Sisukord

Teetegu **2**

Avaldis **3**

Sipelgas **5**

Noorim algkoosseis **7**

Elukvaliteediindeks **9**

Sonic 3 & Knuckles **13**

Poolik tekst **25**

1. Teetegu (tee)

1 sekund 20 punkti

Idee ja teostus: Sandra Schumann, lahenduse selgitus: Ahto Truu

> Veekannus on 100-kraadine vesi ja tunni aja pärast on jälle vaja 100-kraadist vett. Leida, kui palju kulub energiat (a) veekannu soojas hoidmiseks ja (b) selle õigeks ajaks uuesti 100 kraadini kuumutamiseks, kui sellel vahepeal jahtuda lasta. On teada, et veekannu kuumutamiseks kulub A kJ/° ja B sek/°, et veekannu soojas hoidmiseks kulub C kJ/min, et ilma soojas hoidmiseta jahtub see D°/min (kus A, B, C, D on kõik võrdlemisi väikesed reaalarvud) ja et vesi ei jahtu kunagi alla 22 kraadi.

Vastus esimesele küsimusele on üsna lihtne: kui vee soojas hoidmiseks kulub C kJ/min ja seda on vaja soojas hoida 1 tund = 60 min, siis on energiakulu kokku $60 \cdot C$ kJ.

Üks võimalik arutluskäik teisele küsimusele vastamiseks: kui vee jahtumine 22 kraadi juures ei peatuks, oleks selle temperatuur t minuti jahtumise järel $100 - t \cdot D$ kraadi; sellelt temperatuurilt alustades kuluks vee 100 kraadini kuumutamiseks $(100 - (100 - t \cdot D)) \cdot B = t \cdot D \cdot B$ sekundit ehk $t \cdot D \cdot B/60$ minutit; seega oleks vesi t minuti jahtumise ja siis uuesti kuumutamisega 100 kraadi juures tagasi $t + t \cdot D \cdot B/60 = t \cdot (1 + D \cdot B/60)$ minuti järel; selleks, et vesi oleks uuesti 100-kraadine tunni aja pärast, peab $t \cdot (1 + D \cdot B/60) = 60$ ehk $t = 60/(1 + D \cdot B/60)$.

Kui vesi jahtuks t minutiga alla 22 kraadi, siis tegelikult jääb see 22 kraadi peale ja siis selle uuesti 100 kraadini kuumutamiseks kulub $(100 - 22) \cdot A = 78 \cdot A$ kJ.

Kui vesi jõuab tõesti $100 - t \cdot D$ kraadini jahtuda, siis kulub selle uuesti 100 kraadini kuumutamiseks $(100 - (100 - t \cdot D)) \cdot A = t \cdot D \cdot A$ kJ.

Tulemuseks saame lahendused, mille kood on lühem kui eelnev arutelu.

Python:

```python
a, b, c, d = [float(_) for _ in input().split()]

print(60 * c)

t = 60 / (1 + d * b / 60)
if 100 - t * d < 22:
  print(78 * a)
else:
  print(t * d * a)
```

C++:

```cpp
#include <iostream>

int main() {
  double a, b, c, d;
  std::cin >> a >> b >> c >> d;

  std::cout << 60 * c << '\n';

  const double t = 60 / (1 + d * b / 60);
  if (100 - t * d < 22) {
    std::cout << 78 * a << '\n';
  } else {
    std::cout << t * d * a << '\n';
  }
}
```

2. Avaldis (avaldis)

1 sekund 30 punkti

Idee ja teostus: Rein Prank, lahenduse selgitus: Ahto Truu

> Antud täisarvud A ja B. Väljastada avaldise $A \cdot x + B$ normaalkuju.

Selle ülesande lahendamist alustades võiks kõigepealt läbi mõelda kõik võimalikud erijuhud eraldi lineaar- ja vabaliikme jaoks. Lineaarliikme jaoks saame:

1. Kui $A = 0$, siis lineaarliiget välja ei kirjutata.
2. Kui $A = 1$, siis kordajat välja ei kirjutata ja lineaarliikme kuju on x.
3. Kui A on muu positiivne arv, siis kirjutatakse välja ka kordaja (ilma märgita) ja lineaarliikme kuju on Ax.
4. Kui $A = -1$, siis kirjutatakse kordajast välja ainult märk ja lineaarliikme kuju on $-x$.
5. Kui A on muu negatiivne arv, siis kirjutatakse välja ka kordaja (koos märgiga) ja lineaarliikme kuju on jälle Ax.

Kõigis programmikeeltes, kus negatiivsed arvud väljastatakse vaikimisi märgiga ja positiivsed ilma, võib juhud 3 ja 5 üheks kokku võtta.

Vabaliikme erijuhte läbi mõeldes märkame, et need sõltuvad lineaarliikme olemasolust:

1. Kui $B = 0$, siis vabaliiget üldiselt välja ei kirjutata. Välja arvatud juhul, kui ka $A = 0$, sest mõlema liikme ärajätmisel saaks me tulemuseks tühja väljundi, mis pole ka õige.
2. Kui B on positiivne, siis tuleks ta vabaliikmena väljastada koos märgiga. Välja arvatud juhul, kui $A = 0$, sest üksiku arvu ette plussmärki ei kirjutata.
3. Kui B on negatiivne, siis tuleks ta igal juhul väljastada koos märgiga.

Eelnevaid tingimusi kaaludes osutub otstarbekaks programm struktureerida nii, et kõigepealt käsitleme eraldi juhu $A = 0$: siis koosneb avaldise üldkuju parajasti lineaarliikmest väljastatuna tavaliste arvu esitamise reeglite järgi. Ja kui juba on teada, et lineaarliige on mingisugusel kujul olemas, lihtsustab see oluliselt ka vabaliikme väljastamise loogikat.

Python:

```
a, b = [int(_) for _ in input().split()]
if a == 0:
    # kui lineaarliige on null, väljastame igal juhul vabaliikme
    vastus = str(b)
else:
    # lineaarliige
    if a == 1:
        vastus = 'x' # kordajat 1 välja ei kirjuta
    elif a == -1:
        vastus = '-x' # kordajast -1 kirjutame ainult märgi
    else:
        vastus = str(a) + 'x' # muul juhul väljastame kordaja enda
    # vabaliige
    if b < 0:
        vastus = vastus + str(b) # negatiivse arvu ees on miinus vaikimisi
    elif b > 0:
        vastus = vastus + '+' + str(b) # positiivse ette lisame plussi ise
    else:
        pass # nulli välja ei kirjuta
print(vastus)
```

C++:

```
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;

  if (a == 0) {
    // kui lineaarliige on null, väljastame igal juhul vabaliikme
    std::cout << b;
  } else {
    // lineaarliige
    if (a == 1) {
      // kordajat 1 välja ei kirjuta
      std::cout << "x";
    } else if (a == -1) {
      // kordajast -1 kirjutame ainult märgi
      std::cout << "-x";
    } else {
      // muul juhul väljastame kordaja enda
      std::cout << a << "x";
    }
    // vabaliige
    if (b < 0) {
      // negatiivse arvu ees on miinusmärk juba niigi
      std::cout << b;
    } else if (b > 0) {
      // positiivse arvu ette peame plussi ise lisama
      std::cout << "+" << b;
    } else {
      // nulli välja ei kirjuta
    }
  }
  std::cout << "\n";
}
```

3. Sipelgas (sipelgas)

1 sekund 40 punkti

Idee, teostus ja lahenduse selgitus: Targo Tennisberg

> Risttahukal asuvad sipelgas ja meetilk, antud oma koordinaatidega. Leida sipelga lühim tee meetilgani.

Seda ülesannet on kõige lihtsam lahendada kääride ja paberiga. Lõika paberist välja vastavate mõõtmetega risttahuka pinnalaotus, kus mõned tahud "jäävad üle" ehk siis meetilgaga tahku on erinevatel viisidel dubleeritud (vaata eeskujuna alltoodod jooniseid). Nüüd voldi risttahukas kokku ja märgi sellele sipelga ning meetilga asukohad. Nüüd pane tähele, et kui see pinnalaotus taas lahti voltida, siis sipelga lühim teekond tähendab lihtsalt pinnalaotusel vastavate punktide vahele sirglõigu tõmbamist. Ainus vajaminev geomeetriline teadmine on Pythagorase teoreem.

Järgmine raskus tekib sellega, et variante tundub olevat väga palju ja nende ükshaaval läbi programmeerimine oleks ühelt poolt tüütu ja teiselt poolt on üsna kindel, et selles tekib palju vigu. Variantide läbivaatamise vältimiseks on hea pöörata risttahukas "kanoonilisse asendisse", näiteks nii, et sipelgas on alati ülemisel tahul.

Risttahuka pööramiseks võib realiseerida näiteks järgmised teisendused:

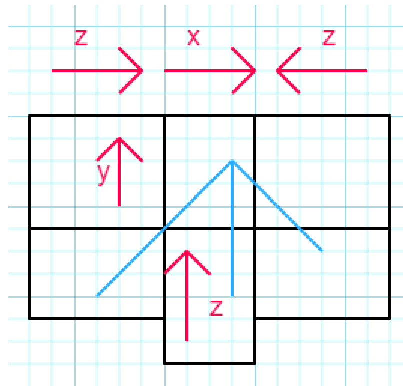
```
void RotX() {
  (Yr, Zr) = (Zr, Yr);
  (Zs, Ys) = (-Ys + Zr, Zs);
  (Zm, Ym) = (-Ym + Zr, Zm);
}

void RotY() {
  (Xr, Zr) = (Zr, Xr);
  (Xs, Zs) = (-Zs + Xr, Xs);
  (Xm, Zm) = (-Zm + Xr, Xm);
}

void RotZ() {
  (Xr, Yr) = (Yr, Xr);
  (Xs, Ys) = (-Ys + Xr, Xs);
  (Xm, Ym) = (-Ym + Xr, Xm);
}
```

Edasi on kolm võimalust:

1. Kui mesi on samuti ülemisel tahul, on kauguse leidmine triviaalne.
2. Kui mesi on mõnel külgmisel tahul, on kõige lihtsam see tahk ette keerata ning edasi on kolm võimalust: liikuda kas otse, üle vasaku nurga või üle parema nurga. Nendest variantidest miinimumi võtmine annabki õige vastuse.

