punkti

Idee: Heno Ivanov; teostus ja lahenduse selgitus: Ralf Robert Paabo

Antud 7 punkti tasandil.

Leida, kas eksisteerivad 3 sirget, mis eraldavad punktid üksteisest.

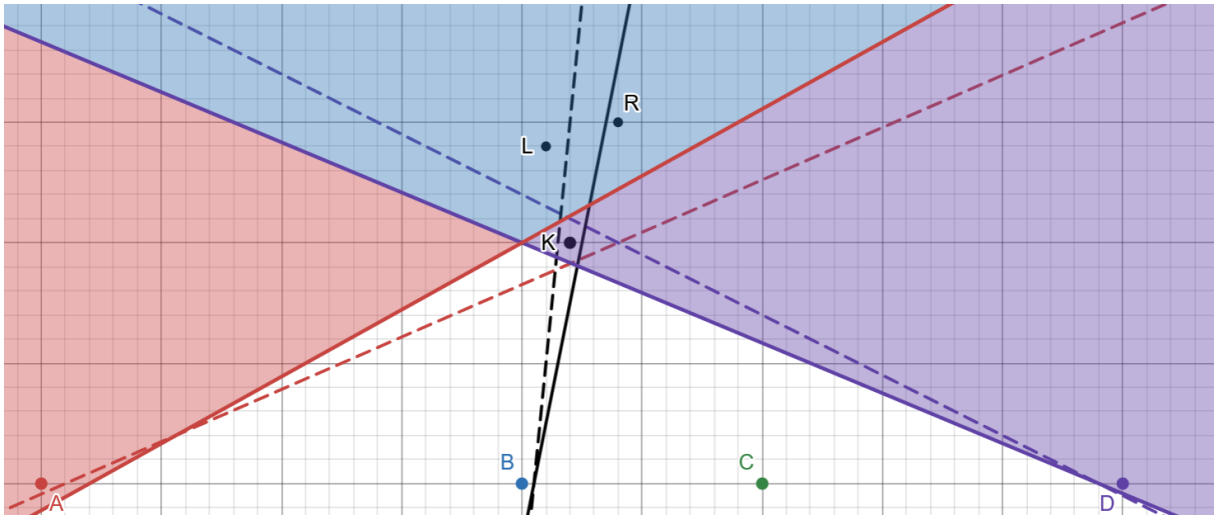
Alamülesanne: vähemalt neli punkti on x-teljel ning ülejäänud punktid on x-teljest kõrgemal.

Lahenduse idee

Alamülesande lahendus on üsna erinev täislahendusest, kuid võib aidata sirgete omadustest paremini aru saada ja seeläbi siiski täislahenduseni juhatada. Alamülesande lahenduse saab läbi teha pliiatsi ja paberiga ning ei vaja nii palju abstraktset mõtlemist kui täislahendus.

Alamülesande lahendus: Kolme sirge lõikumisel tekkivast seitsmest alast keskmine on ümbritsetud kõigi sirgetega ja on kolmnurga kujuga. Kuna see on ainuke täielikult ümbritsetud piirkond seitsmest, on hea hakata ka lahendust ehitama selle ümber. Keskendume edaspidi juhule, kus x-teljel on täpselt 4 punkti, sest on lihtne veenduda, et kui ühel sirgel oleks 5 või rohkem punkti, ei leiduks lahendust igal juhul. Koos teise tingimusega ($Y_i \geqslant 0$) saame olla kindlad, et ükski x-teljel olevatest punktidest ei ole keskmise piirkonna sees, sest vastasel juhul avaneks vähemalt üks piirkondadest allapoole (kus ei leidu alamülesande tingimuse järgi ühtegi punkti). Seega saame otsida keskmist punkti ülejäänud kolme punkti seast.

Sorteerime x-teljel asuvad punktid ära nende x-koordinaadi järgi (vt. joonist, punktid A, B, C, D). Nimetame keskpunkti kandidaadiks punkti K. Siis joonistame sirge (punane sirge), mis möödub punktist A paremalt ja punktist K vasakult või paremalt. Mõlemast punktist peab sirge mööduma väga lähedalt, et mitte mõjutada teisi punkte. Sarnaselt joonistame sirge punkti D lähedusse (lilla sirge). Mõlemad sirged peavad mööduma K-st vasakult või paremalt, seega leidub kaks sobivat sirgete paari (joonisel punktiiriga või pidevjoonega). Ülejäänud kaks punkti L ja R peavad olema ülemises sinises alas, sest punane ala on hõivatud punktiga A ning lilla ala punktiga D. Kuna sinine ala avaneb alati üles, järeldame, et ainuke võimalus on keskmiseks punktiks valida kolmest mustast punktist kõige madalam (vähima y-koordinaadiga) punkt.



Kuna me nüüd teame, milline punkt on keskmises alas, saame otsustada, kas lahendus leidub või mitte. Kui punktid L ja R on mõlemad sinises alas, on lahendus olemas, vastasel juhul mitte, sest punases või lillas alas oleks siis mitu punkti, mille eraldamine pole võimalik. Kolmas sirge (must) tuleb joonistada kusagilt L ja R vahelt kusagile B ja C vahele ning kuhu täpselt, pole isegi tähtis. Loeb ainult see, kas must sirge on punktist K paremal või vasakul, sest selle järgi tuleb valida lilla ja punase sirge paar. Joonisel toodud näites on mõlemad võimalikud (vt. punktiiriga lahendus ja pidevjoonega lahendus), kuid see ei pruugi alati nii olla.

Täislahendus: Paneme tähele, et iga õiges lahenduses kasutatav sirge jaotab tasandil olevad punktid kaheks nii, et ühel pool on 3 punkti ja teisel pool 4 punkti.

Selle tõestamiseks mõtleme korraks, mitme sirgega saab üksteisest eraldada seitsmest vähemaid punkte. 2 punkti saab selgelt eraldada ühe sirgega, kuid 3 jaoks on vaja juba kahte. 4 punkti on samuti enamasti võimalik eraldada kahe sirgega (kui need ei asu kõik samal sirgel), kuid 5 ja 6 punkti eraldamiseks läheb juba vähemalt kolm sirget. Seega tulles tagasi algse ülesande juurde, ei oleks mingil juhul mõttekas poolitada seitset punkti sirgega nii, et ühele poole jääb viis või kuus punkti, sest täielikuks eraldamiseks kuluks veel kolm sirget ehk kokku vähemalt neli sirget.

Leiame kõik võimalikud viisid jaotada 7 lammast kaheks nii, et ühel pool on 3 ja teisel pool 4 lammast. Iga jaotuse jaoks tuleb muidugi välja pakkuda ka üks sirge, mis seda teeb. Edasi on võimalik kontrollida kõik nende sirgete kolmikud. Selliseid kolmikuid on $O(N^3)$, kus N on nn. *kandidaatsirgete* arv, mis jääb maksimaalselt paarikümne kanti. Kandidaatsirgete leidmiseks käime üle kõik punktide paarid ja konstrueerime sirged, mis on väga lähedal nendele kahele punktile, kuid nihutatud natuke vasakule või paremale, et need antud punkte ei läbiks.

Implementatsiooni detailid

Nii täislahenduses kui ka alamülesande lahenduses on vaja konstrueerida sirgeid, mis on punktidele väga lähedal, kuid ei läbi neid. Selliste sirgete koostamiseks on olemas väga lihtne meetod. Alustame kahte punkti läbivast sirgest $ax + by + c = 0$. Muutes vaid parameetrit c ühe võrra saame algsega paralleelse, aga nihutatud sirge. Ülesande lahendus on piiratud täisarvudele, seega ühest väiksema nihke tegemiseks korrutame kõigepealt terve võrrandi läbi suure arvuga k. Kui k on suurem kui maksimaalne kaugus kahe punkti vahel, on tekkiv vahe kahe sirge vahel piisavalt väike, et ei teki probleeme kolmanda punktiga. Ülesande teksti järgi siis peab olema $k > 20000$. Näiteks sirged

$$ax + by + c = 0 \quad ja \quad 30000ax + 30000by + 30000c + 1 = 0 \tag{1}$$

on paralleelsed ning nende vahel on kindlasti piisavalt väike vahe, et ükski kolmas punkt ei mahuks nende vahele ülesande tingimusi arvestades.

