Sisukord

1. Buss nr 6 **2**

2. Viie pildiga robotilõks **3**

3. Neljakandilised saelauad **4**

4. Kahepäevane võistlus **5**

5. Kolmanda mõõtme saladus **9**

6. 7-segmendilised indikaatorid **13**

7. Ühetaoline tööaeg **16**

1. Buss nr 6 (buss)

1 sekund 20 punkti

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

Selle ülesande lahendamiseks peame kõik peatused läbi vaatama ja iga peatuse juures kontrollima, kas Juku sinna peatusse joostes bussi peale jõuaks või mitte. Jõudmise korral peame lisaks leidma, kui palju tal aega varuks jääb ja mittejõudmise korral, kui palju tal aega puudu jääb.

See kõik vajab aegade võrdlemist ja aegadega arvutamist. See oleks võimalik, kui aegu hoida tundide ja minutitena. Näiteks J minuti liitmine ajale $H : M$ võiks välja näha umbes nii:

```
M = M + J        # liidame J minutit
H = H + M // 60  # kanname täistunnid üle tunninäitu
M = M % 60       # ja võtame need minutinäidust maha
```

Kahe kellaaja lahutamine või võrdlemine oleks umbes sama tülikas. Hoopis lihtsam on kõik ajad teisendada minutiteks alates südaööst (1 tund = 60 min):

```
T = 60 * H + M  # esitame aja H:M minutites
```

Siis saame edaspidi teha kõik arvutused ja võrdlused mugavalt täisarvudega.

Et teada saada, kas Juku jõuab bussi peale või mitte, peame võrdlema aega, mis kulub Jukul peatusse jõudmiseks, ajagapraegusest hetkest bussi sellest peatusest väljumiseni. Jukul kuluva aja saame sisendist. Bussil kuluva aja leidmiseks lahutame bussi väljumisajast praeguse kellaaja.

Näitame, et ülesande lahendamiseks piisab leida maksimaalne vahe, mis tekib, kui lahutame bussil kuluvast ajast Jukul kuluva aja.

Kui Jukul ei kulu rohkem aega kui bussil, siis ta jõuab bussi peale. Seega lahutades bussil kuluvast ajast Jukul kuluva aja, on tulemus mittenegatiivne ning see ongi aeg, mis Jukul üle jääb ning mida tahame maksimeerida.

Kui Jukul kulub aga rohkem aega kui bussil, siis ta bussile ei jõua ning sama tehte vastus on negatiivne. Selle vastuse absoluutväärtus näitab, mitu minutit jääb Jukul bussi peale jõudmisest puudu.

Et iga mittenegatiivne arv on suurem igast negatiivsest arvust, siis saab suurim bussil kuluva aja ja Jukul kuluva aja vahe olla negatiivne ainult siis, kui Juku ei jõua bussi peale üheski peatuses. Kuna negatiivne arv on seda suurem, mida väiksem on selle absoluutväärtus, siis neist negatiivsetest arvudest maksimum vastab peatusele, kus Jukul jääb bussi peale jõudmisest puudu kõige vähem.

Selle idee elluviimiseks teeme kaks muutujat, millest üks peab meeles seni suurimat ajavahet ja teine seda peatust, kus selline vahe tekkis. Kuna bussil peatusesse minekuks kuluv aeg on mittenegatiivne ja Jukul saab kuluda maksimaalselt 10 000 minutit, ei saa see vahe kunagi olla väiksem kui −10 000. Seega võtame algväärtuseks midagi sellest veel väiksemat, näiteks −10 001.

Seejärel käime kõik peatused läbi ja kui leiame suurema ajavaru, siis uuendame muutujaid.

Sellised lahendused ongi toodud võistluse materjalide arhiivis alamkaustas `buss/solution` failides `sol.py` ja `sol.cpp`.

2. Viie pildiga robotilõks (`captcha`)

1 sekund 30 punkti

Idee: Heno Ivanov, teostus ja lahenduse selgitus: Targo Tennisberg

Lahendus koosneb kahest osast: väikevenna klõpsude läbisimuleerimine ja selle järel õige lahenduse leidmine.

Programmeerimisaja kokkuhoiu mõttes on hea kasutada ühtset protseduuri klõpsude simuleerimiseks. Vaja on hoida järjekorda märgitud piltidest, iga uus klõps kas eemaldab pildi järjekorrast või siis lisab selle järjekorra lõppu. Sellise protseduuri võib kirjutada ise või siis kasutada standardteegis olemasolet funktsionaalsust. Näiteks Pythoni loendil on juba olemas append ja remove meetodid, mis teevad just seda, mida vaja.

Kui väikevenna klõpsud on simuleeritud, tähistame lihtsuse mõttes pildid numbritega 1 kuni 5. Paneme tähele, et kui pilt 1 ei ole kohe järjekorra alguses, tuleb kõik eelnevad pildid sealt niikuinii eemaldada.

Seega, kui järjekorra alguses on juba pilt 1, on kõik korras. Vastasel korral eemaldame järjekorra algusest pildi ja lisame sellele vastava täiendava klõpsu vastusesse. Kui pilt 1 on leitud, hakkame uurima järjekorra teist positsiooni ning kordame sama protseduuri pildi 2 jaoks. Seejärel teeme sama pildi 3 jaoks jne.

Kui esialgne järjekord saab tühjaks, lisame kõik allesjäänud pildid õiges järjekorras vastusesse.

3. Neljakandilised saelauad (`vandra`)

1 sek / 3 sek 40 punkti

Idee, teostus ja lahenduse selgitus: Andres Alumets

Esimese 20 punkti saamiseks võime kirjutada lihtsa lahenduse. Kuna $N \leqslant 1000$, siis jääb ajalimiidi piiridesse lahendus, kus võrdleme iga lauda iga teise lauaga ja vaatame, kas need kattuvad. Kattumise kontrollimiseks on mitmeid võimalusi.

Üks sellistest on kontrollida, kas laudade servad lõikuvad. Seda saab teha nii, et võtta ühe laua horisontaalne serv ja teise vertikaalne serv. Seejärel vaatame, kas vertikaalse serva x-koordinaat on horisontaalse serva x-koordinaatide vahel ja horisontaalse serva y-koordinaat on vertikaalse serva y-koordinaatide vahel. Kui mõlemad tingimused kehtivad, siis servad lõikuvad ja lauad kattuvad. Peale selle on kattumiseks veel üks võimalus: kui üks laud asub tervenisti teise peal. Selle kontrollimiseks piisab, kui vaadata kas ühe laua kõigi nurkade koordinaadid on teise laua piiride vahel. Kui kumbki tingimus ei kehti, siis võib kindel olla, et lauad ei kattu. Kui mõni lauapaar kattus, siis kirjutame väljundisse '`JAH`', muidu '`EI`'.