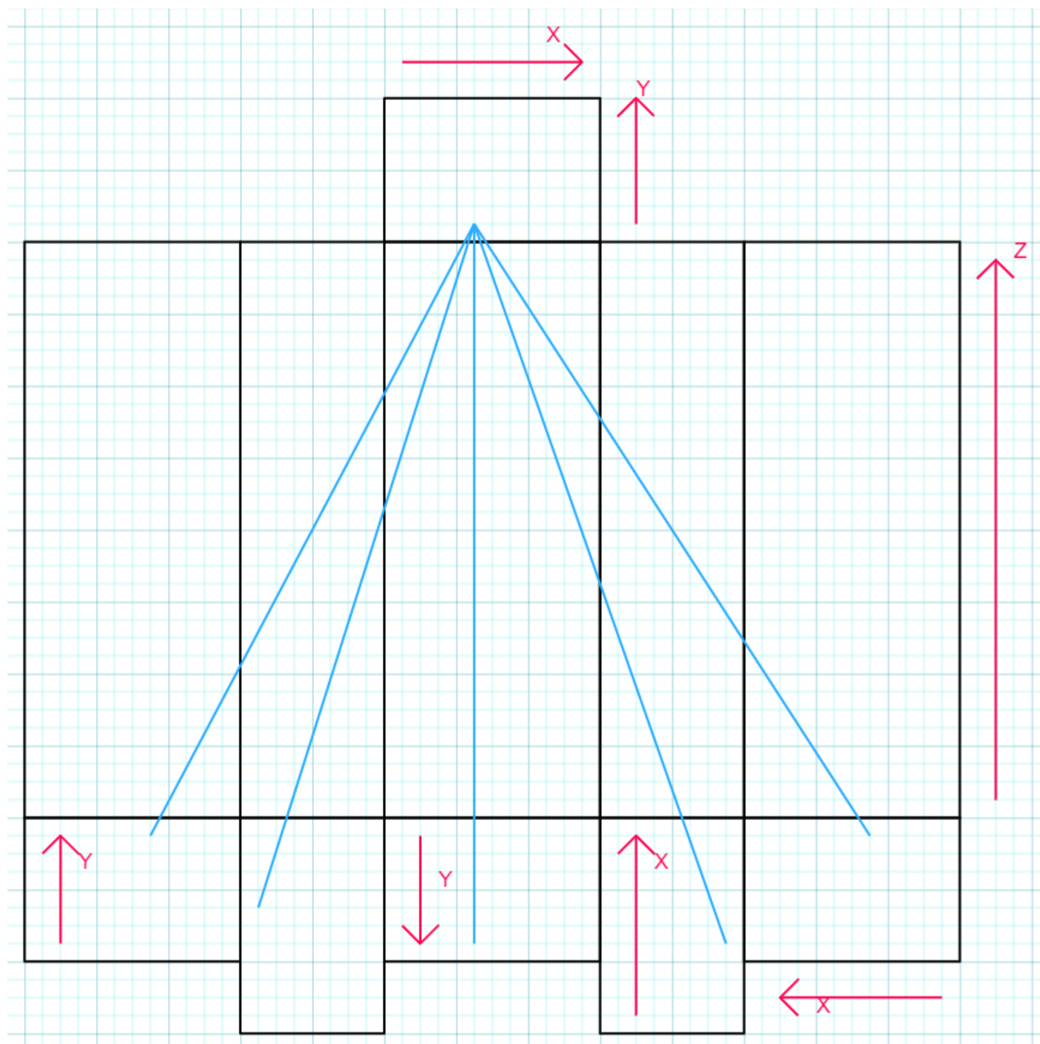
   Võttes ülesande tekstist teise näite ning keerates seal sipelga ülemisele ja mee eesmisele tahule, saame joonisel 1 näidatud situatsiooni, kus sinised jooned illustreerivad kolme võimalikku teed. Nende teede pikkuste võrdlemine annab ülesande tekstis antud vastuse.

   Visualiseerimise abistamiseks võib oma väljalõikele peale märkida erinevate koordinaattelgede suunad, mille põhjal on hea leida järgmistes sammudes kasutatavad valemid.



Joonis 1: Sipelgas ülemisel, mesi eestahul

3. Kui mesi on alumisel tahul, on sinna liikumiseks viis võimalust: otse, üle vasaku tahu, üle kahe vasaku tahu, üle parema tahu ning üle kahe parema tahu. Minimaalne nendest võimalustest ongi õige vastus. Joonisel 2 on toodud näide, kus risttahuka mõõtmed on $12, 8, 32$, sipelgas asub koordinaatidel $(5, 1, 32)$ ning mesi koordinaatidel $(5, 7, 0)$. Viis korda Pythagorase teoreemi kasutamine annab tulemuseks, et lühim tee on üle kahe vasaku tahu.



Joonis 2: Sipelgas ja mesi vastastahkudel

4. Noorim algkoosseis (`nak`)

3 sek / 10 sek 60 punkti

Idee, teostus ja lahenduse selgitus: Jaagup Tamme

Antud täisarvumassiiv V. Leida etteantud lõikudes suuruselt üheteistkümnes arv.

Selle võistluse esimene ülesanne, kus tuleb olulisel kohal mängu ajalimiit. Paljud osalejad kirjutavad valmis ilmselgena näiva lahenduse, aga pärast selle esitamist vaatab neile vastu skoor 20/60 ja teade “programmi täitmise ajalimiit ületatud”.

Peaaegu iga informaatikaülesannet on võimalik *kuidagimoodi* lahendada, kui programmeerimine piisavalt hästi käpas on. Mõnikord on selliseks lahenduseks variantide läbivaatus, kus läbi vaadatavate variantide arv kasvab kiiresti astronoomiliseks. Mõnikord ei ole asi küll nii hull, aga siiski mitu suurusjärku liiga aeglane. Näiteks selles ülesandes tähendab “naiivne” lähenemine iga päringu (L, R) korral massiivi kõikide elementide $V_L, \ldots, V_R$ läbi vaatamist, sorteerimist ja tulemuse üheteistkümnenda elemendi väljastamist. Isegi kui sorteerimine kõrvale jätta, tähendab see, et kokku võib vaja minna $N \cdot Q = 10^{10}$ elemendi läbivaatamist. See aga on liiga aeglane: lihtsustatult võib öelda, et ühes sekundis saab teha umbes 10^8 sellist “läbivaatamist”.

Raskemate ülesannete lahendamine ei tähenda niisiis vaid mingisuguse toimiva programmi kirjutamist, vaid *piisavalt efektiivse* toimiva programmi kirjutamist. Siin ei ole ka eriti abi teatud pisioptimeerimisest: näiteks keeles C++ ei muuda `i++` asemel `++i` kasutamine praktiliselt mitte midagi. Eesti olümpiaadil, kus Pythoni ja C++ jaoks on eraldi ajalimiit, ei ole suurt abi ka lahenduse teise keelde ümber kirjutamisest.

Vaja on sisulisi ideid. Nendeks on ühelt poolt klassikalised algoritmid ja andmestruktuurid, teiselt poolt aga ülesandespetsiifilised tähelepanekud.

Vaatame lahendusi alamülesannete kaupa:

1. $11 \leqslant N \leqslant 10^3$ ja $1 \leqslant Q \leqslant 10^3$:

   Iga päringu jaoks valime massiivist V välja vaadeldavate mängijate vanused, sorteerime need kasvavalt ning väljastame üheteistkümnenda.

2. $11 \leqslant N \leqslant 10^3$:

   Enne päringutele vastamist teeme vanuste massiivi peal eeltöötlust. Nimelt salvestame iga lõigu jaoks selle lõigu vanuselt üheteistkümnenda mängija vanuse. Tulemuste kiireks arvutamiseks vaatame iga vasaku otspunkti jaoks läbi kõik paremad otspunktid kasvavas järjekorras, igal sammul lisades ühe uue mängija. Kui hoiame alati meeles, mis on lõigu üheteistkümne noorima mängija vanused, siis uue mängija lisamiseks lõiku: lisame ta vanuse lõigu noorimate hulka, sorteerime vanused ning eemaldame vanima mängija vanuse.

   Iga lõigu tulemuse salvestame otspunktide põhjal $N \times N$ tabelisse ning päringutele vastamiseks väljastame tabeli väärtuse kohal (L, R), s.t tabeli reas L veerus R oleva elemendi.

3. Igas päringus $R - L + 1 = 11$:

   Iga päring on täpselt üheteistkümne mängija kohta, seega vastus on küsitavas lõigus vanima mängija vanus. Üks võimalus on vanuste massiivi peale ehitada lõikude puu, salvestades maksimumväärtust. Teine võimalus on samamoodi eeltöötlust teha nagu eelmises punktis, kuid mitte kõigi lõikude, vaid ainult 11 elemendi pikkuste lõikude vastuseid salvestades.

   Samas, kui esimeses punktis kirjeldatud lahendus efektiivselt implementeerida, siis peaks see ka selle testigrupi testid piisavalt kiiresti lahendama.

4. $55 \leqslant V_i \leqslant 56$ iga i korral:

   Iga päringu vastuseks on kas 55 või 56. Täpsemalt on vastus 55, kui lõigus L kuni R on vähemalt 11 mängijat vanusega 55; vastasel juhul on vastus 56. Selle tingimuse kiireks kontrollimiseks salvestame iga i jaoks, mitu mängijat särginumbritega 0 kuni i on vanusega 55; olgu see arv Y_i. Nüüd lõigus L kuni R on 55-aastaseid täpselt $Y_R - Y_{L-1}$ ning selle arvu põhjal väljastame vastuse.

5. Lisapiirangud puuduvad:

   Arvatavasti kõige lihtsam viis täislahenduseni jõuda on täiendada neljanda alamülesande lahendust. Arvutame iga vanuse v ja iga i jaoks, mitu mängijat särginumbritega 0 kuni i on mitte vanemad kui v; tähistame seda arvu $Y_{v,i}$. Y maatriksi kiireks arvutamiseks vaatame iga v jaoks i-sid kasvavas järjekorras. Kui me teame väärtust $Y_{v,i-1}$ ja vanust V_i, siis $Y_{v,i}$ on $Y_{v,i-1} + 1$, kui $V_i \leqslant v$, ning vastasel juhul $Y_{v,i-1}$. Päringule vastamiseks peame leidma vähima v nii, et $Y_{v,R} - Y_{v,L-1} \geqslant 11$. Seda saab teha kas kordusega üle kõigi võimalike v väärtuste või kasutades kahendotsingut, mõlemad moodused peaksid ajalimiiti mahtuma. Seda lahendust edasi arendades on võimalik lainikumaatriksit (ingl *wavelet matrix*) kasutades mälukasutust paarkümmend korda vähendada.

   Teine võimalus on edasi arendada teise alamülesande lahendust, salvestades mingite lõikude ühteteistkümmet noorimat vanust. Paslik on kasutada näiteks lõikude puud või hõredat tabelit, mille igas tipus hoiame vastava lõigu noorimate mängijate vanuseid. Kahe ühisosata lõigu noorimate mängijate vanuste kombineerimiseks võime (a) lihtsalt vastavad loetelud kokku panna, ära sorteerida ja 11 vähimat arvu alles jätta, või (b) kasutada standardset kahe sorteeritud järjendi ühildamise (ingl *merge*) meetodit. Kiirema ja vähem mälu kulutava programmi saamiseks tasub üksikuid vanuseid hoida ühebaidistena ja 11 noorimat vanust fikseeritud pikkusega massiivina (`array<short,11>` keeles C++ ja `array('b')` keeles Python).

   Kolmas võimalus on lahendada üldisem ülesanne: leida väärtuselt K-s arv lõigus L kuni R. Enamik täislahenduseni jõudnutest kasutas justnimelt seda ideed. Püsiva lõikude puu (ingl *persistent segnment tree*) või püsiva Fenwicki puu (ingl *persistent Fenwkick tree*) abil saame iga indeksi i jaoks leida, mitu arvu lõigus 0 kuni i on mitte suuremad kui mingi etteantud v. Eeltöötluse käigus käime läbi indeksid i kasvavas järjekorras ning lisame arvule V_i puus $i - 1$ ühe esinemise juurde, nii saame puu nr i.