Aga kuidas leida, kas punkt on sirgest vasakul või paremal? Seda on tähtis teada mitmes lahenduse osas. Selle lahendamiseks saame kasutada samuti sirge võrrandit, pannes testitava punkti koordinaadid selle valemisse. Kui paremale poole võrdusmärki tuleb 0 asemel (nagu sirge peal asuva punkti puhul) negatiivne arv, on punkt ühel pool sirget, kui positiivne siis teisel pool. Kumb on vasak ja kumb on parem, on lahendaja enda valida, aga tegelikult tähtsust ei oma, sest enamasti tegeleme kahe punkti omavahel võrdlemisega. Kuna see on lahenduses palju kasutatav trikk, tasub see panna eraldi funktsiooni (`leftRight`).

```
def leftRight(point, line):
    value = point.x * line.a + point.y * line.b + line.c
    if value == 0:
```

```
        return 0 # täpselt sirgel
    if value < 0:
        return -1 # ühel pool sirget
    if value > 0:
        return +1 # teisel pool sirget
```

Kandidaatsirgete seast õige kolmiku leidmiseks tuleb üle kontrollida kõik kolmikud. Konkreetse kolmiku kontrollimiseks tuleb käia läbi kõik punktide paarid ja leida, kas mõni sirge eraldab neid. Seda teeb funktsioon `checkLines`. Mõned piirjuhtude kontrollid ja muud detailid välja jättes on üks võimalik lahendus järgmine:

```
def checkLines(lines, pts):
    # kontrollime kõiki punktide paare
    for pt1 in pts:
        for pt2 in pts:
            # teeme võrdluse iga sirgega
            for line in lines:
                if leftRight(pt1, line) != leftRight(pt2, line):
                    break # eraldav sirge leitud
            else:
                return False # eraldavat sirget ei leidunud
    return True

pts = [] # ülesande sisend
candidates = [] # siia lisame kandidaatsirged
for pt1 in pts:
    for pt2 in pts:
        line = lineThrough(pt1, pt2) #funktsioon, mis loob sirge kahest punktist
        # lisame sirged mõlemale poole punktidest
        candidates.append({a:line.a*30000, b:line.b*30000, c:line.c*30000+1})
        candidates.append({a:line.a*30000, b:line.b*30000, c:line.c*30000-1})

# käime käbi kõik kolmikud
found = False # kas leidsime vastuse?
for c1 in candidates:
    for c2 in candidates:
        for c3 in candidates:
            found = checkLines([c1, c2, c3], pts) or found
print(found)
```

5. Ristmike remont (`remont`)

3 sek / 7 sek 100 punkti

Idee, teostus ja lahenduse selgitus: Tähvend Uustalu

Antud N tipuga graaf ja Q päringut:

- Antud u; blokeeri tipp u.
- Antud u; tipp u ei ole enam blokeeritud.
- Antud u ja v; leia lühim teekond $u \rightsquigarrow v$, mis ei läbi vahepeatusena ühtegi blokeeritud tippu.

Iga tipp blokeeritakse ülimalt ühe korra. $N \leqslant 400$, $Q \leqslant 10^6$.

See on võistluse esimene ülesanne, kus olulist rolli mängib ajalimiit. Raskeid ülesandeid lahendades tuleb tähelepanu pöörata sisendi piirangutele. Näiteks proovisid paljud osalejad lahendada ülesannet, jooksutades iga päringu kohta eraldi Dijkstra algoritmi. Dijkstra algoritmi keerukus on $O((N+M)\log N)$, mis tiheda graafi korral on $O(N^2 \log N)$. Kui iga päringu kohta eraldi Dijkstra algoritmi kasutada, tuleb keerukuseks $O(QN^2 \log N)$.

Kasulik rusikareegel on, et see mahub paarisekundilisse ajalimiiti siis, kui $QN^2 \log N \leqslant 10^8$. Antud ülesandes võib aga olla, et $Q = 10^6$ ja $N = 350$, ja sel juhul

$$QN^2 \log N \approx 10^{12}.$$

Isegi esimeses alamülesandes, kus küll $N \leqslant 60$, võib Q endiselt olla miljon; kui võtta $N = 60$ ja $Q = 10^6$

$$QN^2 \log N \approx 2{,}1 \cdot 10^{10}.$$

Seega on iga päringu jaoks Dijkstra algoritmi eraldi jooksutamine isegi esimeses alamülesandes umbes kakssada korda liiga aeglane. Teatud võtetega saaks Dijkstra algoritmi kirjutada kaks või neli korda kiiremaks, aga Dijkstra 100 korda kiiremaks tuunimine oleks tõsine vägitegu.

Lahendust optimeerides tuleb aru saada, mis on tegelikult pudelikael. Et igas alamülesandes võib Q olla ülimalt miljon (ja enamus testides on Q täpselt miljon!), ei tohiks ühe 3. tüüpi päringu töötlemine võtta rohkem kui $O(\log N)$ või äärmisel juhul $O(\log^2 N)$ aega. Toimiv lahendus peab kuidagi arvesse võtma fakti, et päringud ei ole sõltumatud, vaid kõik toimub samal graafil.

Niisugune keerukuse arvutamine on ülesande lahendamise osa. Algoritmi välja mõeldes tasub kohe mõelda ka tema keerukusele ja jälgida, kas ja milliseid alamülesandeid selline algoritm lahendaks.

Esimene alamülesanne ($N \leqslant 60$)

Esimese alamülesande jaoks paneme esiteks tähele, et kuigi Q võib olla väga suur, saab graafis olla ainult $2N$ sisulist muudatust (iga tipp läheb remonti ülimalt ühe korra). Arvutame kõikide tippude vahelised kaugused üle pärast iga muudatust; päringu korral vaatame vastuse lihtsalt tabelist järgi. Kui iga tipupaari jaoks jooksutada Dijkstra algoritmi eraldi, võtaks see $O(Q + N^5 \log N)$ aega ($O(N)$ muudatust, $O(N^2)$ tipupaari ja iga Dijkstra võtab $O(N^2 \log N)$ aega). Kui $Q = 10^6$ ja $N = 60$, siis

$$Q + N^5 \log N \approx 4{,}5 \cdot 10^9,$$

mis on endiselt liiga aeglane.

Et lahendust edasi optimeerida, tuletame esiteks meelde Floyd-Warshalli algoritmi lühimate teede leidmiseks. See leiab $O(N^3)$ ajas iga tippude paari vahelise kauguse.

```
// alguses:
// dist[i][i] = 0
// dist[i][j] = serva ij kaal (või lõpmatus, kui serva pole)

for (int k = 1; k <= n; k++) {
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
    }
  }
}
```

Mõistmaks, kuidas seda algoritmi muuta nii, et see arvestaks blokeeritud tippudega, uurime välja, kuidas see töötab. Tuleb välja, et:

- pärast esimest iteratsiooni on `dist[i][j]` lühim tee $i \rightsquigarrow j$, mis kasutab vahepeatusena ainult tippu 1;
- pärast teist iteratsiooni on `dist[i][j]` lühim tee $i \rightsquigarrow j$, mis kasutab vahepeatusena ainult tippe 1 ja 2;
- pärast kolmandat iteratsiooni on `dist[i][j]` lühim tee $i \rightsquigarrow j$, mis kasutab vahepeatusena ainult tippe 1, 2 ja 3;
- jne.

See tähendab, et algoritmi kahest sisemiset tsüklist võib mõelda kui algoritmist, mis lisab tipu k lubatud vahepeatuste hulka. Tippe võib loomulikult lisada lubatud vahepeatuste hulka suvalises järjekorras. Esimese alamülesande lahendamiseks piisab niisiis sellest, kui jätta läbi käimata need k-d, mis vastavad blokeeritud tippudele:

```
// alguses:
// dist[i][i] = 0
// dist[i][j] = serva ij kaal (või lõpmatus, kui serva pole)

for (int k = 1; k <= n; k++) {
  if (!blocked[k]) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
      }
    }
  }
}
```

Nüüd on keerukus $O(Q + N^4)$ ($O(N)$ muudatust ja $O(N^3)$ ajas Floyd-Warshall). Kui võtta $N = 60$ ja $Q = 10^6$, siis

$$Q + N^4 \approx 1{,}3 \cdot 10^7 \leqslant 10^8,$$

mis on lõpuks ometi piisavalt kiire.

Kolmas alamülesanne

Sama tähelepanek Floyd-Warshalli algoritmi toimimise kohta lahendab suhteliselt otse ka kolmanda alamülesande (kus alguses ei tohi vahepeatusena kasutada ühtegi tippu, ja tippe hakatakse järjest lubatud vahepeatuste nimekirja lisama). Et algoritmi kaks sisemist tsüklit lisavad tipu k lubatud vahepeatuste nimekirja, siis kutsume neid lihtsalt iga "`+ k`" tüüpi päringu korral välja.

```
struct DistanceMatrix {
  vector<vector<ll>> dist;

  void unblock (int u) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        dist[i][j] = min(dist[i][j], dist[i][u] + dist[u][j]);
      }
    }
  }
};

// ... main-funktsioonis
DistanceMatrix dm;
// TODO: loe sisend, initsialiseeri maatriks...

for (auto query : queries) {
  if (query.type == '-') {
    // selles alamülesandes on need päringud ainult sisendi alguses
    // ja need blokeerivad ära kõik tipud. kuna meie lahendus eeldab,
    // et alguses kõik tipud blokeeritud on, siis lihtsalt ignoreerime
    // ja ei tee midagi
  } else if (query.type == '+') {
    // lisame tipu u lubatud vahepeatuste nimekirja
    dm.unblock(u);
  } else if (query.type == '?') {
    // leiame lühima teekonna u--v
    cout << dm.dist[query.u][query.v] << '\n';
  }
}
```

Nii on keerukus $O(Q+N^3)$ ($O(N)$ muudatust ja iga muudatuse korral $O(N^2)$ ajas kaks tsüklit). Võttes $Q = 10^6$ ja $N = 350$ saame, et

$$Q + N^3 \approx 4{,}3 \cdot 10^7 \leqslant 10^8,$$

mis on piisavalt kiire.