Teine võimalus: lauad ei kattu, kui üks neist on tervenisti teisest paremal (kui esimese laua vasaku serva x-koordinaat on vähemalt sama suur kui teise laua parema serva x-koordinaat) või tervenisti teisest vasakul või tervenisti teisest kõrgemal või madalamal. Nii on tehtud failides `sol_naive.py` ja `sol_naive.cpp` toodud lahendustes.

Täislahenduse jaoks kasutame lõikude puud (ingl *segment tree*) ja skaneerivat joont (ingl *sweep line*). Alguses loeme sisse kõik lauad ning jätame meelde milliseid y-koordinaadi väärtuseid meil esines. Seejärel teeme lõikude puu, mille iga leht tähistab kahe järjestikuse y-koordinaadi väärtuse vahelist lõiku: esimene leht kahe kõige väiksema väärtuse vahelist lõiku, teine leht suuruselt teise ja kolmanda vahelist lõiku jne kuni viimane leht tähistab kahe suurima vahelist lõiku. Lihtsam on opereerida puuga, mille lehtede arv on 2 aste; selleks võivad ülejäävad lehed tähistada piirkonda, mis on suurem kui ükskõik milline sisendis olnud y-koordinaadi väärtus. Ülemised tipud puus tähistavad alluvate poolt tähistatud lõikude summat ja väärtusena hoiame seal alluvate maksimumi. Algselt on igal pool väärtus 0.

Edasi sorteerime lauad vasakpoolse ääre järgi mittekahanevalt ja hakkame neid järjest puusse lisama. Igaühe puhul liidame selle y-koordinaadi väärtuste lõigus olevatele tippudele 1 juurde ja jaotame laisalt allapoole. Lisaks hoiame meeles ka parema ääre järgi järjestatud laudade jada. Enne kui uue laua puusse lisame, peame eemaldama kõik need, mille parem äär on enne või samal kohal kui uue vasak. Eemaldamisel lahutame laua y-koordinaadi väärtuste lõiku kuuluvatest tippudest 1. Nii on meil alati iga y-koordinaadi väärtuste lõigu kohta teada, mitu lauda selles lõigus hetkel on. Kui kunagi on mõnes lõigus rohkem kui 1 laud, siis on kattumine ja võime väljastada '`JAH`'. Kui seda kunagi ei juhtunud, siis saame väljastada '`EI`'. Nii on tehtud failides `sol.py` ja `sol.cpp` toodud lahendustes, mille keerukus on $O(N \log N)$ ja mis saavad maksimumpunktid.

4. Kahepäevane võistlus (`voistlus`)

1 sek / 3 sek 60 punkti

Idee ja teostus: Olivia Tennisberg, lahenduse selgitus: Tähvend Uustalu

Alamülesanne $N \leqslant 10$

Esimese alamülesande lahendamise jaoks proovime läbi kõik variandid panna igale esimese päeva skoorile vastavusse mõni teise päeva skoor. Selleks on $N! = 1 \cdot 2 \cdot 3 \cdots N$ varianti, mis $N = 10$ puhul jääb kolme ja poole miljoni ringi. Olgu öeldud, et nii Pythonis kui C++-s on variantide läbi vaatamiseks mugavaid abifunktsioone. Pythonis on üks kasulik funktsioon `itertools.permutations`, C++-s on aga näiteks võimalik kasutada `std::next_permutation`. Näiteks Pythonis võiks lahendus olla midagi sellist.

```
import itertools

N = int(input())
A = list(map(int, input().split()))
B = list(map(int, input().split()))

# answer[i] olgu i-nda õpilase võiduvõimaluste arv, esialgu 0
answer = [0] * N

# itertools.permutations(B) tagastab listi B kõik permutatsioonid
# seejuures kordustega, näiteks itertools.permutations([2, 1, 2])
# sisaldab (1, 2, 2) kaks korda
# see on ka see, mida me praegu tahame
for perm in itertools.permutations(B):
    # leiame, mis oleks sellise vastavuse korral võitja skoor
    # seda saab kindlasti ka mingi one-lineriga, aga jätame
    # praegu lahenduse lihtsaks
    highest_score = 0
    for i in range(N):
        highest_score = max(highest_score, A[i] + perm[i])

    # nüüd lisame ühe kõikide osalejate vastusele, kes selle
    # skoori saavutasid
    for i in range(N):
        if A[i] + perm[i] == highest_score:
            answer[i] += 1

print(*answer, sep="\n")
```

Selline lahendus teenib 18 punkti. Suuremates testides aga ületab ajalimiidi. Mis on selle põhjus? Ülal märkisime, et vaja on läbi vaadata $N!$ varianti. Kui $N = 80$, siis $N! \approx 7 \cdot 10^{118}$; kui $N = 300$, siis juba $N! \approx 3 \cdot 10^{614}$.

Kui arvestada, et ühe variandi läbi vaatamiseks kulub 10 nanosekundit ehk 10^{-8} sekundit, siis 300! variandi läbi vaatamiseks kulub suurusjärgus 10^{599} aastat (mis on kõvasti rohkem, kui universumi eluiga mistahes hääbumise stsenaariumis).

Raskemates ülesannetes on see tüüpiline. Peaaegu kõikidele informaatikaülesannetele on võimalik *mingi* lahendus välja mõelda, näiteks kõikide variantide läbi vaatamise kaudu (kuigi mõnikord osutub ka sellise programmi kirjutamine üsna keeruliseks). Vajame lisaks toimivale lahendusele ka piisavalt kiiret lahendust. Mõnes ülesandes ongi see kiire lahenduse leidmine ülesande lahendamise lõviosa. Selle juurde käib oskus hinnata, kui kaua programm halvimal juhul aega võtab. Üldiselt ei maksa loota sellele, et testide seas seda kõige halvemat juhtu lihtsalt ei ole.

Kindlasti ei piisa siin teatud pisioptimeeringutest, nagu C++-s `i++` asemel `++i` kirjutamine. Meie praegune lahendus ei ole mõne millisekundi võrra liiga aeglane, vaid õige mitu suurusjärku liiga aeglane. Täislahenduseks on vaja hoopis teistsugust lähenemist.