5. Elukvaliteediindeks (eki)

1 sek / 5 sek 100 punkti

Idee ja lahenduse selgitus: Tähvend Uustalu, teostus: Kregor Ööbik

> Antud N riiki, millest igaühe kohta on teada kolm statistilist näitajat. Antud on ka M nõuet kujul “riik A peab olema pingereas riigist B eespool”. Leia statistilistele näitajatele mittenegatiivsed kaalud nii, et nendest moodustuv pingerida rahuldaks ülesande tingimusi.

See ei ole graafiülesanne.

Nõue, et riik A peab olema pingereas eespool kui riik B, tähendab, et

$$\lambda_1 X_1 + \lambda_2 X_2 + \lambda_3 X_3 > \lambda_1 Y_1 + \lambda_2 Y_2 + \lambda_3 Y_3, \tag{1}$$

kus (X_1, X_2, X_3) ja (Y_1, Y_2, Y_3) on vastavalt riikide A ja B statistilised näitajad.

Kaalude mittenegatiivsuse tõttu peab lahendis (kui see üldse leidub) kindlasti kehtima kas $\lambda_3 > 0$ või $\lambda_3 = 0$. Proovime oma lahenduses mõlemad variandid läbi.

Kui $\lambda_3 = 0$, siis saame võrratuse (1) viia kujule

$$\lambda_1 X_1 + \lambda_2 X_2 > \lambda_1 Y_1 + \lambda_2 Y_2$$
$$(X_1 - Y_1)\lambda_1 + (X_2 - Y_2)\lambda_2 > 0.$$

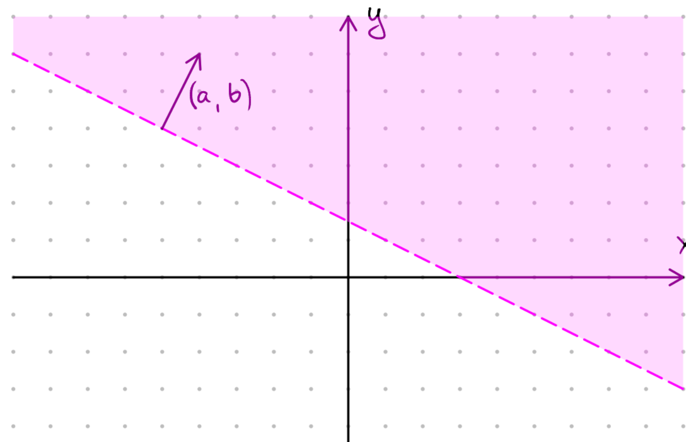
Kui aga $\lambda_3 > 0$, siis võime üldisust kitsendamata eeldada, et lahendis $\lambda_3 = 1$: kuna λ_3 on positiivne, siis võrratuse (1) arvuga λ_3 läbi jagamine võrratuse tõesust ei muuda. Nüüd saame võrratuse (1) viia kujule

$$\lambda_1 X_1 + \lambda_2 X_2 + X_3 > \lambda_1 Y_1 + \lambda_2 Y_2 + Y_3$$
$$(X_1 - Y_1)\lambda_1 + (X_2 - Y_2)\lambda_2 > Y_3 - X_3.$$

Mõlemal juhul taandub ülesanne kujule:

> Antud M võrratust kujul $a\lambda_1 + b\lambda_2 > c$, kus a, b, c on teada ja λ_1, λ_2 tundmatud. Leia paar (λ_1, λ_2), kus $\lambda_1, \lambda_2 \geqslant 0$ ja mis rahuldab kõiki võrratusi.

Tähistame geomeetrilise intuitsiooni eesmärgil edasises $x = \lambda_1$ ja $y = \lambda_2$. Geomeetriliselt näeb kõikide võrratust $ax + by > c$ rahuldavate punktide hulk välja järgmine:



Joonis 3: Pooltasand

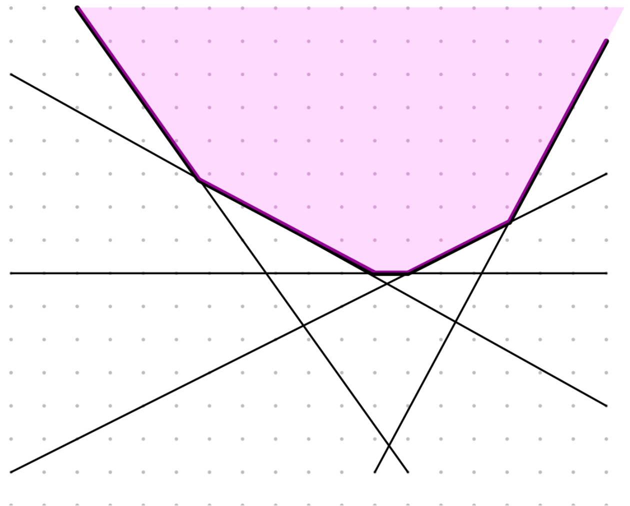
Tasandil on sirge, mille defineerib võrrand $ax + by = c$. Sirgega on risti vektor (a, b). Võrratust rahuldavad need punktid, mis jäävad sellest sirgest rangelt sellele poole, kuhu on suunatud vektor (a, b).

Sellist kujundit nimetatakse *pooltasandiks*. Meil on antud palju pooltasandeid ja peame kontrollima, kas leidub punkt, mis asub kõikides nendes pooltasandites. Või teiste sõnadega: peame kontrollima, kas leidub punkt kõikide nende pooltasandite ühisosas. Seda ülesannet nimetatakse *pooltasandite ühisosa ülesandeks* (ingl *halfplane intersection*) ja tegu on küllaltki tuntud ülesandega. Kirjeldame üht võimalikku lahendust.

Abiks võivad olla ka järgmised artiklid:

- `https://cp-algorithms.com/geometry/halfplane-intersection.html` — kirjeldab sama algoritmi, mis meie;
- `https://codeforces.com/blog/entry/61710` — pisut erinev, aga üldiselt sarnane lähenemine;
- `https://codeforces.com/blog/entry/63823` — sisuliselt üks osa lahendusest.

Vaatleme esialgu ainult neid pooltasandeid, mille vektor (a, b) on suunatud üles (s.t. $b > 0$). Nende ühisosa on "kausi" kujuga. "Kaussi" piirab alt hulk sirglõike; ülalt aga mitte midagi.



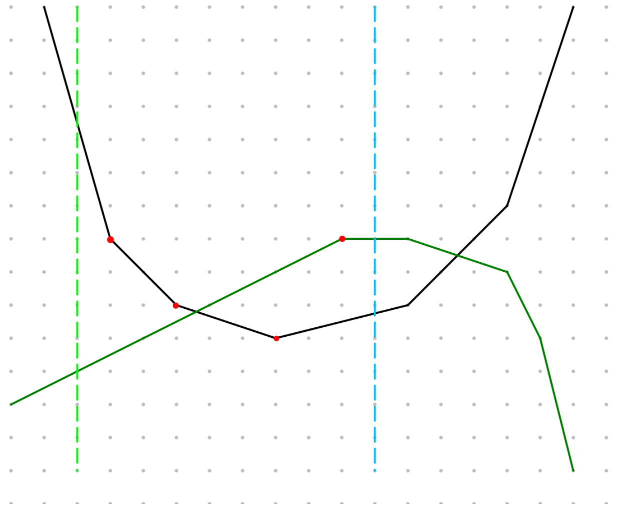
Joonis 4: Pooltasandite ühisosana moodustunud "kauss"

Meie eesmärk on leida "kaussi" alt piiravad sirglõigud. Paneme tähele, et iga sirge "domineerib" ülimalt ühe korra, ja sirged domineerivad tõusu kasvavas järjekorras ehk $\frac{a}{b}$ kahanevas järjekorras. Sorteerime sirged $\frac{a}{b}$ kahanevas järjekorras ja hakkame kaussi vasakult poolt ehitama. Hoiame seda kaussi meeles pinu (ingl *stack*) kujul, kus pinu iga element on paar (x_s, s). Siin s tähistab sirget ja x_s punkti, kus sirge s domineerima hakkab.

- Iga sirge $ax + by = c$ kohta:
  - Kui pinu on tühi, siis lisame pinusse paari $(-\infty, ax + by = c)$ ja läheme järgmise sirge juurde.
  - Olgu pinu pealmine element (x_s, s).
  - Seni kuni sirgete $ax + by = c$ ja s lõikepunkt on väiksem, kui x_s:

  * Eemaldame pinu pealmise elemendi.
- Lisame pinusse paari $(x_0, ax + by = c)$, kus x_0 on sirgete s ja $ax + by = c$ lõikepunkti x-koordinaat.

Analoogse protsessi teeme läbi ka nende sirgete jaoks, mille vektor on suunatud alla ($b < 0$). Lisaks saame sirgetest, kus $b = 0$ teada x minimaalse ja maksimaalse väärtuse $x_{\min}$ ja $x_{\max}$.



Joonis 5: Ühisosa leidmine

Nüüd peame kontrollima, kas lõigus $[x_{\min}, x_{\max}]$ leidub punkt, kus kujundit ülevalt piirav sirglõikude kogum on rangelt kõrgemal, kui kujundit alt piirav sirglõikude kogum. Selleks piisab, kui kontrollida iga vahemikku $[x_{\min}, x_{\max}]$ jääva punkti jaoks ja punktide $x_{\min}$, $x_{\max}$ endi jaoks, kas ülevalt piirav sirglõikude kogum on rangelt kõrgemal alumisest.

Selleks, et arvutada, kus asub ülemine (või alumine) piirjoon antud x-koordinaadi korral, võib kasutada kahendotsingut.

Lineaarplaneerimisest

See osa on kõik mõeldud huvi korral lisamaterjalina.

Üldisemalt nimetatakse lineaarvõrratuste süsteemi lahendamist *lineaarplaneerimiseks* (ingl *linear programming*). Üldjuhul vaadeldakse ülesannet kujul

$$\begin{array}{lrl} \text{maksimeerida} & c_1x_1 + c_2x_2 + \cdots + c_nx_n & \\ \text{tingimustel} & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & \leqslant b_1 \\ & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & \leqslant b_2 \\ & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n & \leqslant b_m, \end{array}$$

kus a_{ij}, b_i ja c_j on teada ning x_j tundmatud. Võrratuste suunad võib muidugi ümber vahetada, samuti maksimeerimise minimeerimise vastu. Võrratuste süsteemi lahendite hulk on alati kas tühihulk või mõni polüeeder. Maksimum saavutatakse alati kas polüeedri tipus või lõpmatuses.

Lineaarplaanide lahendamiseks on teada efektiivseid algoritme (simpleksmeetod, ellipsoidmeetod, sisepunktimeetodid (ingl vastavalt *simplex algorithm*, *ellipsoid algorithm*, *interior-point methods*), kusjuures neist esimene on kiire ainult praktikas ja teine ainult teoorias.