Täislahendus

Ülesande maksimumpunktidele lahendamiseks on aga vaja ka käsitleda olukordi, kus suvalistel hetkedel eemaldatakse tippe lubatud vahepeatuste nimekirjast. Kui säilitame kaugusmaatriksi kõik eelmised versioonid, võime küll eemaldada viimati lisatud lubatud vahepeatuse:

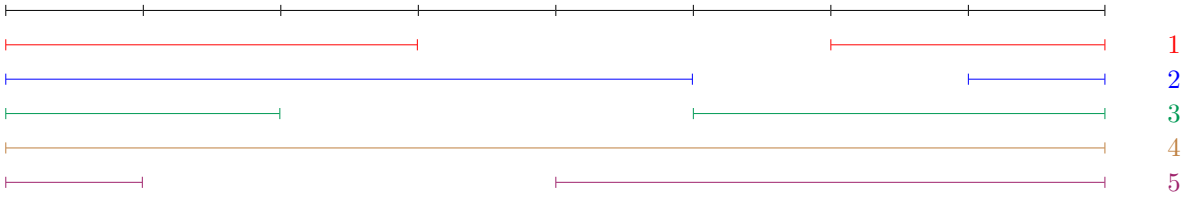
```
struct DistanceMatrix {
  vector<vector<vector<ll>>> dist_stk;

  void unblock (int u) {
    vector<vector<ll>> dist = dist_stk.back();

    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        dist[i][j] = min(dist[i][j], dist[i][u] + dist[u][j]);
      }
    }

    dist_stk.push_back(dist);
  }
```

```
  void block (int u) {
    // toimib AINULT EELDUSEL, et u oli viimati lisatud lubatud vahepeatus
    dist_stk.pop_back();
  }
};
```

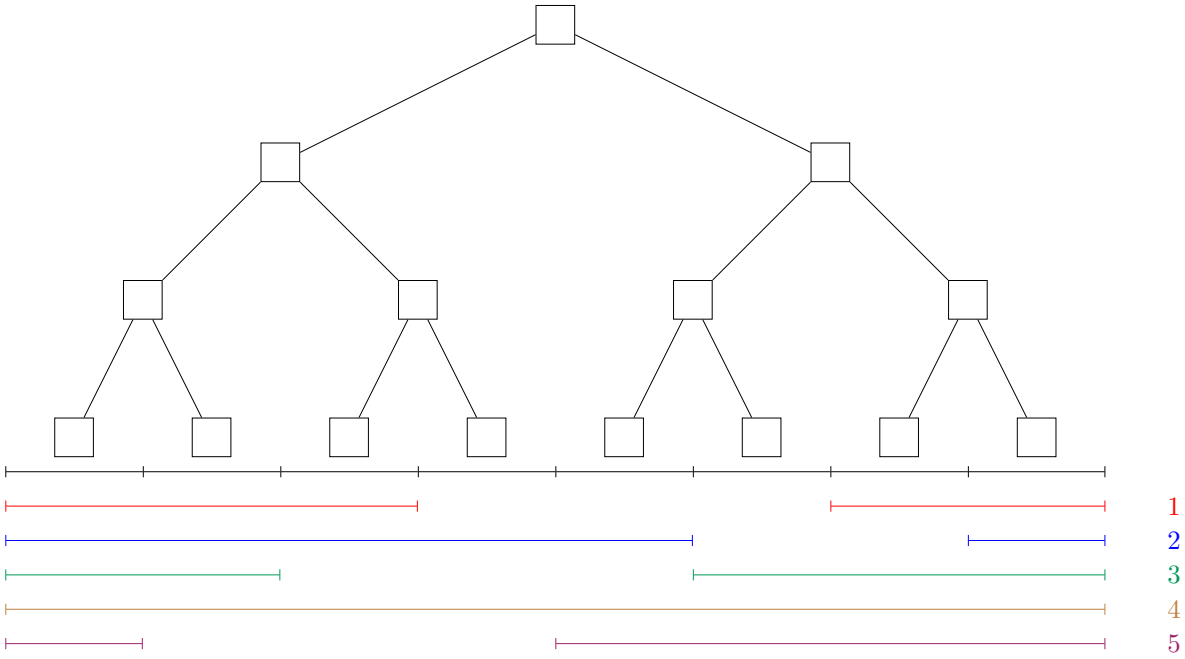
Sellest aga mõistagi ei piisa, sest blokeeritav tipp ei pruugi sugugi olla see, mis viimati lisatud.

Täislahenduseni jõudmiseks kasutame ära fakti, et programmile antakse kõik päringud korraga ette. Ülesande tekstis on lubatud, et ühtegi tippu ei blokeerita rohkem kui ühe korra, seega on lubatud vahepeatuste nimekirjas ülimalt $2N$ sisulist muudatust ja päringuid võidakse küsida $2N + 1$ "ajahetkel".

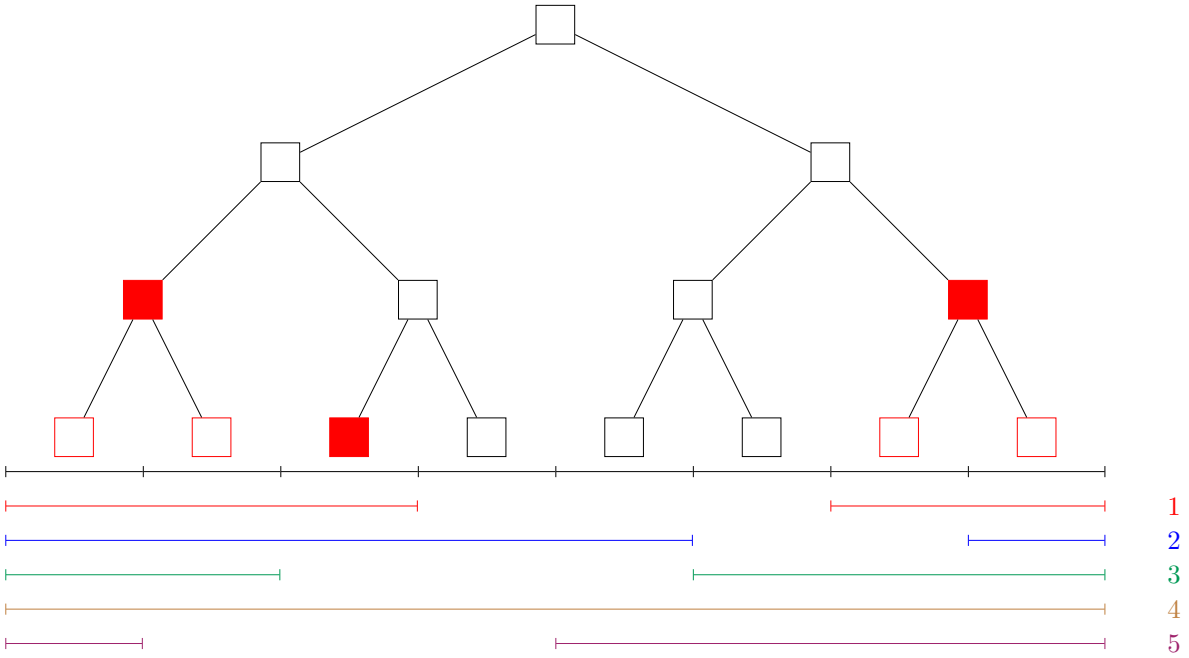
Kujutame ajajoonel iga tipu kohta hetked, mil see tipp on lubatud vahepeatus. Ülesande tingimustest tuleneb, et need hetked moodustavad üks või kaks lõiku.