Täislahendus

Sorteerime teise päeva tulemused kasvavas järjekorras. Edasi on täislahendust hea selgitada näite abil. Vaatleme järgnevat testi:

A	12	10	9	8	6	1
B	1	2	5	7	8	9

Uurime iga paari (i, j) kohta, mitu võimalust on õpilasel i võistlus võita, kui ta saab teisel päeval j-nda tulemuse.

```
N = int(input())
A = list(map(int, input().split()))
B = list(map(int, input().split()))
B.sort()
for i in range(N):
    for j in range(N):
        # TODO: mitu võimalust on i-ndal õpilasel võistlus võita
        # eeldusel, et ta saab teisel päeval j-nda tulemuse
```

Näiteks selgitame välja, mitu võimalust on võita õpilasel, kes sai esimesel päeval 8 punkti, eeldusel, et ta sai teisel päeval 7 punkti. See tähendab, et ükski ülejäänud õpilane ei tohi saada üle $8 + 7 = 15$ punkti.

Ülejäänud õpilased on siis:

A	12	10	9	6	1
B	1	2	5	8	9

Proovime välja arvutada, mitu võimalust on igale ülejäänud õpilasele valida vastav teise päeva tulemus. Alustame sellest, kellel oli esimesel päeval kõige kõrgem tulemus, sest tal on kõige vähem variante.

- Õpilane, kes sai esimesel päeval 12 punkti, pidi teisel päeval saama 1 või 2 punkti (kui ta oleks saanud juba 5 punkti, siis oleks tal kokku $12 + 5 = 17 > 15$ punkti). Seega 2 varianti.
- Õpilane, kes sai esimesel päeval 10 punkti, pidi teisel päeval saama 1, 2 või 5 punkti. Üks nendest valikutest on aga juba eelmisele õpilasele reserveeritud. Seega $3 - 1 = 2$ varianti.
- Õpilane, kes sai esimesel päeval 9 punkti, pidi teisel päeval saama 1, 2 või 5 punkti. Kaks nendest valikutest on eelmistele õpilastele reserveeritud. Kokku $3 - 2 = 1$ variant.
- Õpilane, kes sai esimesel päeval 6 punkti, pidi teisel päeval saama 1, 2, 5, 8 või 9 punkti, neist kolm on eelmistele reserveeritud. Kokku $5 - 3 = 2$ varianti.
- Viimasel õpilasel on analoogiliselt $5 - 4 = 1$ variant.

Kokku on sellel õpilasel $2 \cdot 2 \cdot 1 \cdot 2 \cdot 1 = 8$ võimalust võita eeldusel, et ta sai teisel päeval 7 punkti.

```
MOD = 10**9 + 7
N = int(input())
A = list(map(int, input().split()))
B = list(map(int, input().split()))
B.sort()
for i in range(N):
    total_ways = 0
```

```
    for j in range(N):
        # mitu võimalust on i-ndal õpilasel võistlus võita
        # eeldusel, et ta saab teisel päeval j-nda tulemuse
        # selleks peavad ülejäänud õpilased saama ülimalt A[i] + B[j] punkti
        ways = 1

        # mitu sobivat teise päeva skoori on?
        good_scores = 0

        # hakkame kahanevas järjekorras õpilasi läbi käima, nagu ülal näidatud
        for k in range(N):
            if i == k:
                # õpilane, kes pidi võitma
                # tema jätame siinkohal vahele
                continue

            while good_scores < N and A[k] + B[good_scores] <= A[i] + B[j]:
                good_scores += 1

            # õpilasel k on good_scores võimalikku teise päeva skoori
            possible_scores = good_scores

            # aga mõned neist on juba eelmistele reserveeritud
            if k < i:
                possible_scores -= k
            else:
                possible_scores -= k - 1

            # lisaks võib good_scores sisse olla arvestatud ka j-s
            # mis on samuti juba ära lubatud
            if j < good_scores:
                possible_scores -= 1

            ways *= possible_scores
            ways %= MOD

        total_ways += ways
        total_ways %= MOD
    print(total_ways)
```

Keerukus on $O(N^3)$.

Lisaväljakutse: Lahenda ülesanne keerukusega $O(N^2 \log N)$ (võib näiteks eeldada, et sisendi piirangutes on $N \leqslant 1000$).

Jäägiga jagamisest

Siia juurde on hea rääkida ka moodulitest. Selles ülesandes (ja üldse paljudes "võimaluste loendamise" ülesannetes) on palutud vastuse asemel väljastada jääk, mis tekib vastuse jagamisel arvuga $10^9 + 7$ (öeldakse ka: "vastus mooduli $10^9 + 7$ järgi").

Esiteks olgu öeldud, et kõikides endast lugu pidavates programmerimiskeeltes on selline jäägi leidmine sisseehitatud. Nii Pythonis kui C++-s leiab `a % b` jäägi, mis tekib arvu a jagamisel arvuga b.[1] Käsitsi mingit kirjalikku jagamist ei ole kindlasti vaja programmeerida!

[1] Selles ülesandes see küll suurt rolli mängida ei tohiks, aga negatiivsete arvudega tuleb kohati ettevaatlik olla: `-8 % 5` on Pythonis `2`, C++-s aga `-3`. Mõlemad on omamoodi loogilised.

Milleks aga sellist täiendavat sammu vaja on? Asi on selles, et vastus võib olla väga suur arv. Näiteks kui antud ülesandes $N = 300$ ja test on niisugune, et alati võidab üks ja sama õpilane, siis on vastus

$$N! = 1 \cdot 2 \cdot 3 \cdots N \approx 3 \cdot 10^{614}.$$

C++-s on tavalised täisarvud 32- või 64-bitised, mis tähendab, et nende väärtus saab olla vastavalt ülimalt 2 147 483 647 või 9 223 372 036 854 775 807. Need ülempiirid on palju väikesemad kui ülaltoodud 615-kohaline arv. Pythonis, kus tavaline täisarv saab olla kuitahes suur, on selles mõttes lihtsam, aga see lihtsus on omamoodi petlik: väga suurte arvudega arvutamine võib osutuda aeglaseks. Antud ülesandes ei ole see nii oluline, aga paljudes variantide loendamise ülesannetes ei ole vastused mitte mõnesaja-, vaid miljardikohalised.

Paneme tähele,[2] et

$$\begin{aligned}(a + b) \bmod c &= (a \bmod c + b \bmod c) \bmod c \\ (a - b) \bmod c &= (a \bmod c - b \bmod c) \bmod c \\ a \cdot b \bmod c &= [(a \bmod c) \cdot (b \bmod c)] \bmod c.\end{aligned}$$

Siin tähistab mod mooduli võtmise ehk jäägi leidmise tehet. See tähendab, et matemaatilises mõttes ei ole vahet, kas võtta moodul alles arvutuskäigu lõpus või pärast iga vahevastuse leidmist.

Ülesannetes, kus vastus tuleb väljastada mooduli järgi, ongi tavaliselt mõeldud, et lahenduses võetakse moodulit iga tehte järel, sest nii ei kasva ka vahetulemused liiga suureks.