Meil on aga ülesanne leida suvaline punkt $(\lambda_1, \lambda_2, \lambda_3)$, mis rahuldaks võrratuste süsteemi

$$
\begin{aligned}
a_1\lambda_1 + b_1\lambda_2 + c_1\lambda_3 &> 0 \\
a_2\lambda_2 + b_2\lambda_2 + c_2\lambda_3 &> 0 \\
\dots\dots\dots\dots\dots\dots & \\
a_m\lambda_m + b_m\lambda_2 + c_m\lambda_3 &> 0 \\
\lambda_1 &\geqslant 0 \\
\lambda_2 &\geqslant 0 \\
\lambda_3 &\geqslant 0,
\end{aligned}
$$

kus a_i, b_i, c_i on tuletatud nõuetest nagu ülal näidatud. Midagi maksimeerida ega minimeerida pole vaja.

See ei ole päris sama, kuna kasutame rangeid võrratusi. See on oluline: kui lubada mitterangeid võrratusi, oleks igas testis $(0, 0, 0)$ aktsepteeritav lahend. Võtame kasutusele täiendava muutuja ε, et sundida võrratusi rangeteks. Saame lineaarplaani

$$
\begin{aligned}
\text{maksimeerida} \quad & & \varepsilon & \\
\text{tingimustel} \quad & & a_1\lambda_1 + b_1\lambda_2 + c_1\lambda_3 &\geqslant \varepsilon \\
& & a_2\lambda_2 + b_2\lambda_2 + c_2\lambda_3 &\geqslant \varepsilon \\
& & \dots\dots\dots\dots\dots\dots & \\
& & a_m\lambda_m + b_m\lambda_2 + c_m\lambda_3 &\geqslant \varepsilon \\
& & \lambda_1 &\geqslant 0 \\
& & \lambda_2 &\geqslant 0 \\
& & \lambda_3 &\geqslant 0,
\end{aligned}
\tag{2}
$$

Kui lineaarplaanil (2) leidub lahend, kus $\varepsilon > 0$, siis vastame “jah”; vastasel juhul “ei”. Nii saaks põhimõtteliselt ülesande ära lahendada, kui kuskilt leida kasutusvalmis LP lahendaja. Näiteks on sellised funktsioonid olemas Pythoni teekides `ortools` ja `scipy`. Serveris aga ei ole võimalik kasutada ühtki välist teeki. Küll aga võib internetist leida LP-sid lahendavaid eraldiseisvaid programme, ja nende koodi on võimalik oma koodi sisse kopeerida.

Mitmed tuntud algoritmilised ülesanded, näiteks maksimaalse voo leidmine (ingl *maximum flow*), on lineaarplaneerimise erijuhud. Võistlustel on olnud ülesandeid, kus oodatakse lineaarplaani “otse” lahendamist, kuid sellised on väga haruldased. Lineaarplaneerimist on aga palju uuritud ja avastatud huvitavaid omadusi, näiteks duaalsus (ingl *duality*) või täiendav mitterangus (ingl *complementary slackness*). On ette tulnud ülesandeid (pigem ülikooli taseme võistlustel), kus ei tule küll lineaarplaani otse lahendada, aga kus saab ülesannet selliste omaduste kaudu analüüsida. Paar näidet (need on võrreldes Eesti olümpiaadi ülesannetega väga, **väga** rasked):

- Ülesanne H siit: `https://qoj.ac/files/ucup/statements/statements-1-10.pdf`;
- `https://atcoder.jp/contests/wtf22-day1/tasks/wtf22_day1_d`.

6. Sonic 3 & Knuckles (`sonic`)

100 punkti

Idee, teostus ja lahenduse selgitus: Tähvend Uustalu

Avatud testidega ülesandeid on informaatikavõistlustel väga erinevaid. Osad on sellised, kus andmed on elulised või žürii poolt lihtsate reeglite abil genereeritud ja kus optimaalne lahendus pole ka žüriile teada — osalejad võistlevad selle peale, kes kõige parema lahenduse leiab. Sellised ülesanded on EIO lahtisel võistlusel olnud näiteks `ruudustik` aastal 2022 ja `moo` aastal 2018. Osad on aga sellised, kus igas testis on mingi spetsiifiline muster või konstruktsioon ja žüriil on iga testi jaoks olemas lahendusidee, mis on just selle testi pihta suunatud. Selline ülesanne on EIO lahtisel olnud näiteks `linn` aastal 2018. Eksperimendi korras on koostatud ka igasugu muid ülesandeid.

Antud ülesanne kuulub teise kategooriasse. Iga testi tuleb vaadelda sisuliselt eraldi ülesandena ja potentsiaalselt lausa iga testi jaoks eraldi programm kirjutada.

- Testid 0 ja 1 on piisavalt väikesed, et need peaksid olema enam-vähem käsitsi tehtavad.
- Testid 2 kuni 4 on sellised, et "peale vaadates" peaks olema selge, mida on vaja teha, aga testid on vastuse käsitsi tippimise jaoks liiga suured ja mõistlik oleks kirjutada programm.
- Testides 5 kuni 9 on simuleeritud erinevaid (võrdlemisi tuntud) graafiülesandeid. Siin ei ole õige teekonna leidmise loogika testile peale vaadates ilmne.

Ruudustikega opereerimine on üsna vastik. On aga mõned trikid ja meetodid, millega seda talutavaks teha. Sellepärast ongi hea pikal võistlusel läbi mõelda, kuidas ruudustikuga mingit operatsiooni teha; siis ei pea lühikesel võistlusel selle peale aega kulutama.

Vaatleme näiteks, kuidas leida ruudustikul lühimat teed punktist A punkti B. Seda on ülesande lahendamise ajal mõnel pool vaja: näiteks testis 6 tuleb liikuda ühe sinise palli juurest teise juurde, samas võib tee peal ees olla valgeid palle. Kõige lihtlabasem laiuti läbimine näeb välja selline:

```
vector<vector<int>> dist (n, vector<int> (m, INF));
// dist[x][y] on punkti (x, y) kaugus alguspunktist
// esialgu igal pool lõpmatus

vector<vector<pair<int, int>>> last (n, vector<pair<int, int>> (m));
// last[x][y] on punkt, mis lühimal teekonnal alguspunktist
// punkti (x, y) punktile (x, y) vahetult eelneb.

queue<pair<int, int>> Q;
Q.emplace(start_x, start_y);
dist[start_x, start_y] = 0;

while (!Q.empty()) {
  auto cur = Q.front(); Q.pop();

  // grid[x][y] tähistab ruudus (x, y) olevat sümbolit
  // ülemine naaber
  if (cur.first != 0 &&
      grid[cur.first - 1][cur.second] == '.' &&
      dist[cur.first - 1][cur.second] == INF) {
    dist[cur.first - 1][cur.second] = dist[cur.first][cur.second] + 1;
    last[cur.first - 1][cur.second] = cur;
    Q.emplace(cur.first - 1, cur.second);
  }
```

```
        // vasakpoolne naaber
        if (cur.second != 0 &&
            grid[cur.first][cur.second - 1] == '.' &&
            dist[cur.first][cur.second - 1] == INF) {
          dist[cur.first][cur.second - 1] = dist[cur.first][cur.second] + 1;
          last[cur.first][cur.second - 1] = cur;
          Q.emplace(cur.first, cur.second - 1);
        }

        // alumine naaber
        if (cur.first != n - 1 &&
            grid[cur.first + 1][cur.second] == '.' &&
            dist[cur.first + 1][cur.second] == INF) {
          dist[cur.first + 1][cur.second] = dist[cur.first][cur.second] + 1;
          last[cur.first + 1][cur.second] = cur;
          Q.emplace(cur.first + 1, cur.second);
        }

        // parempoolne naaber
        if (cur.second != m - 1 &&
            grid[cur.first][cur.second + 1] == '.' &&
            dist[cur.first][cur.second + 1] == INF) {
          dist[cur.first][cur.second + 1] = dist[cur.first][cur.second] + 1;
          last[cur.first][cur.second + 1] = cur;
          Q.emplace(cur.first, cur.second + 1);
        }
      }

      // siis tuleb veel last abil teekond taastada
```

See kood on aga väga pikk ja lohisev. Siin on palju korduvat loogikat ja seega veaohtlikke situatsioone, kus kasvõi ühe sümboliga eksimine võib viia vale lahenduseni. Ja kui hakata lisama näiteks nurkapidi naabreid ka, siis läheb kood veel kaks korda pikemaks.

Esiteks proovime lahti saada sellest, et nelja naabrit koheldakse igaüht eraldi. Selleks on väga mugav trikk lisada koodi read

```
const int dx [4] = {1, 0, -1, 0};
const int dy [4] = {0, 1, 0, -1};
```

Nüüd on punkti `x, y` naabrid parajasti need, mis on kujul `x + dx[k], y + dy[k]`, kus `k` on 0, 1, 2 või 3. See võimaldab ülejäänud koodi ümber kirjutada kujule:

```
vector<vector<int>> dist (n, vector<int> (m, INF));
// dist[x][y] on punkti (x, y) kaugus alguspunktist
// esialgu igal pool lõpmatus

vector<vector<pair<int, int>>> last (n, vector<pair<int, int>> (m));
// last[x][y] on punkt, mis lühimal teekonnal alguspunktist
// punkti (x, y) punktile (x, y) vahetult eelneb.

queue<pair<int, int>> Q;
Q.emplace(start_x, start_y);
dist[start_x, start_y] = 0;

while (!Q.empty()) {
  auto cur = Q.front(); Q.pop();

  for (int k = 0; k < 4; k++) {
    int nx = cur.first + dx[k];
```

```
      int ny = cur.second + dy[k];

      if (nx < 0 || nx >= n || ny < 0 || ny >= m)
        continue;

      if (grid[nx][ny] == '.' && dist[nx][ny] == INF) {
        dist[nx][ny] = dist[cur.first][cur.second] + 1;
        last[nx][ny] = cur;
        Q.emplace(nx, ny);
      }
    }
  }

  // siis tuleb veel last abil teekond taastada
```

See ei ole antud koodi korral nii hull, aga ka ruudustikust väljumise kontroll võib olla parajalt tüütu. Mõnes ülesandes võib seda kontrolli vaja olla sagedamini ja keerulisemates kohtades. Üks võimalik trikk on ümbritseda kogu ruudustik seintega ehk lisada üles ja alla üks rida ning vasakule ja paremale üks veerg, mis koosneb ainult sümbolitest `#`. Siis ei ole seda kontrolli enam vaja, kuna laiuti läbimine külastab vaid tühje ruute ja tühi ruut ei ole siis enam kunagi ruudustiku servas.