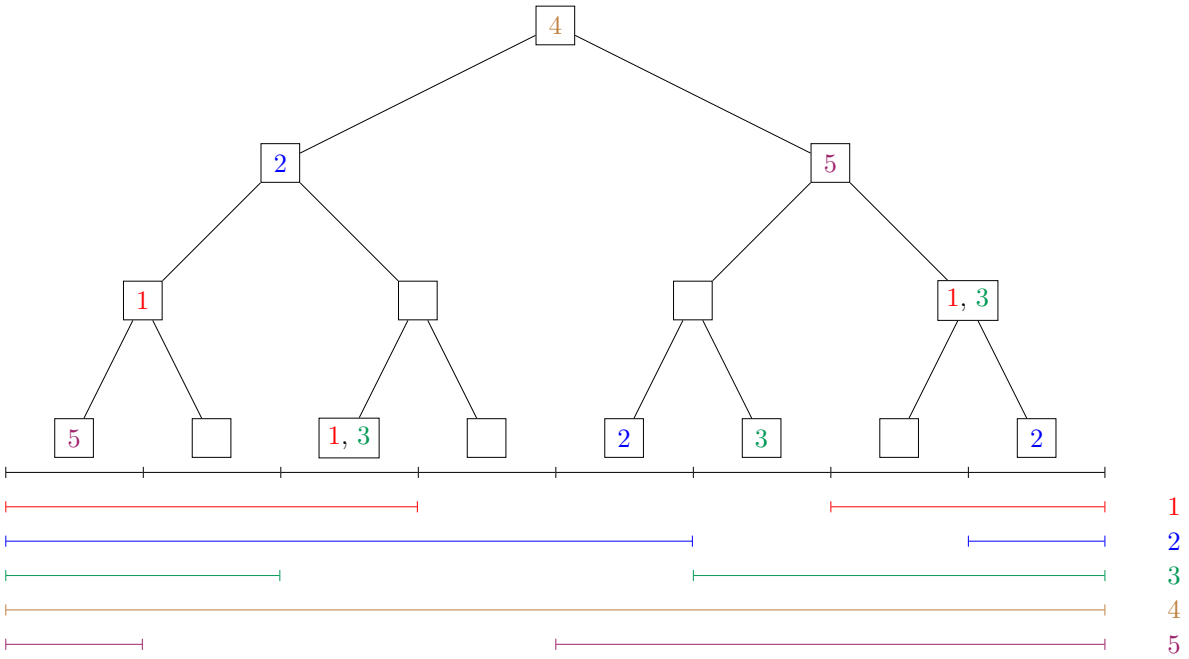
Nüüd ehitame tasakaalus kahendpuu, mille lehed vastavad ajahetkedele:



Iga lõigu kohta saame leida $O(\log N)$ tippu, mille alampuudes olevad lehed katavad täpselt lõiku kuuluvad ajahetked. Näiteks alloleval joonisel on kujutatud need tipud algse graafi tipu 1 lõikude jaoks:



Leiame need tipud iga lõigu jaoks ja paneme neisse kirja algse graafi tipu, millele antud lõik vastab.



Läbime selle puu sügavuti, hoides kogu aeg meeles kaugusmaatriksit. Puu tippu sisenedes lisame lubatud vahepeatuste nimekirja tipus kirjas olevad graafi tipud; tipust lahkudes eemaldame need (mis on võimalik, sest need on "viimasena" lisatud!). Puu lehtedesse jõudes vastame antud ajahetkel küsitud päringud.

Mis on selle algoritmi keerukus? Igale graafi tipule vastab ülimalt 2 lõiku; iga lõigu katmiseks on vaja $O(\log N)$ puu tippu. Kokku kirjutame puu tippudesse $O(N \log N)$ märget. Puud läbides võtab iga märkme töötlemine $O(N^2)$ aega, seega kokku kulub märgete töötlemiseks $O(N^3 \log N)$ aega. Puul endal on $O(N)$ tippu, lisaks tuleb vastata Q päringule. Kokku on ajaline keeru-

kus $O(N^3 \log N + Q)$.

Niisuguse puu ehitamine on üldisem meetod, mida võib kasutada ka teistes ülesannetes. Üldiselt, kui meil on mingi andmestruktuur, mis haldab mingit hulka ja toetab amortiseerimata keerukusega järgmisi operatsioone:

- lisa andmestruktuuri element u;
- eemalda andmestruktuurist viimati lisatud element,

kusjuures elementide lisamise järjekord ei ole oluline, siis on võimalik lahendada ülesanne, kus on vaja ka elemente kustutada. Keerukus korrutub sel juhul teguriga $O(\log K)$, kus K on lisamis- ja kustutamispäringute arv. Eelduseks on aga ka see, et kõik päringud on ette teada.

Näiteks on samal meetodil võimalik keerukusega $O(Q \log N \log Q)$ lahendada *dünaamilise sidususe ülesanne*: antud on graaf ja päringud:

- antud u ja v; lisa graafi serv uv;
- antud u ja v; eemalda graafist serv uv;
- antud u ja v; kontrolli, kas u ja v on samas sidususkomponendis.

6. Peitkanal (`kanal`)

1 sek / 1 sek 100 punkti

Idee, teostus ja lahenduse selgitus: Heno Ivanov

> Antud N-elemendiline korduvate elementidega hulk (multihulk) ja $v \in \{0, 1, \ldots, M-1\}$. Kodeerida v multihulga permutatsioonina, dekodeerida v permutatsiooni minimaalse prefixi järgi.
>
> Alamülesanded: korduvad elemendid puuduvad, hulk on $\{1, 2, \ldots, N\}$, $M = 2$

Lahenduse idee

Alustame lahendamist üldjuhust. Kõige loomulikum viis selles ülesandes andmeid kodeerida on (korduvate elementidega) permutatsioonide leksikograafiline järjestus ning *rank* ja *unrank* funktsioonid, mis vastavalt leiavad permutatsiooni järjenubri või genereerivad antud järjenumbriga permutatsiooni. Nii saaks ülesannet lahendada, kuid R-päringute arv oleks väga suur.

Multihulga permutatsioonide arv on $P = \frac{(n_1+n_2+\cdots+n_k)!}{n_1!n_2!\cdots n_k!}$, kus n_i erinevate väärtuse esinemiste arvud. On vaja, et igale v väärtusele vastaks erinev permutatsioon, seega peab $P \geqslant M$.

Kodeerija saab valida minimaalse hulga elemente, mille puhul $P \geqslant M$. Kuna elementide kordsus vähendab permutatsioonide arvu, lisame algul igast erinevast väärtusest ühe eksemplari, seejärel nendest, mida oli 2 või rohkem järgmise, jne. Iga elemendi lisamisel kontrollime permutatsioonide arvu (seda võib soovi korral ka jooksvalt arvutada).

Kuidas saab dekodeerija teada, millised elemendid kodeerija valis? Kuna M väärtus on teada, siis saame iga R-päringu järel arvutada teadaoleva prefiksi jaoks P. Kui $P \geqslant M$, dekodeerime vastuse. Kaval on seejärel meelde jätta kasutatud elementide arv ja nende väärtused — järgmise väärtuse dekodeerimisel saame nii ühe päringu kokku hoida (eelviimase elemendi lugemise järel teame alati järgmist).

Kodeerimise efektiivsust on võimalik täiendavalt parandada juhu $P > M$ korral. See on tõenäoline, kuna elementide lisamisel permutatsioonide arv kasvab "hüpetega".

- Teisendame kodeerivad väärtused $\{0, 1, \ldots, M-1\}$ vahemikku $\{0, 1, \ldots, P-1\}$.
- Esimese väärtuse dekodeerimine toimub nagu varem (lisandub ainult v õigesse vahemikku tagasi teisendamine).
- Järgmistel dekodeerimistel arvutame seni teadaoleva prefiksi jaoks minimaalse ja maksimaalse võmaliku (järgnevad väärtused on vastavalt kasvavas või kahanevas järjekorras) permutatsiooni numbri — ja kui sellesse vahemikku jääb ainult üks kodeeritav väärtus, saame anda vastuse.

Alamülesanne: $M = 2$

Võime valida 2 erinevat väärust ning kodeerida näiteks $0 = (\text{suur}, \text{väike}, \ldots)$, $1 = (\text{väike}, \text{suur}, \ldots)$. See lahendus on antud alamülesandes samaväärne üldise lahendusga.

Alamülesanne: $\{1, 2, \ldots, N\}$

Sellele alamülesandele saab kirjutada spetsiifilisi erilahendusi, mis on oluliselt efektiivsemad kui üldjuht. Idee: v kodeerida sobiva pikkusega N-süsteemi arvuna.

Alamülesanne: korduvad elemendid puuduvad

Idee: sorteerime elemendid ja kasutame elemendi väärtuse asemel tema kohta suurusjärjekorras. Kodeerime väärtuse N-süsteemi arvuna. Esimesel dekodeerimisel peame lugema sisse kõik arvud, järgmistel ainult vajalikud, kuna teame juba vastavust.

7. Välgud (`valgud`)

100 punkti

Idee, teostus ja lahenduse selgitus: Marko Tsengov

> Antud tasandiliste kahendpuude juured, lehed maksimaalsel kõrgusel ja lehtede koguarv, samuti eri hinnangute kordajad. Puude esitusele tasandil on seatud lisapiirangud.
>
> Leida puude kogum (mets), mis minimeerib kaalutud hinnangute summa.

Ülejäänud seletuses kasutame valdavalt puude (kui andmestruktuuride) terminoloogiat, arvestades samas ikkagi välgunooltele ülesande tekstis seatud piiranguid. Puu struktuuri annab välgunooltele tingimus, et iga punkt (peale juure) asub *täpselt* ühest punktist ühe ühiku võrra madalamal ja vasakul / paremal. Samuti välistab tingimus, et eri välgunoole punktid peavad üksteisest vähemalt 2 ühiku kaugusel asuma, võimaluse, et kaks eri välgunoolt omaks ühiseid punkte.

Et selles ülesandes on väike arv teste, mis on kõik avalikud, on mõistlik vaadata iga alamülesannet (testi) eraldi.