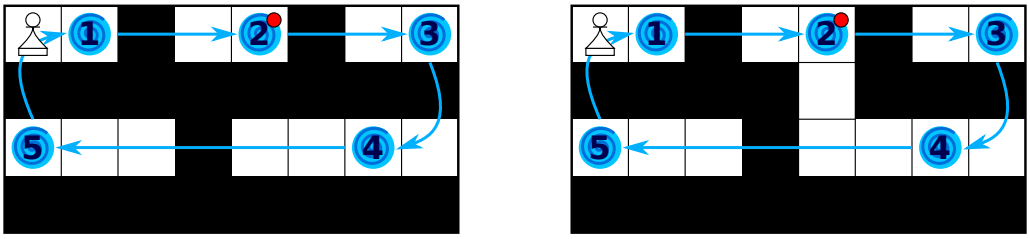
[2] Jagamisega on pisut keerulisem. Selles ülesandes ei ole see oluline, aga saab näidata, et kui p on algarv ning a jagub b-ga, siis $\frac{a}{b} \bmod p = (ab^{p-2}) \bmod p$.

5. Kolmanda mõõtme saladus (portaal)

1 sek / 3 sek 100 punkti

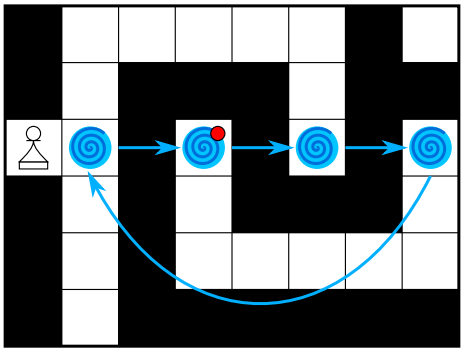
Idee, teostus ja lahenduse selgitus: Tähvend Uustalu

Mis eristab seda ülesannet tüüpilisest "labürindiülesandest", mida saab "tavalise" BFS või DFS abil lahendada? Kuna portaalid mõjutavad ka münte, siis võib münt "eest ära liikuda". Vaatleme näiteks alloleval joonisel kujutatud olukorda. Kui läheme portaali 1 juurde ja tõmbame kangist, siis liigume küll portaali 2, kuid münt liigub samal ajal portaali 3. Kui uuesti kangist tõmmata, jõuame portaali 3 juurde, kus münt enne oli, aga münt liigub samal ajal portaali 4 ja nii edasi. Münti ei olegi võimalik kätte saada.



Kui aga osade portaalide vahel oleks võimalik liikuda, siis saab kavalalt liikudes mündi enda kätte manipuleerida. Paremal pool olevas näites saab mündi kätte nii: kõndida portaali 1 juurde, tõmmata kangist, kõndida portaali 4 juurde, tõmmata 3 korda kangist. Seejärel uuesti kõndida portaali 4 juurde ja tõmmata 3 korda kangist. Lõpuks oleme portaali 2 juures. Münt on meie seitsme kangitõmbega liikunud $2 \to 3 \to 4 \to 5 \to 1 \to 2 \to 3 \to 4$. Saame kõndida portaali 4 juurde ja mündi alati üles võtta.

Ent mitte alati ei ole portaalide vahel kõndimisest abi. Allolevas näites ei ole jälle münti võimalik üles korjata.



Seega ei olegi nii lihtne välja selgitada, milliseid münte on üles võimalik korjata. Väikeste K korral (näiteks $K \leqslant 4$) võib iseenesest kõik variandid kasvõi paberil läbi mõelda. Suuremates testides on vaja süsteemsemat lähenemist.

Esiteks paneme tähele, et kui piisavalt mitu korda järjest kangist tõmmata, jõuab kõik lõpuks algseisu tagasi. Kindlasti juhtub see $K! = 1 \cdot 2 \cdot 3 \cdots K$ kangitõmbe järel, kuigi see võib juhtuda ka varem.[3] Seega, kui tõmmata kangist $K!-1$ korda, liiguvad iga i kohta kõik portaali $A[i]$ juures olevad objektid portaali i. Nii on meil võimalik "-1 korda" kangist tõmmata.

[3] See juhtub kindlasti alati ka VÜK$(1, 2, \ldots, K)$ kangitõmbe järel, aga sõltuvalt konkreetsest permutatsioonist ka veelgi varem.

See tähendab, et kõiki käike on võimalik "tagasi võtta": kui astume naaberruudule, võime tagasi eelmisele ruudule astuda; kui tõmbame kangist, võime -1 korda kangist tõmmata. Seega võime mõelda igast mündist individuaalselt: kui valime mingi mündi ja selle mündini on üleüldse võimalik liikuda, siis liigume mingil viisil selle mündini, korjame üles ja läheme mööda tuldud teed alguspunkti tagasi. Kõik ülejäänud mündid on siis täpselt samadel positsioonidel, nagu alguses. Seega piisab meil ülesande lahendamiseks sellest, kui arvutame iga mündi kohta eraldi välja, kas seda on võimalik üles korjata: müntide kogumise järjekord ei loe ja kunagi ei teki olukorda, kus peame kahe mündi vahel valima.

Alamülesanded $K \leqslant 4$, $K \leqslant 8$, $K \leqslant 16$

Oma "olekut" saame kirjeldada kolmikuga (x, y, t), kus x ja y on ruudu koordinaadid, kus me asume, ning t on tehtud kangitõmmete arv. Ülal veendusime, et pärast teatud arv kordi kangi tõmbamisi jõuab kõik algseisu tagasi. Tähistame seda kordade arvu P. P väärtuseks võib võtta:

- faktoriaali $K! = 1 \cdot 2 \cdot 3 \cdots K$;
- vähima ühiskordse $\mathrm{VÜK}(1, 2, \ldots, K)$;
- antud permutatsiooni tsüklite pikkuste vähima ühiskordse.

Nimekirjas allpool olevad arvud on väikesemad ja seega paremad.

Niisiis võime olekuid $(x, y, 0)$ ja (x, y, P) lugeda samaväärseteks. Oma tegevuse käigus saame olekust (x, y, t) liikuda:

- olekutesse $(x+1, y, t)$, $(x-1, y, t)$, $(x, y+1, t)$ ja $(x, y-1, t)$, aga mitte siis, kui see vastab blokeeritud ruudule;
- kui ruudul (x, y) on portaal, mis viib ruudule (x', y'), siis olekusse $(x', y', (t+1) \bmod P)$.

Leiame tavalise BFS või DFS abil, millistesse olekutesse on võimalik jõuda.