Paaride `nx, ny` või `cur` pidev kokku- ja lahtipakkimine võib olla tüütu. Ühelt poolt tuleb paar lahti teha, et kasutada maatrikseid `dist` ja `grid`, teiselt poolt aga kokku pakkida, et seda `queue` sees kasutada. Üks võimalus on kirjutada oma klass, mis töötab nagu 2D maatriks, aga mida saab paaridega indekseerida:

```
template<typename T>
class Grid {
  vector<vector<T>> arr;

public:
  Grid (int _n, int _m, T _init) : arr (_n, vector<T> (_m, _init)) { }

  T& at (int i, int j) {
    return arr[i][j];
  }

  T& at (pair<int, int> ij) {
    return arr[ij.first][ij.second];
  }
}
```

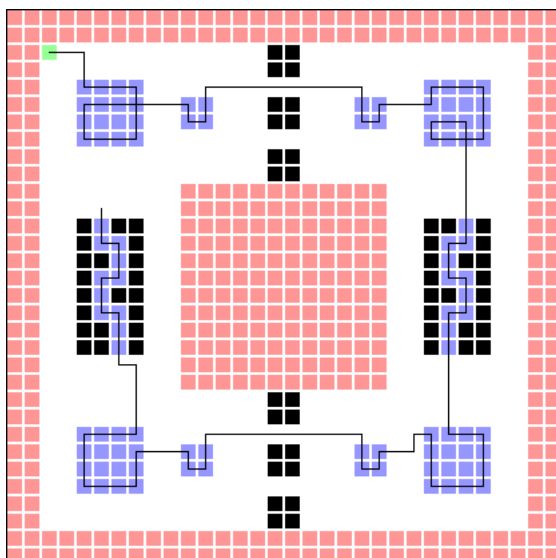
Ülejäänud kood näeb siis välja nii:

```
Grid<int> dist (n, m, INF);
Grid<pair<int, int>> last (n, m, make_pair(0, 0));

queue<pair<int, int>> Q;
Q.emplace(start_x, start_y);
dist.at(start_x, start_y) = 0;

while (!Q.empty()) {
  auto cur = Q.front(); Q.pop();

  for (int k = 0; k < 4; k++) {
    auto nxt = make_pair(cur.first + dx[k], cur.second + dy[k]);

    if (grid.at(nxt) == '.' && dist.at(nxt) == INF) {
      dist.at(nxt) = dist.at(cur) + 1;
      last.at(nxt) = cur;
```

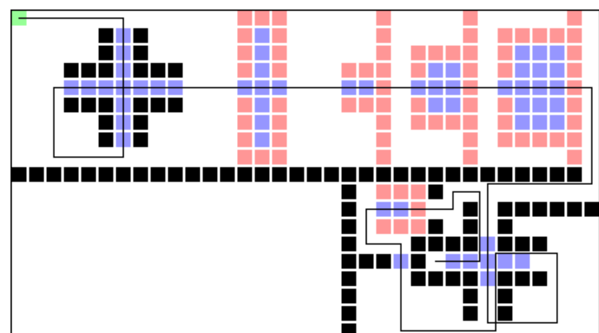
```
        Q.push(nxt);
      }
    }
  }

  vector<pair<int, int>> path;
  for (auto cur = make_pair(finish_x, finish_y);
      cur != make_pair(start_x, start_y); cur = last.at(cur)) {
    path.push_back(cur);
  }
  path.emplace_back(start_x, start_y);

  reverse(path.begin(), path.end());
```

Vaatleme nüüd ka konkreetseid teste. Kõikidel joonistel tähistab must ruut valget palli, valge ruut tühja ruutu ja roheline ruut Sonicu algpositsiooni. Sinine ja punane ruut tähistab vastava värviga palli.

Testid 0 ja 1: käsitsi tehtavad



(a) Test 0 koos võimaliku lahendusega



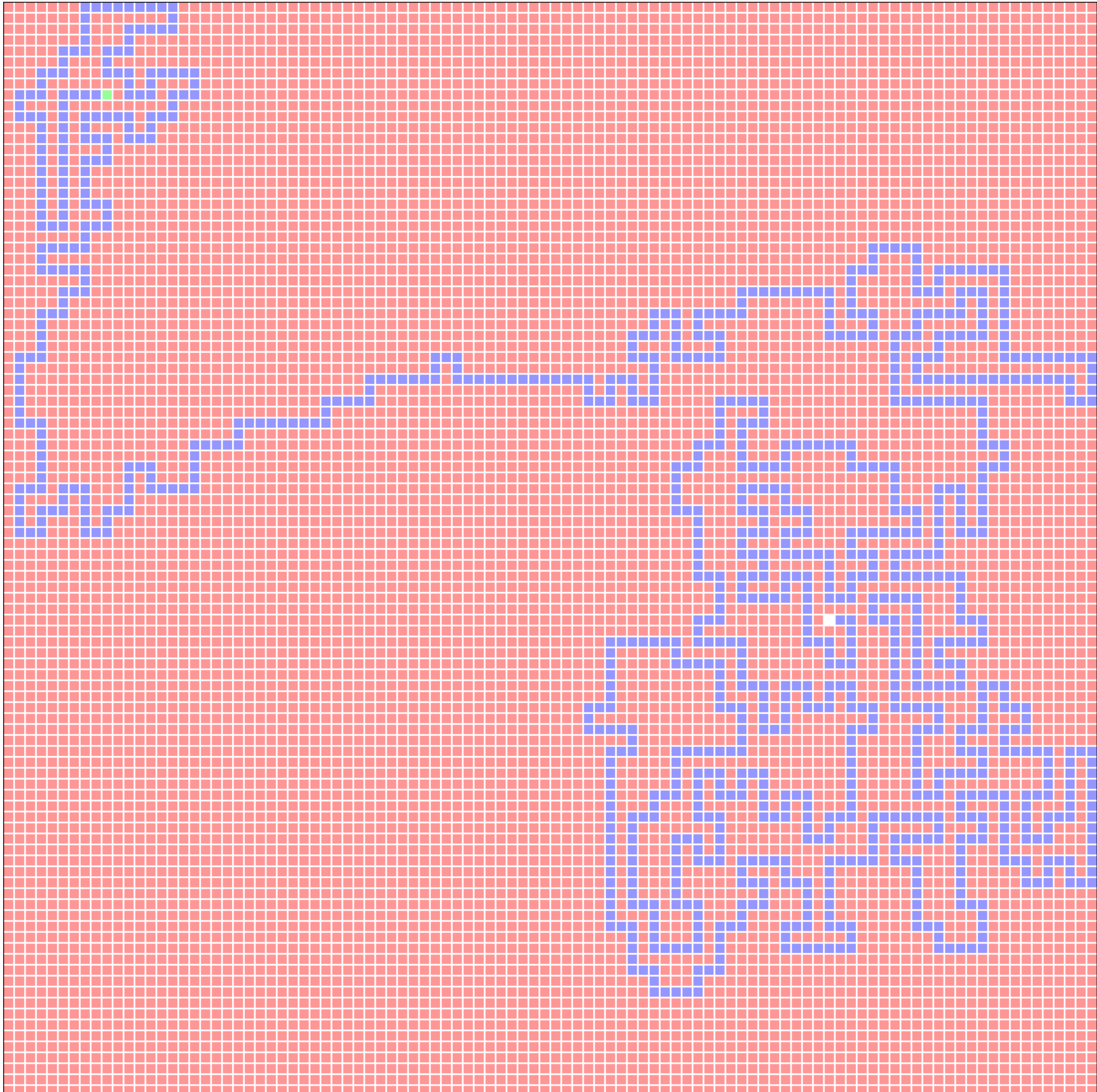
(b) Test 1 koos võimaliku lahendusega

Test 0 pärineb mängust endast — tegemist on sealse kõige esimese lisatasemega. Test 1 koosneb mõnest konstruktsioonist, mis tundusid huvitavad. Need testid on piisavalt väikesed, et neid peaks olema võimalik ilma suurema vaevata käsitsi lahendada. Samas on väljundid ikkagi 100–200 sümboli pikkused ja nende päris käsitsi väljundfaili tippimine üsna vaevarikas ja veaohtlik töö. Seega ka siin tuleb programmeerimine kasuks.

Läheneda võib näiteks nii: kirjuta "interaktiivne" programm, mis:

1. Trükib välja mänguseisu.
2. Loeb sisendist ühe sümboli.
3. Liigub ühe sammu vastavas suunas ja jätkab punktist 1.

Nii saab ülesannet ühe sammu kaupa käsitsi lahendada ja samal ajal mänguseisu jälgida. Selleks on väga hea kasutada ülesandega kaasa antud faili `checker.cpp`: jätta `Game` klass nii nagu on



Joonis 6: Test 2

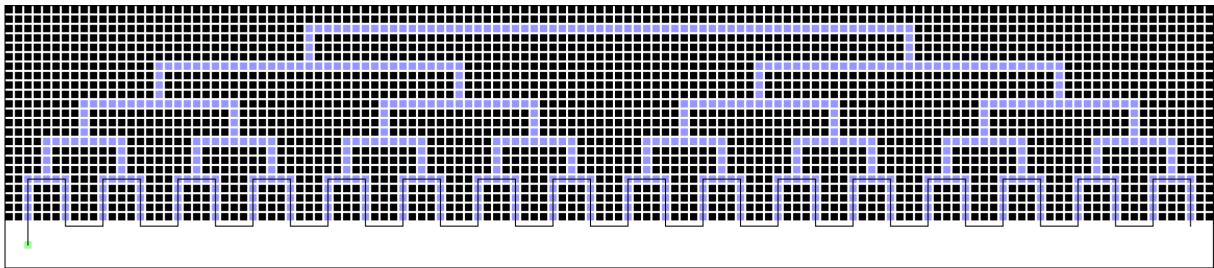
ja muuta `main`-funktsiooni nii, et sellega saab mängu interaktiivselt mängida. Näide sellisest programmist on lahenduste ja testide pakis failis `sonic/solution/tracer.cpp`.

Test 2: teekonna järgimine

See test koosneb ühest teekonnast, mis iseendaga ei lõiku ja mis on igalt poolt punaste pallidega ümbritsetud (v.a. otstest, et vältida seda, et kõik punased pallid "iseenesest" kaovad). Nüüd on aga test juba piisavalt suur, et selle käsitsi lahendamine oleks paras kilplase töö.

Lahendus võiks siis olla midagi sellist:

- Nii kaua, kuni test pole lahendatud:
  - Leiame naaberruudu, kus on sinine pall või mis on tühi ja kus me pole veel käinud.



Joonis 7: Test 3 koos võimaliku lahendusega

Selliseid naabreid on ainult üks.
- Liigume sellele naaberruudule.

Test 3: kahendpuu I

See on esimene kolmest kahendpuuga testist.

Selles testis piisab käia läbi kõik kõrvutiolevate lehtede paarid nagu näidatud joonisel 7. Niimoodi on ka puu ülemine osa igalt poolt punaste ja valgete pallidega ümbritsetud ja test on lahendatud.

Test 4: kastid

Selles testis on ruudustik valgete pallidega "kastideks" jaotatud. Igas kastis on keskel "ülesanne" ja iga kasti servas on 0, 1 või 2 sinist palli, mille kaudu teise kasti pääseb. Saame kasutada sarnast lahendust nagu testis 2:

- Nii kaua, kuni test pole lahendatud:
  - Lahendame kasti keskel oleva "ülesande". Neid on kokku viis erinevat, piisab programmi iga ülesande lahendus "sisse kirjutada".
  - Leiame kasti servas oleva sinise palli, mida me pole veel külastanud. Selliseid saab olla vaid üks.
  - Liigume läbi sinise palli vastavasse naaberkasti.