Mitme alamülesande lahendamisel ilmneb, et ühe hinnangu optimeerimine muudab üht või mitut teist (arvestatavat) hinnangut kehvemaks, mistõttu tuleb leida kompromiss kahe hinnangu väärtuse vahel, mille määrab (osaliselt) hinnangute kordajate väärtuste suhe.

Alamülesanne 1

Selles alamülesandes on vaja konstrueerida ainult üks puu, mille kõikide lehtede asukohad on antud. Hinnanguid mõjutavad vaid valitud lõikude pikkused, täpsemini lõikude pikkuste ruutude summa ja sama pikkusega lõikude arvude ruutude summa.

Samuti on selles alamülesandes puu kõrguseks määratud 11 (koos lehtedega kõrgusel 0). See on piisavalt väike, et lubada kõikide juhtude läbivaatust, mille saab implementeerida rekursiivselt juurest alates kõiki seni sobivaid puid konstrueerides ning nendele hindamisprogrammi rakendades. Näidislahenduses on toodud funktsioon, mis implementeerib kõik hinnangud ning paljud tingimustele mittevastavad puu konstruktsioonid (mõned tingimused on jäetud puid genereeriva koodi tagada).

Uurime vastava jõulahenduse teostamist. Konstrueerime puu kõrgusest H kuni kõrguseni 1 (kõrgusel 0 on lehed juba määratud) Igal kõrgusel enne 1 saab puu igas tipus jätkuda ülimalt neljal moel:

1. lõppeda sel kõrgusel, tekitades lehe;
2. jätkuda ainult vasakule;
3. jätkuda ainult paremale; või
4. hargneda (jätkuda mõlemale poole).

Teame, et antud juhul on lehti kokku neli, mistõttu igal antud kõrgusel saab olla maksimaalselt neli tippu. Samuti välistab lehtede paigutus kõigil kõrgustel peale 0 võimaluse 1 ning teame, et hargnemine toimub kuni kolm korda.

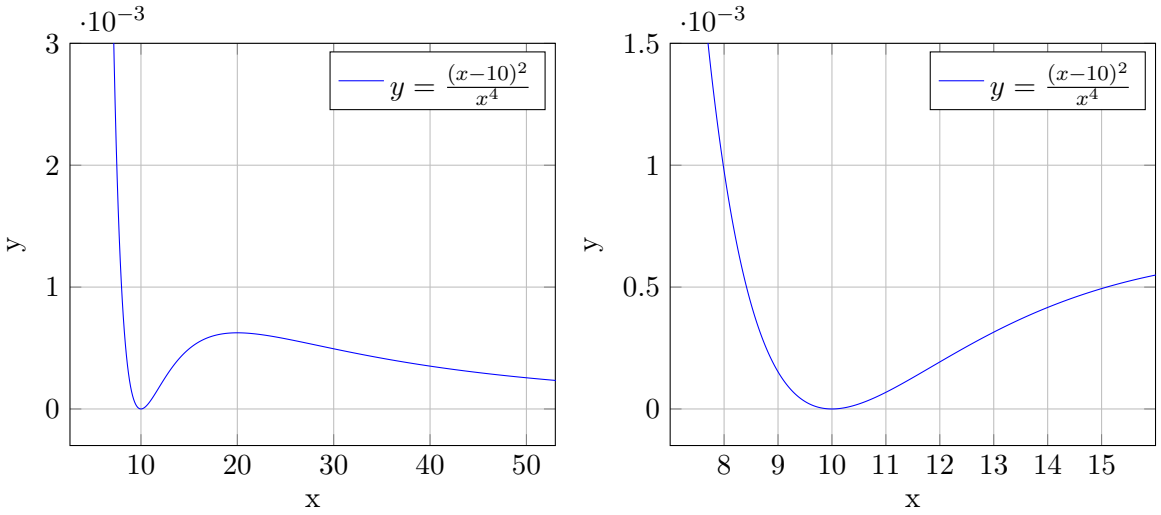
Nendest tähelepanekutest piisab piisavalt kiire kõigi juhtude läbivaatuse teostamiseks. Näidislahendus `brute.py` vaatab selles alamülesandes läbi paar miljonit sobiva struktuuriga puud, kuid suur osa ajast kulub mittesobivate konstruktsioonide peale.

Sellise lahenduse saab muuta märksa kiiremaks (mälukasutuse arvelt), konstrueerides pool puud juurest lehtede ning ülejäänud pool puud lehtedest juure poole (reeglid on sarnased), jättes ühel juhul meelde, millised tipud viimasel kõrgusel olid (ning kuidas vastavasse olukorda jõuda sai). Inglise keeles on vastav idee tuntud kui *meet-in-the-middle*.

Alamülesanne 2

Selles alamülesandes hinnatakse ainult n-ö õhus paiknevate lehtede paigutust, täpsemini viie sellise lehe kõrgust maapinnast ning nende omavahelist paiknemist.

Uurime hinnangus e toodud avaldise graafikut:



Antud graafikult on näha, et selle hinnangu minimeerimiseks tuleks iga kaks õhus olevat lehte hoida vähemalt 8 ühiku kaugusel teineteisest. Edasiseks hinnangu parandamiseks tuleks iga kahe lehe vahekaugus viia võimalikult lähedale 10 ühikule (seejuures pigem rohkem kui vähem) või siis väga suureks (> 30 ühikut). Selles väljendub ka ülesande tekstis antud umbmäärane kirjeldus, et vastavad lehed peaksid esinema gruppides.

Alamülesande lahendamiseks saab uurida lehtede paigutusi alas, kuhu puu tipud ulatuvad (juurest alates kolmnurk, samuti arvestada, et kolmnurga sees saab jõuda ainult punktideni, mille puhul koordinaatide $x + y$ paarsus vastab juure omale). Hinnangu d madalana hoidmiseks tuleb konstruktsioon luua võimalikult juure lähedal, kuid piisavalt madalal, et lehtede paigutamiseks oleks ruumi.

Alamülesanne 3

Selles alamülesandes on vaja konstrueerida kolm puud paljude lehtedega (maas), kuid puude kõrgus on väga väike. Seega saab rakendada sarnast kõikide juhtude läbivaatust kui alamülesandes 1. Samas on vaja arvestada nüüd ka hinnanguga c (mis mõjutab kogusummat vähe) ning vältida eri välgunoolte üksteisele lähedale sattumist.

Alamülesanne 4

See alamülesanne on väga sarnane alamülesandele 1, kuid puu kõrgus on ligikaudu kaks korda suurem ning lehti (sh õhus olevaid lehti) on rohkem. Seega on eeltoodud kõigi juhtude läbivaatus liiga aeglane ning vaja on mõnda uut ideed. Üks võimalus oleks kasutada alamülesandes 1 pakutud *meet-in-the-middle* ideed, samuti saab rakendada konstrueeritud puudele heuristikaid (nt piirata lühikeste ja pikkade lõikude hulka).

Alamülesanne 5

See alamülesanne on esimene suure kõrgusega test, mis välistab kõikide võimaluste läbivaatuse.

Sarnaselt alamülesandele 1 mõjutavad koguhinnangut ainult valitud lõikude pikkused. Samas on sel puul ainult üks leht, mis asub horisontaalselt juurega sarnases asukohas, kuid vertikaalselt väga kaugel (kõrgel). Seega tuleb konstrueerida lõikudest ahel, mis kahte antud punkti ühendab.

Ignoreerides horisontaalset asukohta saame selle alamülesande sõnastada ümber järgnevalt:

> Jaota lõik pikkusega H positiivsete täisarvuliste pikkustega tükkideks nii, et [kõikide tükkide pikkuste ruutude summa] · 2 ja [iga lõigupikkuse esinemise arvude ruutude summa] summa oleks minimaalne võimalik.

(Pikkuse arvestamiseks tuleb esimene summa kahega korrutada: $\sum \left(\ell\sqrt{2}\right)^2 = \sum 2\ell^2 = 2\sum \ell^2$)

Üksikute lõikude muutmine antud ülesande lahendusest ei mõjuta ilmselt hinnangut kuigi palju, samuti on ilmselt lahenduses palju eri pikkustega lõike, sealhulgas lõike pikkusega 1, 2, ... Kummagi ülesande hinnangut ei muuda lõikude ümberjärjestamine. Seega saame lihtsustatud ülesande lahendusest saadud lõigud ilmselt ümber järjestada nii, et ahela teise otsa horisontaalne asukoht kattub nõutud lehe asukohaga.