Kui ruudul (x, y) ei ole portaali, aga on münt, siis saame mündi üles korjata juhul, kui on võimalik jõuda vähemalt ühte olekutest $(x, y, 0)$, $(x, y, 1)$, $(x, y, 2)$ jne. Lisaks peame kontrollima, milliseid portaalide juures olevaid münte on võimalik üles korjata: portaali i juures olevat münti on võimalik üles korjata siis, kui on võimalik jõuda ühte olekutest $(x_i, y_i, 0)$, $(x_{A_i}, y_{A_i}, 1)$, $(x_{A_{A_i}}, y_{A_{A_i}}, 2)$ jne.

Niisugune lahendus peab üle kontrollima $N \cdot M \cdot P$ olekut. Sõltuvalt sellest, millist P väärtust kasutati, võib see teenida 30 kuni 55 punkti.

Täislahendus

Joonistame suunatud graafi järgneval viisil: teeme tipu iga blokeerimata ruudu kohta. Siin dokumendis kasutame mugavuse jaoks indekseid, kus portaal i asub alati tipus i. Tippu, kust alustame, tähistame s. Kui u ja v on naaberruudud, siis lisame servad $u \xrightarrow{0} v$ ja $v \xrightarrow{0} u$. Lisaks lisame iga portaaliga ruudu i korral servad $i \xrightarrow{1} A_i$ ja $A_i \xrightarrow{-1} i$.[4] Saadud graafil on omadus: igale servale leidub vastupidise suuna ja kaaluga serv. Seega ka igale teekonnale leidub vastupidise suuna ja kaaluga teekond.

[4] Lahendust programmeerides võib serva $A_i \xrightarrow{-1} i$ lisamise ära jätta. Jääb lugejale mõtlemiseks, miks lahendus endiselt korrektne on.

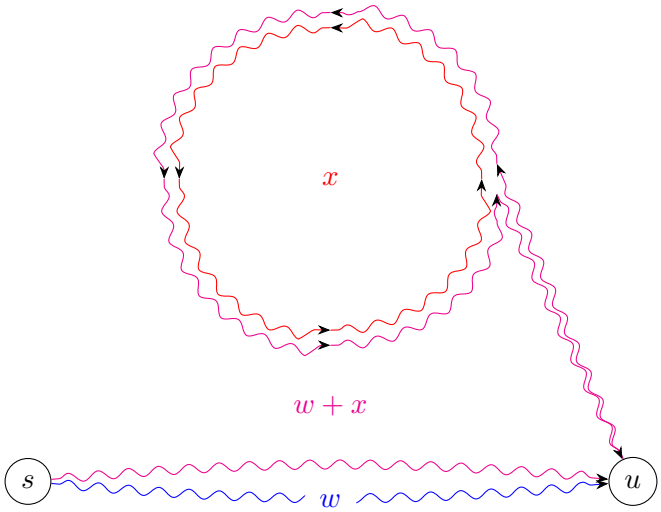
Nüüd saab ruudustikus liikuda ruudust u ruutu v täpselt k kangitõmbega parajasti siis, kui graafis leidub teekond $u \rightsquigarrow v$ kaaluga k. Viskame graafist välja kõik tipud, kuhu algtipust s teekonda ei leidu. Uurime, missuguste kaaludega teekonnad $s \rightsquigarrow u$ graafis leiduda saavad.

Olgu x minimaalse positiivse kaaluga tsükli kaal selles graafis (see tsükkel võib sama tippu või isegi serva mitu korda külastada). Nimetame sellist tsüklit x-tsükliks. Kui graafis ei ole ühtegi positiivse kaaluga tsüklit, tähendab see, et algruudust ei ole võimalik mööda blokeerimata ruute liikuda ühegi portaali i juurde, mille korral $A_i \neq i$. Võimalik, et ei ole üldse võimalik ühegi portaali juurde liikuda. Sel juhul on ülesanne klassikaline.[5]

Näitame, et ülejäänud juhtudel:

- kui leidub teekond $s \rightsquigarrow u$ kaaluga w, siis leiduvad ka teekonnad kaaluga $w + x$ ja $w - x$;
- kui graafis leidub tsükkel kaaluga w, siis $x \mid w$ (arv w jagub arvuga x);
- kui leidub teekond $s \rightsquigarrow u$ kaaluga w, siis ei leidu teekonda kaaluga y, kus $y \not\equiv w \pmod{x}$.

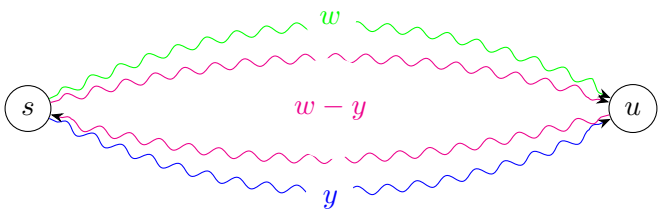
Esimese väite tõestuseks valime teekonna $s \rightsquigarrow u$ kaaluga w, liigume sealt x-tsükli juurde, teeme ühe ringi ja liigume tagasi tippu u. Nii saame teekonna $s \rightsquigarrow u$ kaaluga $w + x$. Kui liikuda mööda x-tsüklit vastupidises suunas (igale servale leidub ju vastupidise suuna ja kaaluga serv), siis saame hoopis teekonna kaaluga $w - x$.



Teise väite tõestuseks võtame tsükli kaaluga w, teeme seal ringi, liigume x-tsükli juurde, teeme seal ühe ringi ja siis liigume mööda tuldud teed tagasi. Saame tsükli, mille kaal on $w + x$. Analoogiliselt võime leida tsükli, mille kaal on $w - x$. Seega saame tsükli kaalu x kaupa vähendada ja suurendada. Kui w ei jaguks x-ga, siis saaksime nii tekitada tsükli, mille kaal jääks 0 ja x vahele: vastuolu eeldusega, et x-tsükkel on vähima positiivse kaaluga tsükkel.

Kolmanda väite tõestuseks võtame kaks teekonda $s \rightsquigarrow u$ kaaludega w ja y. Kui teine teekond ümber pöörata ja need kaks teekonda otsapidi kokku panna, saame tsükli, mille kaal on $w - y$. Seega vastavalt eelmisele väitele $x \mid w - y$ ehk $w \equiv y \pmod{x}$.

[5] Tegelikult see juht erilist eraldi käsitlemist ei vaja. Saab näidata, et alljärgnev lahendus lahendab ka selle juhu. Tuleb ainult (nulliga jagamise vältimiseks) $D[i] \equiv 0 \pmod{x}$ kontrollimiseks mitte teha `D[i] % x`.