Test 5: kahendpuu II

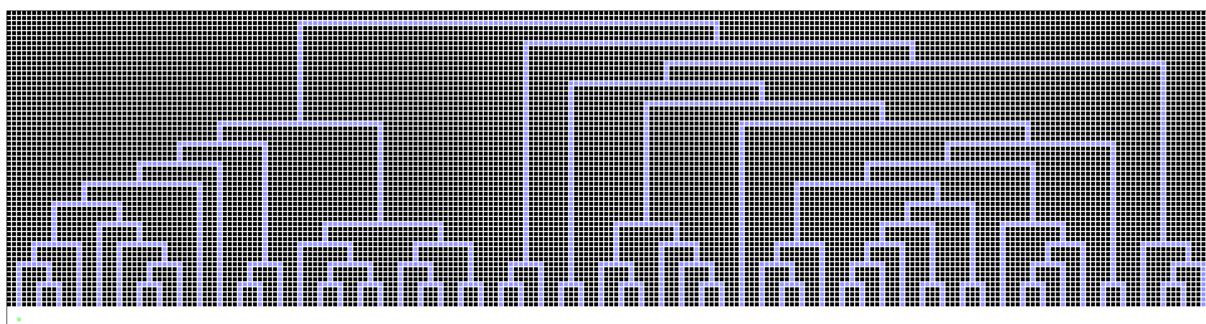
Selles testis on meil jälle kahendpuu, aga nüüd testi 3 lähenemine enam ei tööta, kuna puu ei ole enam nii regulaarne. Meie teekond peab kindlasti külastama puu kõiki lehti, sest lehti ei ole võimalik punaste pallidega ümbritseda. Lisaks nõuame, et külastame puu iga serva vaid ühe korra — nii võime kindlad olla, et me ei astu kunagi ühegi punase palli peale.

Teisisõnu on meil antud puu n lehega. Peame leidma $n/2$ teekonda, igaüks ühest lehest mingisse teise lehte nii, et ühtki serva ei külastata mitu korda. Võtame suvalise tipu (näiteks ruudustikus kõige kõrgema) puu juurtipuks. Läbime puu sügavuti; kui tipu u alampuus on paarisarv lehti, siis kustutame serva tipu u ja tema vanema p vahel. See protsess on illustreeritud joonisel 10.

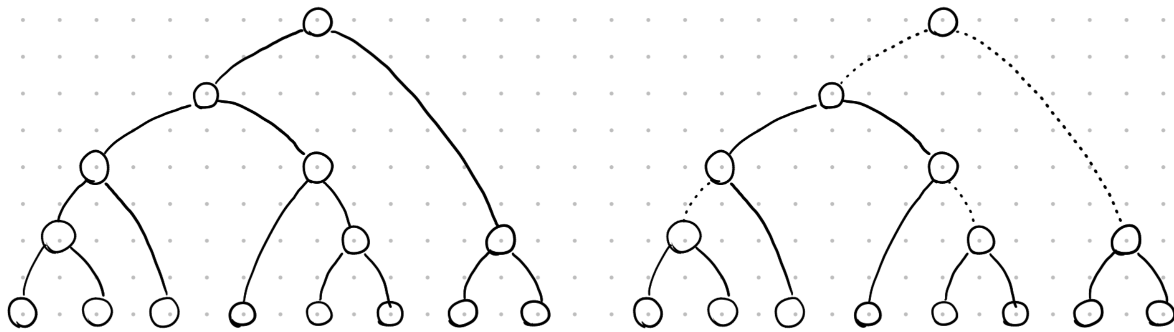
Paneme tähele, et saadud graafis on iga tipu aste 2 või 0, välja arvatud lehed ja võib-olla juurtipp. Tõepoolest: olgu tipu u lapsed l ja r ning vanem p.

- Kui nii l kui r alampuus on paarisarv lehti, siis on ka tipu u alampuus paarisarv lehti. Eemaldatakse servad lu, ru ja up. Tipu u aste on 0.

Joonis 8: Test 4



Joonis 9: Test 5



Joonis 10: Servade eemaldamine

- Kui l alampuus on paaritu arv lehti ja r alampuus paarisarv (või vastupidi), siis on u alampuus paaritu arv lehti. Eemaldatakse serv ru, alles jäävad servad lu ja up. Tipu u aste on 2.
- Kui nii l kui r alampuus on paaritu arv lehti, siis on tipu u alampuus paarisarv lehti. Eemaldatakse serv up, alles jäävad servad lu ja ru. Tipu u aste on 2.

See tähendab, et graaf koosneb hulgast ahelatest. See määrab ära $n/2$ teekonda, mida otsisime.

Test 6: Euleri ahel

Selles testis on ruudustik valgete pallide abil “ruumideks” jagatud. Mõned ruumid on joonisel 11 taustavärviga eristatud. Erinevalt testist 4 ei ole ruumid kõik ruudukujulised, vaid irregulaarsete kujudega. Naaberruumide vahel võib olla üks või mitu sinise palli abil tehtud “ust”. Kuna neid siniseid palle ei ole võimalik punaste pallidega ümbritseda, siis teame, et peame igast uksest täpselt ühe korra läbi minema.

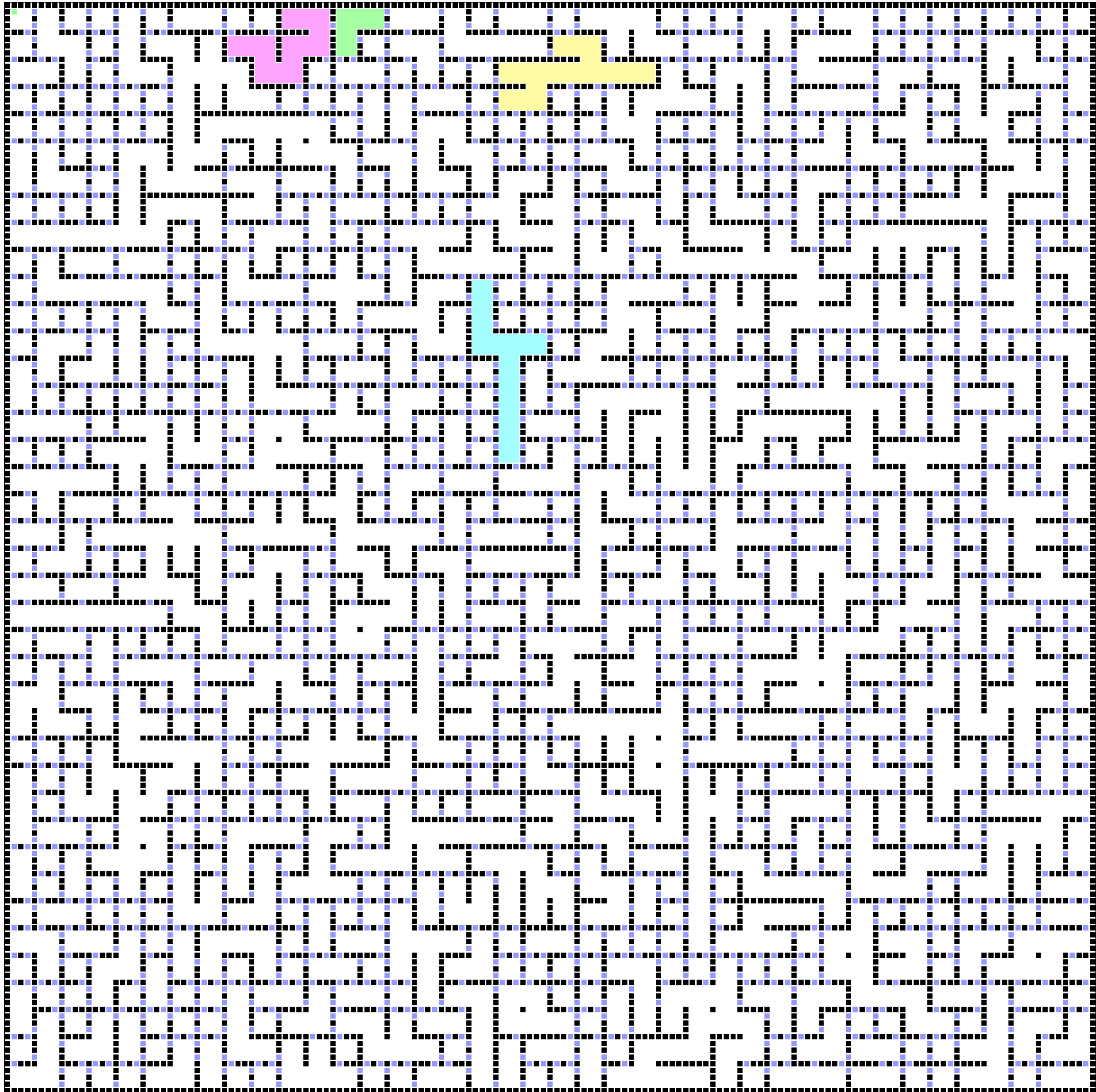
Modelleerime olukorda graafina. Iga ruumi kohta teeme ühe tipu ja iga ukse kohta joonistame serva vastavate tippude vahele. Nüüd on ülesandeks leida teekond, mis alustab etteantud tipust ja käib läbi kõik servad.

See on üsna tuntud ülesanne, mida tuntakse *Euleri ahela leidmise ülesande* (*Eulerian path*) nime all. Lahend leidub ainult siis, kui igast tipust (välja arvatud ehk algtipp ja lõpptipp) väljub paarisarv servi. Tõepoolest — kontroll näitab, et kõik ruumid peale alguse ja ühe veel piirnevad paarisarvu ustega.

Test 7: Hamiltoni tee I

Selles testis on ruudustikus 24 sinise palliga tähistatud “ristumiskohta”. Ristumiskohad on omavahel ühendatud ühe ruudu laiusega teekondade abil. Kuna teekonnad on ühe ruudu laiused, siis ei ole võimalik teekonnal ümber pöörata: kui otsustame ühest ristumiskohast minna mingis suunas, siis peame liikuma selles suunas nii kaua, kuni jõuame järgmise ristumiskohani. Sinine pall ristumiskohal tähendab, et igas ristumiskohas saame käia vaid ühe korra.