Lihtsustatud ülesandele saab leida hea lähendi näiteks ahnelt lõigupikkusi lisades, proovides lisada lõike pikkusega 1, 2, 3 jne kuni lõigupikkuseni, mida veel lisatud pole, misjärel valida lõigupikkus, mille korral lisanduv hinnangute väärtus (pikkusühiku kohta) on minimaalne. Vastav lähenemine on implementeeritud failis `ab_segments.py`

Alamülesanne 6

See alamülesanne sarnaneb väga eelmisele alamülesandele, kuid arvestada tuleb mitmete alguses ja lõpus lähestikku olevate välgunooltega ning hinnanguga c. Samas on hinnang c võrreldes teistega (kogemata) väga väike, misläbi võib seda praktiliselt ignoreerida. Et hinnang b uurib lõike iga välgunoole osas eraldi, saab praktikas lahendada alamülesande 5 ning rakendada saadud mustrit kõigile välgunooltele (hoides vastavad lõigud paralleelsed), mis garanteerib tingimuste täitmise.

Lisaülesanne: lahendada see alamülesanne juhul $k_c = 10^4$.

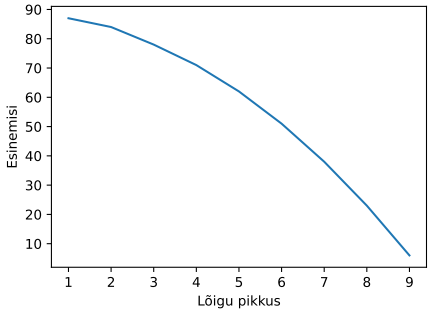
Alamülesanne 7

Selles alamülesandes on nõutud ühte väga palju hargnevat puud, mil on palju lehti õhus ja maas (antud kõrguse $H = 1000$ kohta). Hinnangud on aga sarnased kahe eelneva alamülesandega,

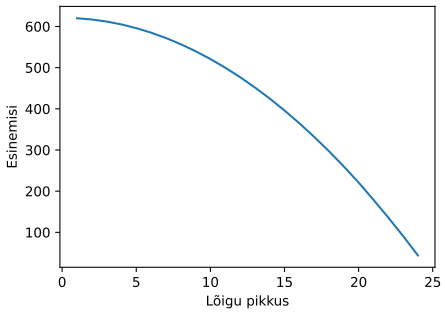
hoolides ainult lõikude pikkustest ja sama pikkusega lõikude arvudest.

Lahendusele võib läheneda sarnaselt alamülesande 5 ahnele lahendusele, kuid võrreldes uusi lõike hinnangu väärtusena n-ö toorelt, mitte pikkusühiku kohta. See lubab genereerida suvalise arvu lõike, millest puu koostada, minimeerides samas hinnangu väärtust. Samas ei ole sellistest lõigupikkustest taolise tugevalt hargneva puu koostamine ilmselt kuigi lihtne.

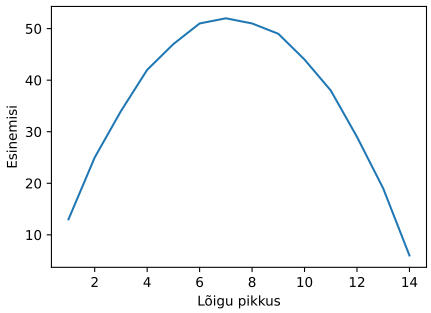
Samas on soovitav (ligikaudne) lõikude jaotus üsna hõlpsasti analüüsitav ja ennustatav, moodustades igal juhul ligikaudu paraboolse graafiku:



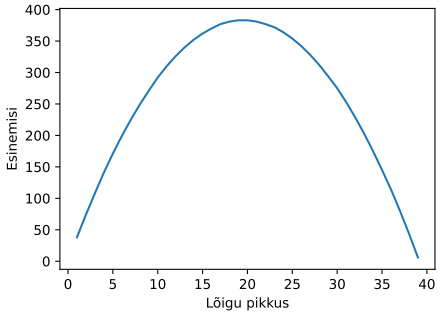
(a) Pikkuste jaotus 500 lõigu puhul



(b) Pikkuste jaotus 10000 lõigu puhul



(c) Pikkuste jaotus 500 lõigu puhul (arvestus pikkusühiku kohta)



(d) Pikkuste jaotus 10000 lõigu puhul (arvestus pikkusühiku kohta)

Seega saab üritada luua lahenduse, mis annab ligikaudu ülaltoodud jaotusega lõikude arvud, et konstrueerida puu juurest maapinnal olevate lehtedeni, misjärel saab lisada sobiva pikkusega lõikudena õhus olevad lehed.

Alamülesanne 8

See alamülesanne keskendub puhtalt õhus olevate lehtede paigutusele, olles praktiliselt laiendus alamülesandest 2. Optimeerimiseks on tarvilik leida hea paigutus nii väikeses ümbruses (20 ühikut, üks grupp) kui ka terve puu lõikes (gruppide vahel).

Alamülesanne 9

See alamülesanne on sarnane eelmisele alamülesandele, kuid kasutada on palju eri üsna lähestikku asetsevaid puid, kasutatavat kõrgust on pisut vähem ning osalt arvestatakse ka puude harude omavahelisi kauguseid (hinnang c). See annab võimaluse kõikide õhus olevate lehtede ühe puu otsa pookimise asemel jaotada lehti eri puude vahel, jättes need ilmselt kõrgemale.

Alamülesanne 10

Selles alamülesandes on kombineeritud kõik hinnangud, samuti on kasutatav ala väga suur ning ehitada tuleb palju eri lehtedega puid. Seega on juba mõningane väljakutse luua üldse tingimustele vastavat puude kogumit; sellise kogumi saavutamisel on mõttekas ilmselt olemasolevat lahendust pisut muuta, et soovitud hinnanguid parandada (näiteks lõike väiksemateks lõikudeks jaotada, lõikude pikkusi muuta, õhus olevate lehtede asukohta muuta).

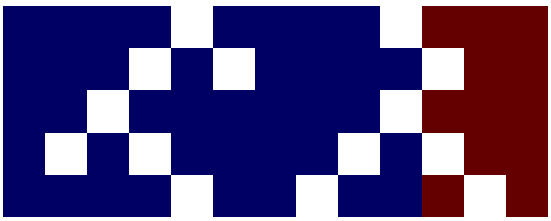
Märkus: alamülesandesse on pooljuhuslikult (testigeneraatori käitumisest tingituna) sattunud eripära, et üle poolte maas olevatest lehtedest tuleb ühendada vasakpoolseima juure külge.

Parimad esitatud lahendused

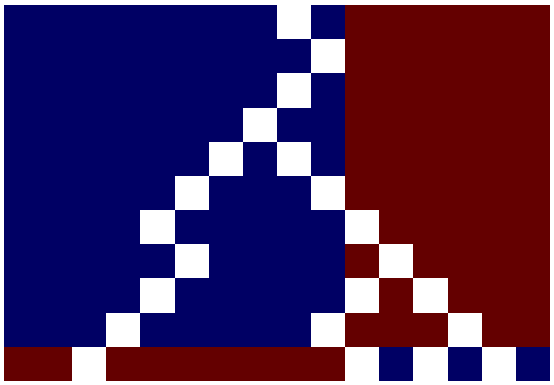
Vastavad lahendused on toodud ka kaustas `output_good`.

id	h_a	h_b	h_c	h_d	h_e	h	Autor
0	40	16	0.866	1.333	0.076	75.462	`brute.py`
1	102	39	0.974	0.000	0.000	160.500	`brute.py`
2	436	52.000	0.601	0.246	0.004	0.258	kl0989e
3	148	29	1.825	0.000	0.000	175.599	`brute.py`
4	246	120	1.700	0.750	0.000	306.000	Arian
5	222260	75561	0.000	0.000	0.000	297821.000	kl0989e
6	11190548	3699576	5431.847	0.000	0.000	14895555.847	kl0989e
7	528060	5220007	417.102	36.091	673.951	5748067.000	kl0989e
8	747612	17419304	759.443	7.474	144.116	218.854	kl0989e
9	4250576	1495021	571.414	8.846	127.365	626.786	kl0989e
10	809969368	874076	671.983	3.321	24.330	4065013.654	benmo

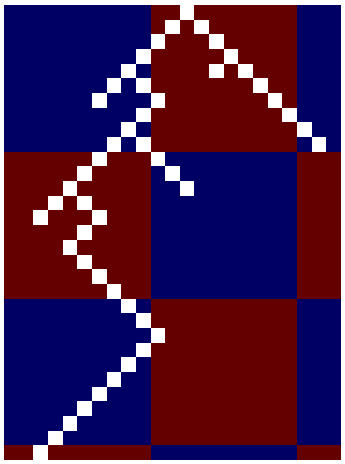
Järgnevatel piltidel on taustaks sini-punane ruudustik, kus iga ruudu küljepikkus on 10 ühikut. Välgu punkid on toodud valgete ruutudena.



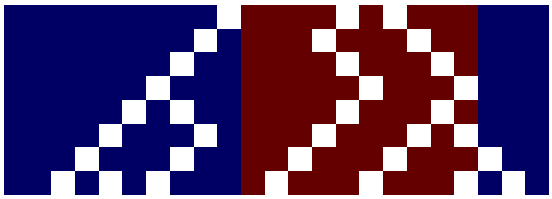
Parim lahendus alamülesandele 0 (`brute.py`).