Kuidas seda x aga leida? Leiame esiteks (näiteks laiuti või sügavuti läbimise teel) iga u jaoks mingi teekonna $s \rightsquigarrow u$ kaalu $D[u]$. Kui tippude u ja v vahel leidub serv kaaluga w, siis on graafis tsükkel kaaluga $D[u] + w - D[v]$. Järelikult $x \mid D[u] + w - D[v]$. Leiame kõikide selliste arvude $D[u] + w - D[v]$ suurima ühisteguri, tähistame selle g. Et x on kõikide $D[u] + w - D[v]$ jagaja, siis on ta ka nende suurima ühisteguri g jagaja, seega $x \mid g$.

Näitame, et teisest küljest $g \mid x$. Olgu minimaalse positiivse kaaluga tsükli tipud $u_1, u_2, \ldots, u_k$, kus $u_1 = u_k$, ja nendevaheliste servade kaalud $w_1, w_2, \ldots, w_k$. Teame, et

$$\begin{aligned} D[u_1] + w_1 &\equiv D[u_2] \pmod{g}, \\ D[u_2] + w_2 &\equiv D[u_3] \pmod{g}, \\ &\ldots \\ D[u_{k-1}] + w_{k-1} &\equiv D[u_k] \pmod{g} \end{aligned}$$

Need väited kokku pannes saame, et

$$\begin{aligned} D[u_1] + w_1 + w_2 + \cdots + w_k &\equiv D[u_k] \pmod{g} \\ D[u_1] + x &\equiv D[u_1] \pmod{g} \\ x &\equiv 0 \pmod{g} \end{aligned}$$

ehk $g \mid x$. Järelikult $g = x$ ehk g ongi selle minimaalse positiivse kaaluga tsükli kaal.

Nüüd on meil iga tipu kohta teada arv $D[u]$ ja teame, et teekond $s \rightsquigarrow u$ kaaluga w eksisteerib parajasti siis, kui $D[u] \equiv w \pmod{x}$. Kuidas selle informatsiooniga ülesannet lahendada? Keeruline on ainult nende müntidega, mis asuvad portaaliga ruutudel. Iga sellise mündi jaoks peame kontrollima, kas leidub teekond $s \rightsquigarrow i$ kaaluga 0 või $s \rightsquigarrow A_i$ kaaluga 1 või $s \rightsquigarrow A_{A_i}$ kaaluga 2 jne. Need väited on loomulikult samaväärsed: münt on võimalik üles korjata ainult juhul, kui $D[i] \equiv 0 \pmod{x}$.

6. 7-segmendilised indikaatorid (`ekraan`) 1 sek / 3 sek 100 punkti

Idee ja teostus: Heno Ivanov, lahenduse selgitus: Tähvend Uustalu

Sellistes ülesannetes tasub vähemalt esialgu läheneda võimalikult süsteemselt (kui just ei tule kohe mingit kavalat ideed). Kui hakata kohe alguses paberil välja mõtlema *ad hoc* reegleid stiilis "kui 5. bitt on sees ja 3. bitt väljas, aga eelmine kord oli sees, siis...", siis läheb lahendusprotsess kiiresti käest ära.

Ekraanil on $60\,480 = 5\,040 \cdot 2 \cdot 2 \cdot 3$ võimalikku "režiimi":

- 7! võimalust valida, milline bitt (igas 7-bitises grupis) vastab millisele segmendile.
- 2 võimalust valida, kas indikaatoritele vastavad grupid lähevad vasakult paremale või paremalt vasakule (mõtlemiseks: miks muud permutatsioonid võimalikud ei ole?).
- 2 võimalust valida, kas sisse lülitatud segmenti tähistab 0 või 1.
- 3 võimalust valida, kuidas arv joondatud on: kas tühikutega vasakule, tühikutega paremale või nullidega paremale.

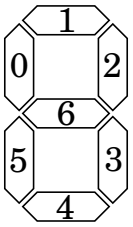
Igas testis võtame kõik need režiimid ette ja arvutame välja, mida ekraan antud bittide korral näitaks. Kui ekraanile ei ilmu korrektne arv (näiteks ei ole mõnel indikaatoril number, numbrite vahel on tühikud või arv on valesti joondatud), siis viskame selle režiimi kõrvale. Seejärel hakkame testimissüsteemilt järgmisi arve küsima. Igal korral heidame kõrvale režiimid, mis ei väljasta korrektseid arve, aga ka režiimid, mille väljastatud arv ei ole ühe võrra suurem eelmisest väljastatud arvust. Seda teeme nii kaua, kuni kõik allesjäävad režiimid väljastaksid ekraanile ühe ja sama arvu. See arv ongi vastus.

Siin on aga mitu ohtlikku kohta. Esiteks ei ole kindlasti mõtet oodata, kuni jääb alles vaid üks režiim. Näiteks kui testimissüsteemi antud kood on kahendsüsteemis

$$1111111\ 1111111\ 1111111\ 1111111\ 1111111,$$

siis teame kohe, et ekraanil on arv 88888. Tõepoolest, 1111111 saab vastata kas numbrile 8 (8) või tühikule . Ainult tühikutest koosnev ..... ei ole korrektne arv, seega peab ekraanil olema 8.... ehk 88888. Seejuures ei ole oluline, milline bitt vastab millisele segmendile: seitse sisse lülitatud bitti on igal juhul 8 (8). Režiime on sõelal väga palju, aga kõik nad annavad sama arvu.

Eriti salakavalad on vead joondamise ja arvu alguses olevate tühikutega. Vaatleme näiteks režiimi, kus väikesemad kümnendkohad on paremal, sisse lülitatud segmenti tähistab 1, arv on tühikutega joondatud paremale ning bitid vastavad alljärgnevatele segmentidele (kahendkohti loeme paremalt vasakule):



Oletame, et testimissüsteemi antud kood on kahendsüsteemis järgmine.

$$0000000\ 0000000\ 0111111\ 1111011\ 1001101.$$

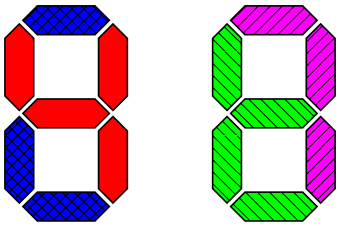
See tähendab, et ekraanile ilmub 88064: kaks tühikut ja seejärel 064. See aga ei ole korrektne tühikutega paremale joondatud arv, sest tühikutele järgneb null. Arv 64 näeb tühikutega paremale joondatult välja hoopis nii: 8864.

Niisugune lahendus saab 70 punkti. Igas testis on vaja läbi vaadata 60 480 režiimi; pärast ülimalt nelja koodi küsimist jääb sõelale ainult üks arv. Kokku 70 punkti väärt testides on ühes sessioonis ülimalt 100 testi. Halvimal juhul tähendab see, et neis testides on sessiooni jooksul vaja arvu dekodeerida $100 \cdot 4 \cdot 60\,480 \approx 2 \cdot 10^7$ korda. Väikese arvu punkte saab korjata ka ülejäänud testidest, kui koodis jälgida, millal aeg täis saab. Just niisugune lahendus on toodud failis `sol_brute.cpp`.