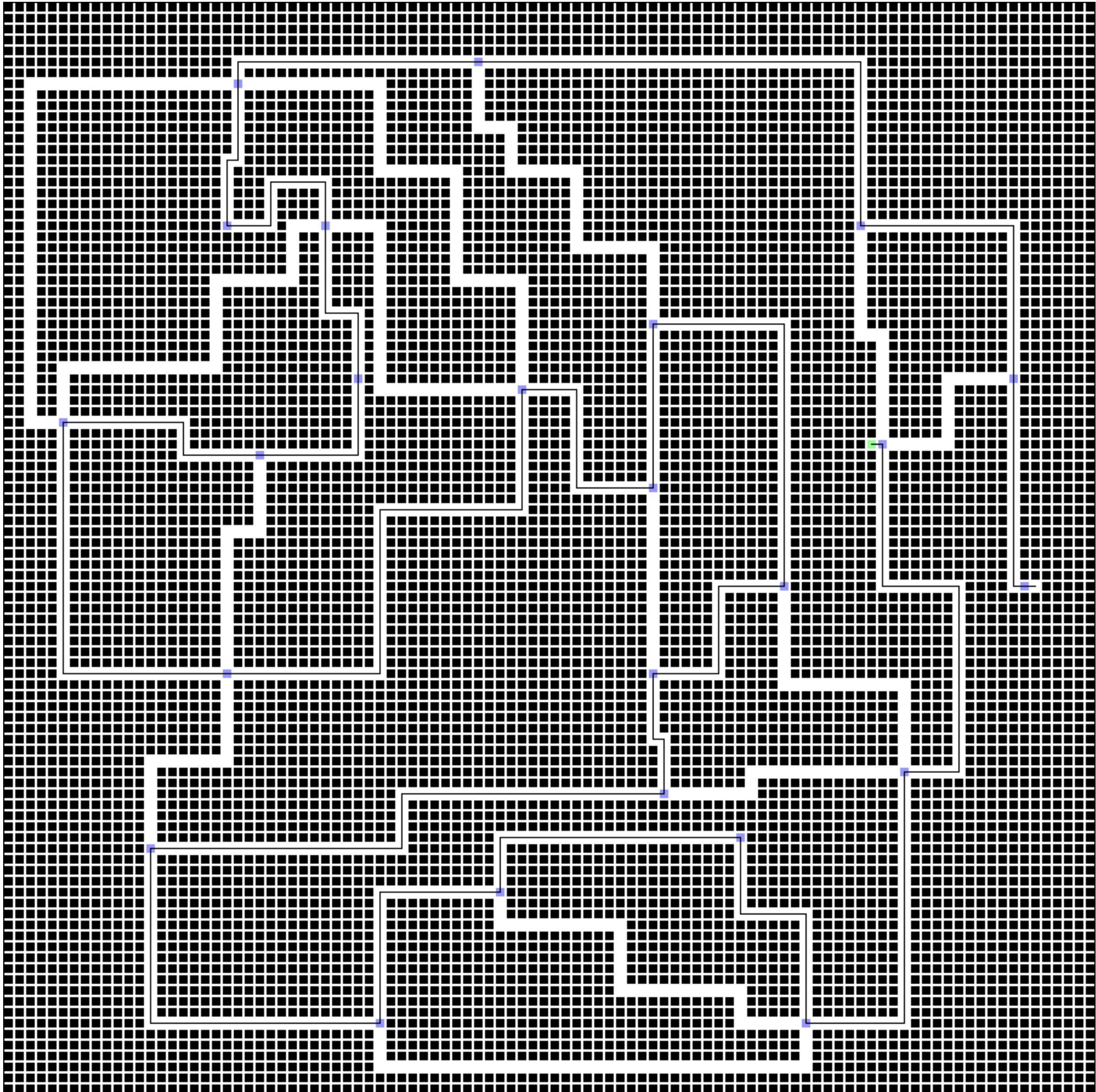
Taas on kasulik mõelda graafiteooriale. Joonistame tipu iga ristumiskoha kohta ja kahe tipu vahele serva, kui nende vahel on tühjadest ruutudest moodustatud teekond. Meil on antud algustipp; peame graafis leidma teekonna, mis külastab iga tippu täpselt ühe korra. See on jälle tuntud ülesanne, mida tuntakse *Hamiltoni tee leidmise ülesande* (*Hamiltonian path*) nime all.



Joonis 11: Test 6

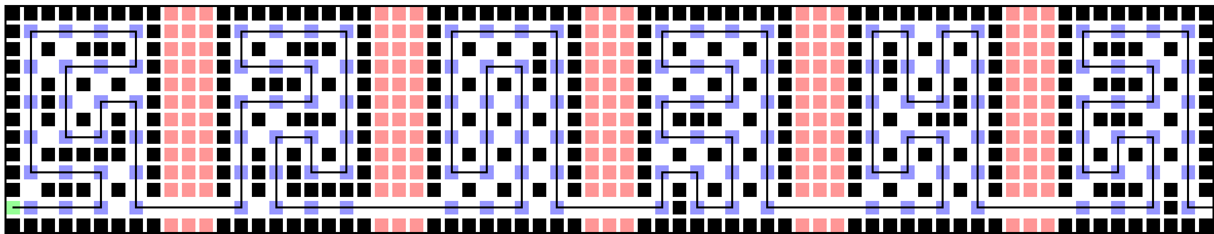
Antud ülesandes on Hamiltoni tee leidmiseks piisavalt kiire juhtude läbivaatus, mida võib implementeerida rekursiooniga näiteks nii:

```
// igale tipule on määratud numbriline indeks 0...n-1
bool dfs (int cur, // praeguse tipu indeks
          vector<int> &path, // praegu vaatluse all olev teekond
          vector<bool> &visited, // visited[i] == kas tipus i juba käidud
          const Graph &graph) {
  if (path.size() == graph.n) {
    return true; // teekond leitud
  }

  for (int nxt : G.adj[cur]) { // G.adj[cur] on tipu cur naabrite loetelu
    if (visited[nxt]) {
      continue; // külastatud tippe teekonna otsa panna ei proovi
    }
```

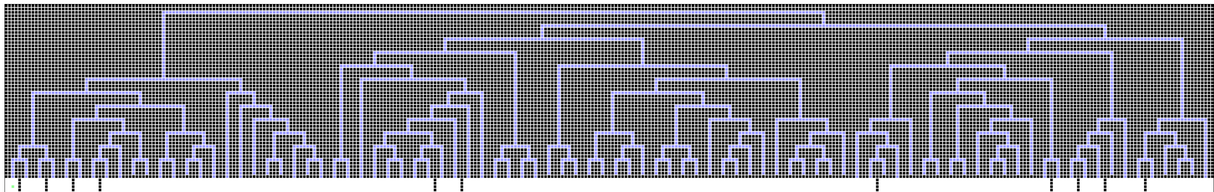


Joonis 12: Test 7 koos võimaliku lahendusega

```
    // lisame teekonna otsa tipu nxt
    visited[nxt] = true;
    path.push_back(nxt);

    bool ret = dfs(nxt, path, visited, graph);
    if (ret) {
      // väljume kohe, et vektorisse path jääks alles kogu teekond
      return true;
    }

    // teekonda ei leitud. tühistame tipu nxt teekonna otsa panemise
    visited[nxt] = false;
    path.pop_back();
  }
```



Joonis 13: Lõik testist 8, pööratud 90 kraadi, koos võimaliku lahendusega



Joonis 14: Test 9

```
        return false;
    }
```

Hamiltoni tee leidmiseks on olemas ka efektiivsem dünaamilisel plaanimisel põhinev algoritm, mille keerukuseks on $O(2^n n^2)$. Rekursiooniga Hamiltoni tee leidmine on üldjuhul palju aeglasem, keerukusega $O(n!)$.

Antud juhul on aga igal tipul ülimalt neli naabrit, seega peame igas sammus läbi vaatama ülimalt kolm varianti. See annab keerukuseks $O(3^n)$. Teame, et $3^n \approx 2824 \cdot 10^8$. Kui arvestame, et iga variandi läbi vaatamiseks kulub 10^{-8} sekundit, siis jookseb programm halvimast halvimal juhul mõnikümmend minutit. Tegelikult on programm aga palju kiirem (alla sekundi!), sest väga sageli satub see üsna ruttu tupikusse ja suur hulk variante välistatakse palju varem.

Sel põhjusel ei olnud ka kahjuks või õnneks võimalik luua testi, mis sunniks osalejaid kasutama kiiremat, $O(2^n n^2)$ algoritmi.

Test 8: Hamiltoni tee II

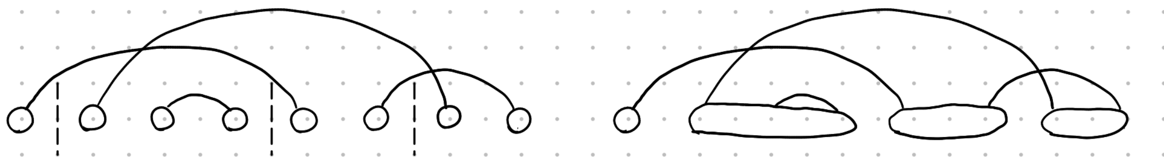
See võib teekondade lühiduse tõttu esialgu tähelepanuta jääda, aga selles testis on tegu täpselt sama situatsiooniga, mis eelmises testis. Nüüd on aga ristumiskohti 720 tükki ja variantide läbivaatus enam ei aita. Ei piisa ka dünaamilisest plaanimisest.

Küll aga võime tähele panna, et test on jaotatud punastest pallidest moodustatud "barjääridega" sisuliselt sõltumatuteks juppideks, milles igaühes on 24 ristumiskohta. Võime lahendada iga jupi eraldi, kasutades sama algoritmi, mis eelmises testis.

Test 9: kahendpuu III

See test sarnaneb testile 5, aga lisandunud on kaks komplikatsiooni:

- Osad puu lehed on "blokeeritud".



Joonis 15: Taandamine Euleri ahelale

- Testi alumisse ossa on tekkinud seinad nii, et ei ole võimalik enam liikuda suvalisest lehest suvalisse teise lehte.

Blokeeritud lehtede osas piisab, kui lihtsalt kustutame puust kõik harud, mis on täielikult blokeeritud. Nüüd oleme (peale seinte) samas olukorras, mis testis 5.

Teeme nüüd puu peal sama transformatsiooni, mis test 5. See jaotab lehed paaridesse $(u_1, v_1), (u_2, v_2), \ldots$, mis tähendavad, et peame läbima täpselt ühe korra teekonna $u_1 \rightsquigarrow v_1$, täpselt ühe korra teekonna $u_2 \rightsquigarrow v_2$ ja nii edasi. See on omaette graaf, mille tipud on kõik algse puu lehed ja servad on u_1v_1, u_2v_2 jne. Surume selles graafis kaks tippu kokku, kui need ei ole omavahel seinaga eraldatud, sest kui kaks tippu ei ole seinaga eraldatud, siis saab ühest lehest vabalt teise liikuda. Nüüd on meil graaf, kus peame leidma teekonna, mis läbib iga serva täpselt ühe korra. See on juba varasemast tuttav Euleri ahela ülesanne.

7. Poolik tekst (`poolik`)

100 punkti

Idee: Sandra Schumann, teostus: Sandra Schumann ja Ahto Truu, lahenduse selgitus: P. Randla ja Ahto Truu

Testides kasutatud tekstide iseloomustused:

000. Arvatavasti kõigile tuttav raamatu "Kevade" esimene lause UTF-8 kodeeringus. Silma järgi äraarvatav.
001. 1988. aasta informaatikaolümpiaadi lõppvooru ülesannete tekstid ISO-Latin-9 kodeeringus.
002. "Manifest kõigile Eestimaa rahvastele" DOS Baltic kodeeringus. Tekst tuntud, kuid kodeeringuga peab hoolikas olema.
003. Eestikeelne tekst UTF-8 kodeeringus. Tekst genereeritud osaliselt ChatGPT abiga, ei ole internetist leitav.
004. Inglisekeelne pikem tekst, ASCII. Genereeritud ChatGPT abil, ei ole internetist leitav.
005. Vanem Edda, UTF-8.
006. Umbes 40 rida Pythoni koodi, ASCII.
007. Umbes 40 rida C++ koodi, ASCII.
008. Umbes 60 rida assemblerkeeles koodi, esimese ülesande lahendus, ASCII.
009. Umbes 25 rida COBOL-keeles koodi, ASCII.