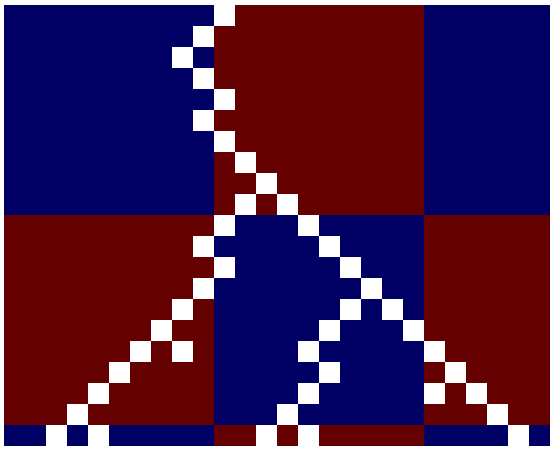
Parim lahendus alamülesandele 1 (`brute.py`).



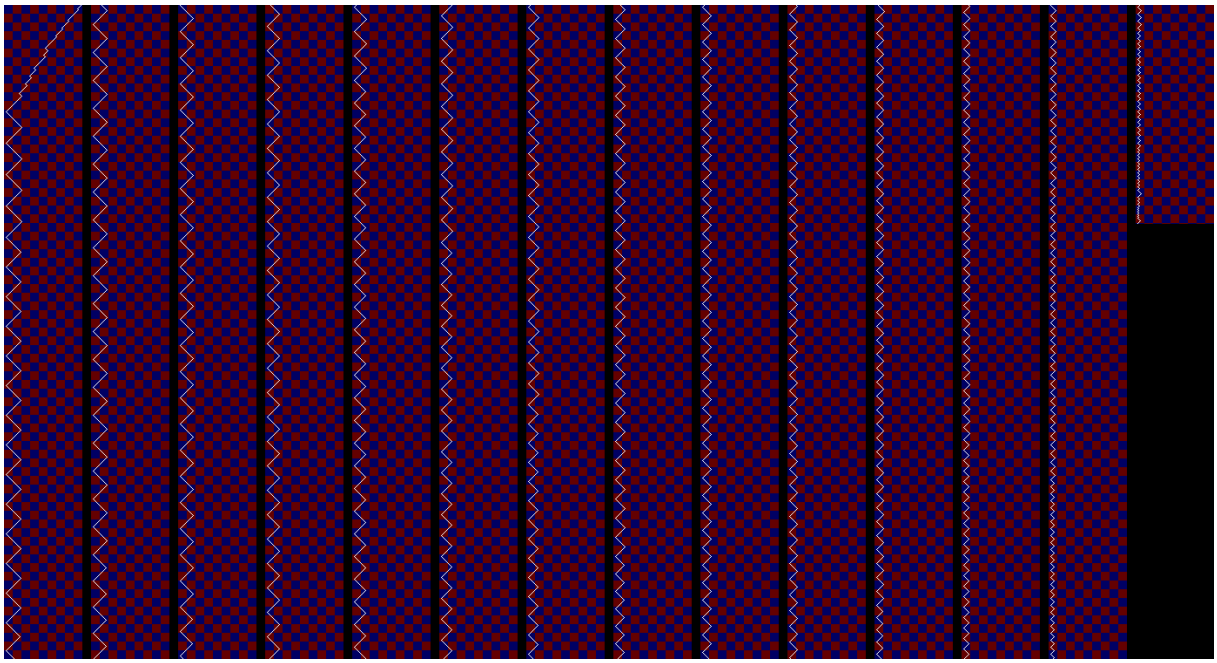
Parim lahendus alamülesandele 2 (kl0989e).



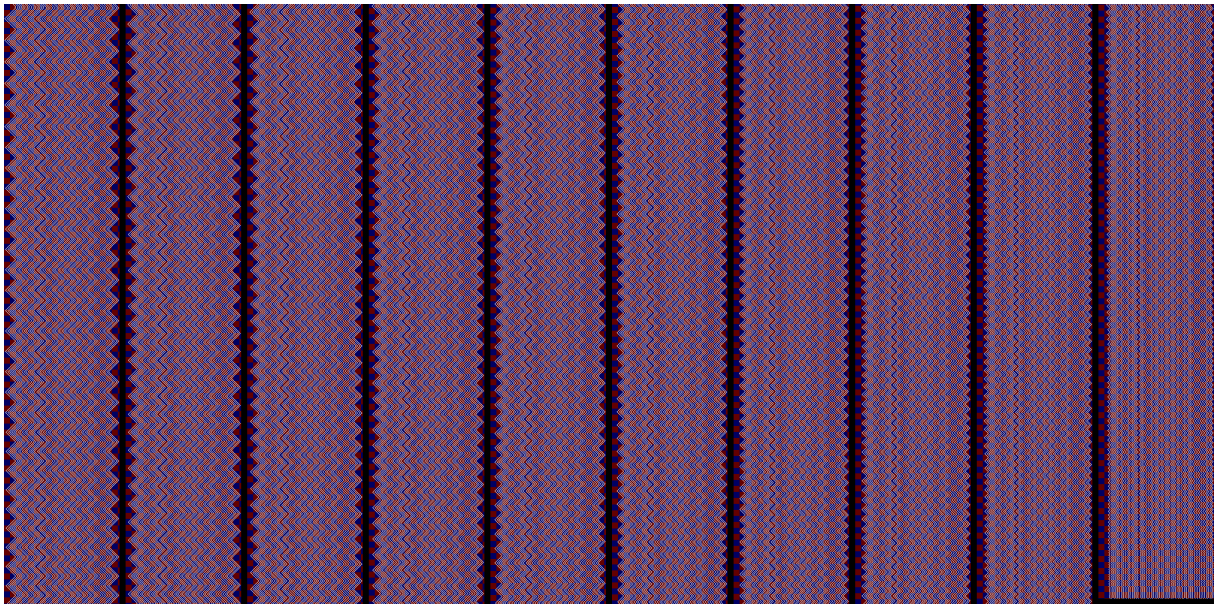
Parim lahendus alamülesandele 3 (`brute.py`).



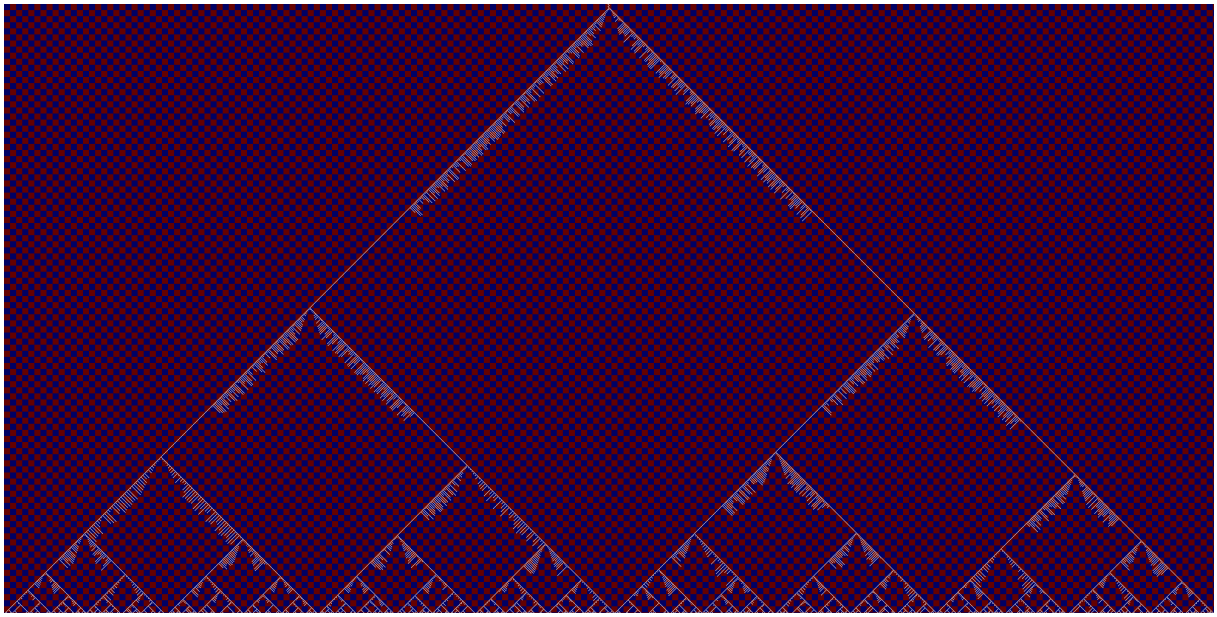
Parim lahendus alamülesandele 4 (Arian).