Ülejäänud 30 punkti saamiseks on vaja seda lahendust kuidagi optimeerida. Selleks on kindlasti palju erinevaid variante ja kõik laekunud (peaaegu) täislahendused teevad seda pisut isemoodi. Üks žürii koostatud täislahendus on aga järgmine.

Arvutame alguses (enne testide lahendamise juurde asumist) iga 7-kohalise bitijada kohta välja kõik permutatsioonid (bittide ja segmentide vastavused), mille korral sellest bitijadast tekib ekraanile korrektne number (või tühik). Näiteks bitijada 0001111 korral peab bittide ja segmentide vastavus olema ühel alltoodud kujudest.

- Bitid 0, 1, 2 ja 3 vastavad (mingis järjekorras) alloleval joonisel punastele segmentidele, bitid 5, 6, 7 sinistele segmentidele.
- Bitid 0, 1, 2 ja 3 vastavad (mingis järjekorras) alloleval joonisel rohelistele segmentidele, bitid 5, 6, 7 lilladele segmentidele.



Enamiku bitijadade korral on võimalikke permutatsioone 288 või 960. Kui jadas on seitse ühte või seitse nulli, siis on võimalikke permutatsioone 5 040, kui kuus ühte või kuus nulli, siis 2 160. Testi töötlema hakates leiame alguses kõikide antud koodi gruppide võimalike permutatsioonide hulkade ühisosa. Seda saab teha keerukusega $O(N \cdot M)$, kus M on kõige väikesema võimalike permutatsioonide arvuga grupi võimalike permutatsioonide arv. Valdavas enamikus testidest on M kas 288 või 960 ja pärast ühisosa võtmist jääb võimalike permutatsioonide arv veelgi väikesemaks.

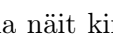
Nii saame ilma kõiki permutatsioone või režiime läbi vaatamata suure osa nendest välistada. See teeb lahenduse piisavalt kiireks, et 100 punkti teenida. Sellise lahenduse võib leida failist `sol_from_bits.cpp`.

Tasub märkida, et selline lahendus peab siiski mõnel juhul (näiteks, kui kõik numbrid on kaheksad) kõik režiimid läbi vaatama, ja kui testimissüsteem esitaks sessiooni, kus kõik arvud on kaheksaid täis, siis jääks ka see lahendus liiga aeglaseks.

Kui tahta lahendust veel optimeerida nii, et see ka niisugustel sessioonidel piisavalt kiirem oleks, tuleks välja mõelda lahendusele eraldi loogika näiteks juhuks, kui ekraanil olev arv koosneb ainult sümbolitest 8 ja 8. Teame, et oleme sellises juhus, kui masinalt saadud koodis on kõik 7-bitilised grupid kas 0000000 või 1111111.

Mõnikord on kohe selge, millise arvuga tegu on. Näiteks kui masin annab koodi

$$0000000\,0000000\,0000000\,1111111\,1111111\,1111111,$$

siis on masina näit kindlasti kas  või . Mõlemad vastavad arvule 888. Samuti on arv teada, kui kõik bitid on nullid või kõik ühed.

Enamasti on aga kaks võimalikku arvu. Kui masin annab koodi

$$1111111\,1111111\,1111111\,0000000\,0000000,$$

võib see vastata näitudele  ,  , ja ehk me veel ei tea, kas ekraanil on arv 888 või 88. Sellisel juhul teeme veel ühe päringu. Selle tagajärjel muutub üks bitigrupp: see, mis vastas enne numbrile  . Nüüd teame, kas sisse lülitatud segmendile vastab 0 või 1, ja seega ka seda, missugune arv ekraanil on.

Kui on soov siit veel edasi optimeerida, siis võib proovida välja mõelda erikäsitlemise ka juhu jaoks, kus arvus saavad leiduda lisaks  ja ka , , .

7. Ühetaoline tööaeg (`konstant`)

5 sek / 10 sek 100 punkti

Idee, teostus ja lahenduse selgitus: P. Randla

Antud ülesanne on üks vähestest Eesti informaatikaolümpiaadi ülesannetest, kus on kasutatud IOI-stiilis funktsioonidega sisendit-väljundit (tavaliselt on neid Eesti võistlustest vaid IOI valikvõistlustel). Seetõttu oli paljudel võistlejatel raskusi üldse mingisuguse lahenduse esitamisega.

Edasised probleemid olid juba rohkem lahendamisega seotud: prooviti võtta alamülesande standardne lahendus ja seda mudida, et see `secret_int`idega töötaks. Tihti tähendab see, et asjad, mis originaalses lahenduses võtsid ühe "sammu", võtavad nüüd N sammu, kus N on sisendi või mingi andmestruktuuri suurus. Seega võib niimoodi naiivselt tõlkimine programmi liiga aeglaseks muuta.

Siin on toodud C++ lahendused, analoogsed Pythoni lahendused on toodud `solution` kaustas.

Lisaväljakutse: Ülesande tekstis on möödaminnes mainitud, et ka korrutamine ja bitinihutamine võivad olla varieeruva tööajaga. Kõik alamülesanded on võimalik lahendada ka ilma ühegi korrutamiseta, ning kasutades bitinihkeid vaid nii, et salajane argument on alati vasakpoolne (ehk "nihke kaugus" on alati avalik).

Massiivi indekseerimine

See ülesanne on üsna sarnane ülesande tekstis näitena toodud massiivide võrdlemise ülesandega. Idee on käia läbi kogu massiiv ja iga elemendi juures vastust uuendada vaid siis, kui õige indeksi juures ollakse. Kasuks tuleb see, et võrdlusoperaator == tagastab arvu 1 või 0, mida saab siis korrutamises kasutada.

```
secret_int ith_element(std::vector<secret_int> A, secret_int i) {
    secret_int res = 0;
    for (int j = 0; j < A.size(); j++) {
        res += (i == j) * A[j];
    }
    return res;
}
```

Massiivi sorteerimine

Esimese alamülesande saab ära lahendada üpris mitme klassikalise $O(N^2)$ sorteerimisalgoritmiga. Üks võimalus põhineb pistemeetodil (ingl *insertion sort*):

```
// võrdleb kahte elementi ja vahetab need, kui need vales järjekorras on
void cmp_sort(secret_int& a, secret_int& b) {
    secret_int a_lt_b = (a < b);
    secret_int new_a = a_lt_b * a + (1 - a_lt_b) * b;
    secret_int new_b = a_lt_b * b + (1 - a_lt_b) * a;
    a = new_a; b = new_b;
}
void sort_arr(std::vector<secret_int>& A) {
    for (int i = 1; i < A.size(); i++) {
        for (int j = i; j > 0; j--) {
            cmp_sort(A[j-1], A[j]);
        }
    }
}
```