Üldiselt võiks selle ülesande lahendamiseks kasutada kombinatsiooni programmidest, mis sõnaraamatute abiga potentsiaalseid sõnu pakuvad, teadmisi arvutiteaduse valdkonnast ja oskust internetist tuntud tekste leida.

Mistahes lahenduse esimese sammuna peaks sisendis taastama sõnade piirid, teades et sümbolid on mõlemas tekstis samadel kohtadel ja seega vastavate sümbolite vahel olevad tähed on sama sõna omad. Siis jääb põhiülesandeks kahest tähejadast taastada esialgne sõna.

Üks võrdlemisi naiivne, kuid see-eest väga üldine viis seda teha on panna igas sõnas kaas- ja täishäälikud vaheldumisi. Siis suure tõenäosusega vähemalt osad tähed langevad kokku, lisaks klapivad kõik sümbolid, seega kokku on kattuvate baitide hulk üsna suur ja võimalik juba ligikaudu 30 punkti kätte saada. See on implementeeritud lahenduses `sol_lihtne.py`.

Sellest parem lahendus on kasutada sõnaraamatut: iga sõna jaoks otsime sõnaraamatust kõik sõnad, mis annavad täis- ja kaashäälikute eraldamisel sisendis olevad tähejadad. Kui neid ei leidu või on mitu, siis küsime kasutajalt, millist sõna kasutada (ja jätame vastuse meelde, kuna samas tekstis esineb sama sõna tõenäoliselt veel paar korda). Lisaks võib sõnaraamatust otsimise tarbeks kõik tähed alguses väikesteks teha ja hiljem, kui õige sõna leitud, vajalikes kohtades suurtähed taastada.

Eestikeelsetes testides on võimalik sõnastikuna kasutada EKI veebilehelt faili `soned2013.txt` ja sealt harvemini esinevad sõnad välja võtta.

Pikas ingliskeelses testis sobib sõnastikuks paljudes Linuxi distributsioonides standardselt olemas olev `/usr/share/dict/words`. Windowsis sellist nimekirja vaikimisi pole ja tuleb internetist otsida.

Vanapõhja keele sõnastikku võib olla keeruline leida. See-eest on antud testi tekst üks kuulsamaid vanapõhjakeelseid tekste ja selle täistekst on internetist kergesti leitav.

Pythoni ja C++ programmides on kõige mõistlikum iga sõna käsitsi ära parandada (vastuseid meelde jättes tuleb seda iga unikaalse sõna jaoks teha vaid üks kord). Seal on palju sõnu, mida

tavalistes sõnaraamatutes tõenäoliselt ei leidu, ja vähemalt ühe levinud sõnaraamatuga tekib Pythoni koodis valepositiiv (muutujanimena on kasutatud `idx`; sõnaraamatus seda ei ole, aga on nimi "Dix").

Assemblerprogrammis jaoks võivad vilunumad käsitsi kõik sõnad ära parandada, aga võib ka internetist otsida GNU assembleri süntaksi värvimise faili, sealt kõik lubatud assemblerkeele käsud eraldada ja siis seda sõnaraamatuna kasutada.

COBOLi koodi jaoks sobib kasutada ingliskeelset sõnastikku, kuna kõik käsud on korrektsed ingliskeelsed sõnad.

Kõik eelmainitud meetodid on implementeeritud lahenduses `sol_sonastik.py`.

Kodeeringutest

Nagu teada, töötlevad digitaalarvutid infot alati arvudena. See tähendab, et ka tekste esitatakse arvuti mälus arvudena, tavaliselt nii, et teksti igale märgile (tähele, numbrile, kirjavahemärgile, tühikule jne) vastab mingi arv ja teksti esitatakse nende arvude jadana. Sellist arvude ja märkide vastavuse kokkulepet nimetatakse kooditabeliks. Ajalooliselt tähtis kooditabel on ASCII (ingl *American Standard Code for Information Interchange*), milles näiteks 'A'-tähe kood on 65 ja 'a'-tähe kood 97. See kooditabel kasutab 7-bitiseid koode (mida on kokku 128) ja sisaldab ainult inglise tähestiku tähti koos numbrite, valiku kirjavahemärkide ja mõnede vanades telegraafisüsteemides kasutusel olnud juhtsümbolitega. Suurem osa programmikeeli lubab ka tänapäeval programmi tekstis kasutada ainult selles kooditabelis defineeritud märke.

Digitaalse andmetöötluse levimisel laiendati ASCII kooditabel 8-bitiseks, lisades sinna ka teiste keelte jaoks vajalikke tähti. Võimalikke 8-bitiseid koode on kokku 256, millest ei jätku kõigi keelte esitamiseks, ja nii loodi erinevatele keeltele erinevaid kooditabeleid. Üks levinud laienduste pere on ISO-8859 standardite seeria, millesse kuulub ka ISO-8859-1 ehk nn ISO-Latin-1 kooditabel. See sisaldab üldiselt Lääne-Euroopa keelte diakriitikutega tähti ja nende hulgas ka suuremat osa eesti keele jaoks vajalikku. Näiteks on selles tabelis 'Õ'-tähe kood 213 ja 'õ'-tähe kood 245, aga 'Š'- ja 'š'-tähti seal ei ole. ISO-8859 standardipere hilisem liige ISO-8859-15 ehk nn ISO-Latin-9 kooditabel asendas mõned vähem kasutust leidnud märgid eesti keelele vajalikega, näiteks 'Š'-täht sai koodi 166 (mis ISO-Latin-1 kooditabelis tähistab katkestusega püstkriipsu '¦') ja 'š'-täht koodi 168 (mis ISO-Latin-1 kooditabelis tähistab täppe '¨'). Selle ülesande teises testis kasutusel olevate tähtede osas langevad ISO-8859-1 ja ISO-8859-15 kooditabelid kokku nii omavahel kui ka Windowsi 1252 (nn Windows Western) ja 1257 (nn Windows Baltic) kooditabelitega ning selle faili avamisel ükskõik millist neist kooditabelitest kasutava programmiga näidatakse kõiki seal olevaid tähti õigesti.

Mõni aasta enne ISO-8859-15 standardi ilmumist defineerisid IBM ja Microsoft operatsioonisüsteemides PC-DOS ja MS-DOS kasutamiseks kooditabeli, mida tuntakse IBM775, CP775 või DOS Baltic nime all. Ka see sisaldab Läänemere-äärsete maade keelte jaoks vajalikke tähti, kuid ISO-8859 standardiperest hoopis erinevate koodidega. Näiteks on selles tabelis 'Õ'-tähe kood 229 ja 'õ'-tähe oma 228. See tähendab, et kui DOS Baltic kodeeringus tekstifail avada programmiga, mis eeldab näiteks ISO-8859-1 kodeeringut, paistab seal 'Õ' asemel 'å' ja 'õ' asemel 'ä'.

Ühelt poolt on mitmetes maailma keeltes (näiteks hiina ja jaapani keeles) kasutusel märksa rohkem märke kui ka 8-bitiste koodidega esitada saab. Teiselt poolt võib ka nende keelte puhul, mis eraldi võttes saaks 256 märgiga hakkama, olla mitmekeelsete tekstide puhul päris tülikas leida kooditabelit, mis võimaldaks esitada kõiki vajalikke märke. Sellepärast alustati juba 1980. aastate teises pooles universaalse kooditabeli Unicode defineerimisega. Algselt olid selles standardis mär-

kidel 16-bitised koodid, mida hiljem laiendati ja praeguseks defineerib Unicode koodid ligi 150 000 märgi jaoks. Veidi lihtsustatult võib öelda, et loogilisel tasemel on endiselt igale märgile seatud vastavusse üks arv, aga need arvud võivad olla suuremad kui 8-bitisesse baiti salvestada saab.

Tänapäeval kõige levinum viis Unicode koodide esitamiseks tekstifailides on nn UTF-8 (ingl *Unicode Transformation Format — 8-bit*) kodeering, mis seab erinevatele koodidele vastavusse erineva pikkusega baidijadad. Näiteks 'A'- ja 'a'-tähe koodid on Unicode kooditabelis samad, mis ASCII tabelis (vastavalt 65 ja 97), ja need salvestatakse UTF-8 kodeeringus ühebaidistena (samuti 65 ja 97). 'Õ'- ja 'õ'-tähe koodid on Unicode kooditabelis samad, mis ISO-8859 standardites (vastavalt 213 ja 245), aga UTF-8 kodeeringus salvestatakse need kahest baidist koosnevate jadadena (vastavalt 195,149 ja 195,181). 'Š'- ja 'š'-tähe koodid on vastavalt 352 ja 353 ning need salvestatakse samuti kahebaidiste jadadena, vastavalt 197,160 ja 197,161. UTF-8 koodijadad on konstrueeritud nii, et ükski lühem koodijada ei lange kokku ühegi pikema koodijada algusega (sellise omadusega kodeerimisviise nimetatakse üldiselt prefikskoodideks). See tagab, et tekstifaili järjest lugedes on alati selge, kus ühe kirjamärgi esitus lõpeb ja järgmise oma algab.

Muutuvate pikkustega koodide üks risk selle ülesande hindamisskeemi arvestades on, et kui teksti taastamisel panna mõne märgi asemele selline, mille esitus UTF-8 kodeeringus on õige märgi omast erineva pikkusega (näiteks 'a' asemele 'ä' või vastupidi), siis nihutab see kõik failis tagapool olevad baidid algsest erinevale positsioonile ja nii võib ülesannet käsitsi lahendades tehtud ühetähelise trükiveaga päris palju punkte kaotada.

Veel üks riskikoht UTF-8 kodeeringut kasutavates testides on seotud Unicode ajalooga. Nimelt kasutati enne UTF-8 leiutamist Unicode tekstide esitamiseks vormingut, kus iga 16-bitine kood salvestati kahebaidisena. Mõned riistvaraplatvormid salvestavad kahebaidiseid muutujaid nii, et suurema väärtusega bitid on esimeses ja väiksema väärtusega bitid teises baidis ning mõned vastupidi. Need salvestusviiside erinevused kandusid ka Unicode kooditabelit kasutavatesse tekstifailidesse ja nende eristamiseks hakati failide algusse lisama vastavalt baidipaare 254,255 või 255,254. Kuigi UTF-8 kodeeringus on baitide järjekord alati üheselt määratud, võivad mõned tekstitöötlusprogrammid ettevaatamatul kasutamisel lisada ka UTF-8 faili algusse baidijada 239,187,191. Muidugi nihutavad sellised lisabaidid kõik järgnevad baidid nende õigetelt kohtadelt ära ja tulemuseks on jälle suur kaotus punktides.

Tekstifailidest ja kodeeringutest rääkides peab mainima ka reavahetusi. Nimelt on Windowsi tekstifailides realõpu tunnusena kasutusel baidipaar 13,10, aga Unixi-laadsetes süsteemides (mille hulka kuuluvad ka Linux ja MacOSi uuemad versioonid) ainult bait 10. (Vahemärkusena: MacOSi vanemad versioonid aastani 2001 kasutasid ainult baiti 13.) Selleks, et lahendusena esitatud failide hindamine ei sõltuks võistleja kasutatud operatsioonisüsteemist, kustutab selle ülesande hindamisprogramm faile sisse lugedes igast 13,10 paarist baidi 13 ja jätab alles ainult baidi 10.