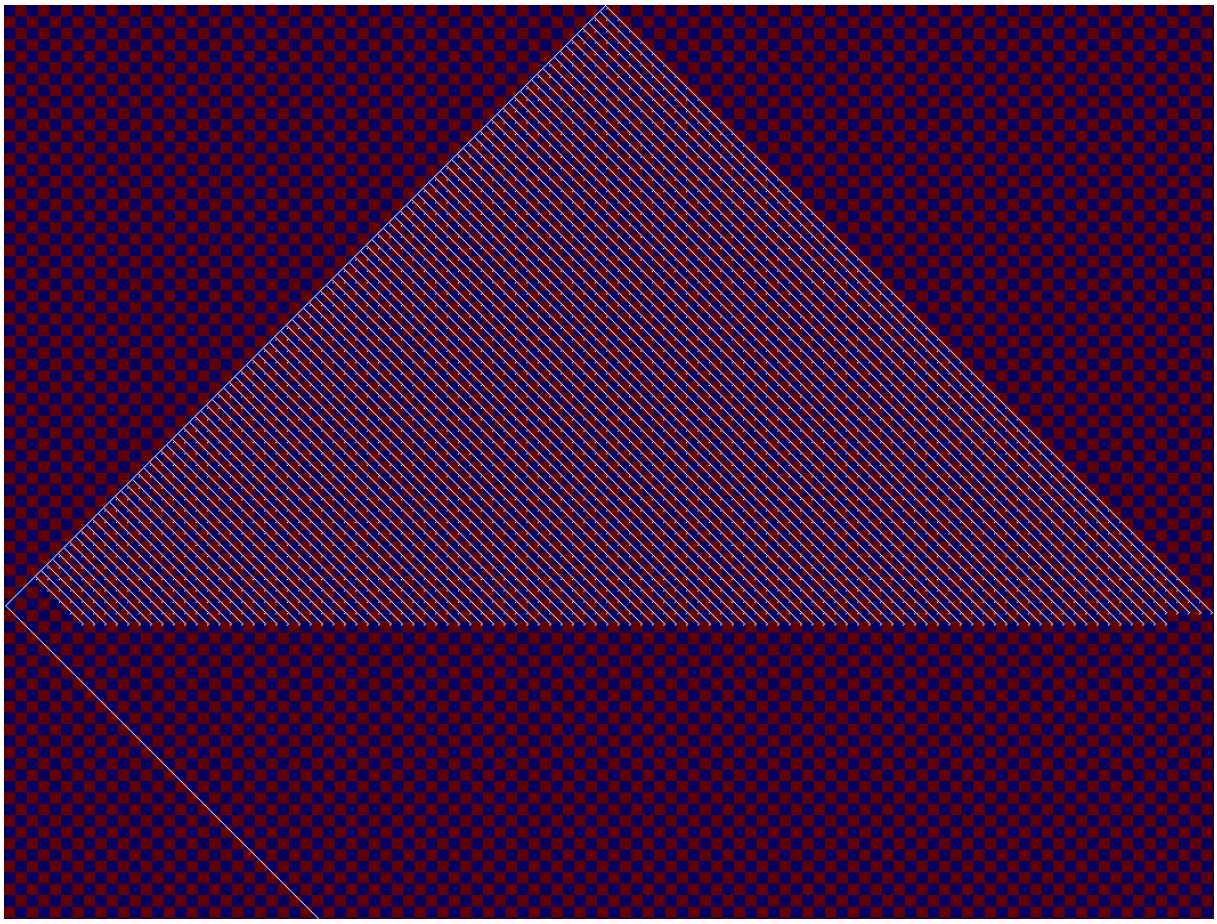
Parim lahendus alamülesandele 5 (kl0989e) lõigatud pikuti tükkideks.



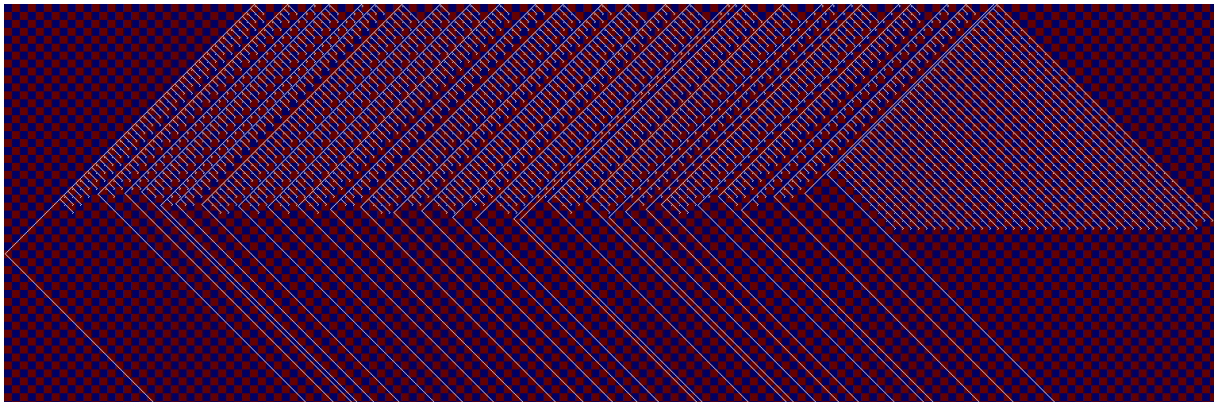
Parim lahendus alamülesandele 6 (kl0989e) lõigatud pikuti tükkideks.



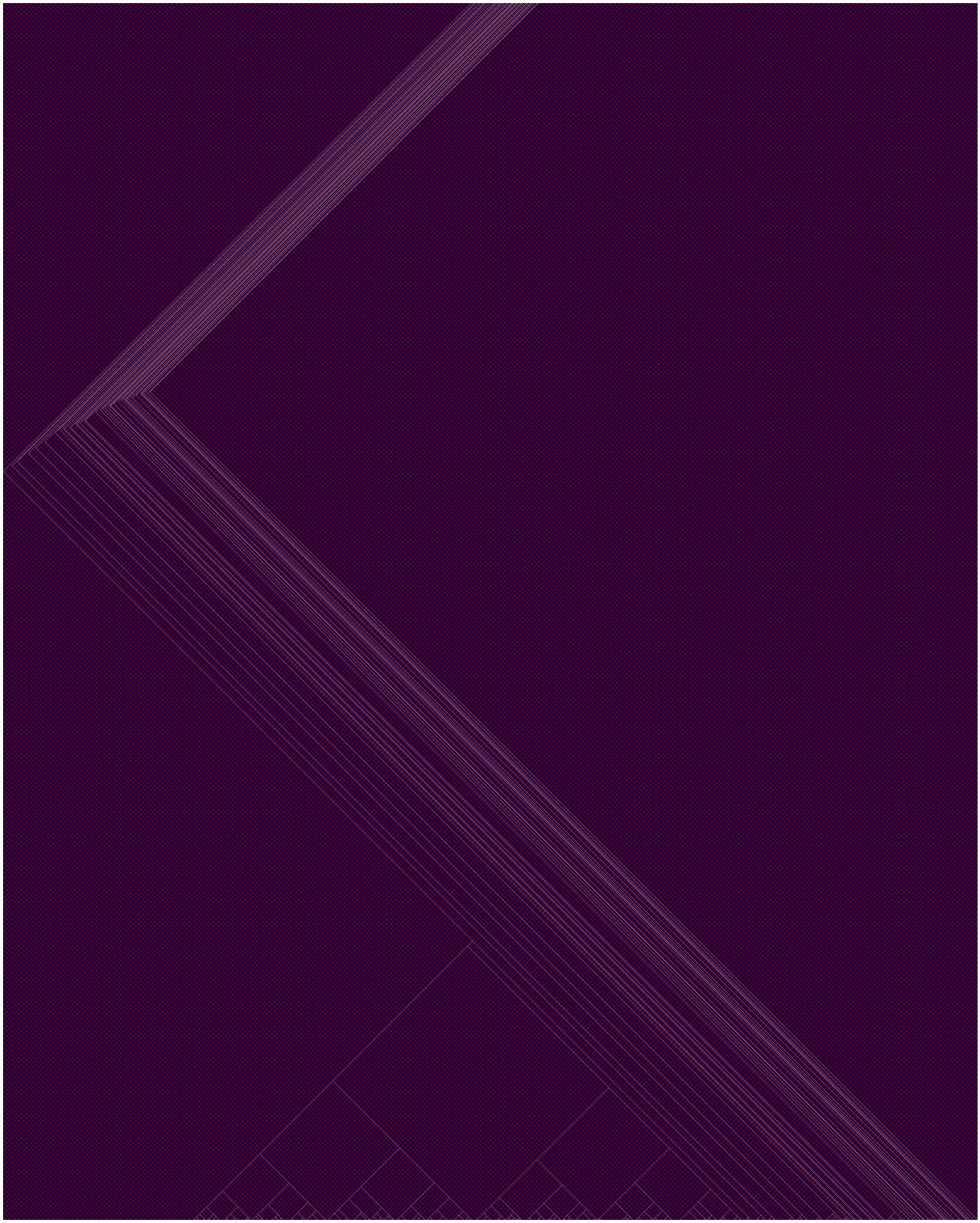
Parim lahendus alamülesandele 7 (kl0989e).



Parim lahendus alamülesandele 8 (kl0989e).



Parim lahendus alamülesandele 9 (kl0989e).



Parim lahendus alamülesandele 10 (benmo).