Teise alamülesande jaoks on vaja kasutada mõnda kiiremat algoritmi. Suurem osa klassikalisi $O(N \log N)$ sorteerimisalgoritme ei ole kahjuks eriti sobilikud, kuna nende tööaeg sõltub väga tugevalt sisendmassiivi väärtustest. Siiski leidub sobivaid algoritme, mis põhinevad niinimetatud "sorteerimisvõrgustikel" (ingl *sorting network*), millel on sarnane struktuur ülaltooduga: igal sammul võrreldakse ja vajadusel vahetatakse kaks massiivi elementi. Üks võimalik realisatsioon on näiteks Batcheri paaris-paaritu ühildusmeetod:

```
void sort_arr(std::vector<secret_int>& A) {
    int n = A.size();
    for (int p = 0; 1<<p < n; p++) {
        for (int k = p; k >= 0; k--) {
            int K = 1<<k;
            for (int j = (k == p ? 0 : K); j < n-K; j += 2*K) {
                for (int i = 0; i < K && i < n-j-K; i++) {
                    if ((i+j) >> (1+p) == (i+j+K) >> (1+p)) {
                        cmp_sort(A[i+j], A[i+j+K]);
                    }
                }
            }
        }
    }
}
```

Graafi sidususkomponentide arvu leidmine

Antud ajalimiidi ja sisendi suuruste juures piisab $O(N^3)$ lahendusest. Selleks, et $O(N^3)$ kätte saada, piisab omakorda lihtsast lahendusest, mis iga võimaliku serva jaoks (mida on kokku ligikaudu $N^2/2$) märgib mingil sobival viisil ära, et need tipud on samas komponendis (tehes seda $O(N)$ ajas), ja loendab pärast komponendid kokku. Üks võimalik lahendus on järgnev, kus hoitakse massiivi iga tipu komponendinumbriga.

```
secret_int connected_components(std::vector<std::vector<secret_int>> A) {
    int n = A.size();
    std::vector<secret_int> components;
    for (int i = 0; i < n; i++) {
        components.push_back(i);
    }
    secret_int num_components = n;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            // kas need tipud tuleb ühendada?
            secret_int do_conn = A[i][j];
            // kui juba ühendatud ei olnud, siis on sidususkomponente vähem
            num_components -= do_conn * (components[i] != components[j]);
            // asendame kõik components[i] esinemised väärtusega components[j]
            secret_int from = components[i];
            secret_int delta = do_conn * (components[j] - components[i]);
            for (int k = 0; k < n; k++) {
                components[k] += (components[k] == from) * delta;
            }
        }
    }
    return num_components;
}
```

Lisaks osutusid üsna edukaks ka lahendused, kus maatriksit A muudeti nii, et $A[i][j]$ näitas lõpuks, kas tippude i ja j vahel leidub mingi tee, ning leiti sellest sidususkomponentide arv.

Suunatud graafis teekonna leidmine

Tippude vaheliste kauguste leidmiseks võib kasutada erinevaid viise: näiteks Floyd-Warshalli algoritmiga on võimalik leida kõikide tipupaaride vahelised kaugused. Samas võib kasutada ka laiuti läbimisest inspireeritud algoritmi ja leida iga tipu kaugus alguspunktist. Mõlemal juhul on keerukus $O(N^3)$. Lisaks tuleb meeles pidada iga tipu jaoks selle eellast, et hiljem teekond rekonstrueerida. Samuti võib abiks olla ka kohe kauguste leidmise ajal juba leitud teekonna pikkuse eraldi meeles pidamine.

Kui kaugus on leitud, siis tuleb rekonstrueerida ka teekond. Floyd-Warshalli algoritmiga leitud kaugusmaatriksi puhul võtab see samuti $O(N^3)$ operatsiooni, kuid kui laiuti läbimisega on leitud kaugused ainult alguspunktist, siis on see vaid $O(N^2)$. Lisaks tuleb teekond leida õiges järjekorras. Klassikalise "leiame teekonna tagurpidi ja pärast teeme `reverse()`" lähenemisega tekib probleem, kuna teekonna pikkus ei ole teada ja ümber tuleks pöörata ainult osa vastusemassiivist. See on küll võimalik (näiteks ühes žürii lahenduses on nii tehtud), kuid lihtsam on hoopis kogu ülesanne tagurpidi lahendada, s.t. otsida teekonda punktist C punkti B ümberpööratud graafis. Sellisel juhul tekib teekond üsna loomulikult õiges järjekorras.

Kummastki ideest on toodud lahendused failides `sol.cpp` ja `sol_slow.cpp`.

Algarvulisuse kontrollimine

Esimeses alamülesandes piisab, kui teha nimekiri kõigist algarvudest, mis on väiksemad kui 10^6, ja siis kontrollida, kas antud arv on mõnega neist võrdne.

Teise alamülesande jaoks tuleb juba natuke rohkem vaeva näha. Žürii lahendus realiseerib Milleri-Rabini algarvulisuse kontrolli, kasutades fikseeritud aluseid 2, 7, ja 61, millest piisab kõigi 32-bitiste arvude puhul algarvulisuses veendumiseks. Tuleb tähele panna, et sellises lahenduses on vaja salajasi arve korrutada ja astendada salajase mooduli järgi, mis tuleb ka ise realiseerida. Selline lahendus on toodud failis `sol.cpp`.

Samas on võimalik maksimumpunktid saada ka kavalamalt jagamist kasutades. Kui sisendarv on kordarv ja väiksem kui 2^{32}, siis leidub sellel algtegur, mis on väiksem kui 2^{16}. Seega saab proovida antud arvu jagada kõigi algarvudega vahemikus 2 kuni 65521. Naiivselt bitikaupa jagamist realiseerides tuleb lahendus aga liiga aeglane (täpsemalt ületab see 10^6 tehte piiri). Siis tuleb jagamist optimeerida, kasutades teadmist, et jagaja on avalik ja peale selle ka võrdlemisi väike: maksimaalselt 16 bitti. Selgub, et jagamisfunktsioonis saab iteratsioonide arvu vähendada selle võrra, mitu jagaja ülemist bitti on 0; seda saab loendada näiteks funktsiooni `__builtin_clz` abil.

Viimase asjana tuleb tähele panna ka seda, et 1 ei ole algarv ja et 2 on ainus paarisarvuline algarv, ja kontrollida, et programm ka nendel juhtudel õige vastuse annab.