1. Kasutajanimed (nim)

20 punkti

Idee: Tähvend Uustalu, teostus: Indrek Jentson, lahenduse selgitus: Ahto Truu

Selles ülesandes on oluline tekstis antud lubadus, et kõik eesnimed on unikaalsed. Kui eesnimed võiks korduda, ei oleks ülesanne üldse üheselt lahenduv, sest kui meil oleks näiteks eesnimi-perenimi paarid “Mati Maasikas” ja “Mati Murakas” ning eesnimi-kasutajanimi paarid “Mati mx” ja “Mati my”, võiks “mx” olla samahästi nii Mati Maasika kui ka Mati Muraka kasutajanimi.

Selle tähelepaneku järel peaks ka esmane lahenduse idee selge olema: vaatame ühe nimekirja järjest läbi ja iga rea jaoks otsime teisest nimekirjast välja sama eesnimega rea. Need kaks rida käivad sama osaleja kohta ja järelikult moodustavad nendel ridadel olevad perenimi ja kasutajanimi ühe rea selles nimekirjas, mille me leidma ja väljastama peaks.

Naiivne lahendus on ühe nimekirja iga rea jaoks teise nimekirja kõik read järjest läbi vaadata, nagu on tehtud failides `sol0.cpp` ja `sol0.py` toodud lahendustes. Selline lahendus vaatab ühe N nimest koosneva nimekirja iga rea jaoks läbi kõik N teise nimekirja rida ja kulutab seega kokku $N \cdot N$ operatsiooni. Selle ülesande väikeste andmemahtude korral on see piisavalt efektiivne. C++ programm jõuab tänapäevastel arvutitel teha sekundiga sada kuni mõnisada miljonit ja Pythoni programm mõni kuni mõnikümmend miljonit operatsiooni. Täpsed arvud sõltuvad ka sellest, milliseid operatsioone tehakse. Näiteks täisarvude võrdlemine võtab vähem aega kui tekstide võrdlemine. Selles ülesandes lubatud maksimaalse $N = 1000$ korral kulub umbes miljon operatsiooni ja see mahub igal juhul lahedalt sekundise ajalimiidi sisse ära.

`sol0.py`

```python
import sys

# osalejate arv
n = int(sys.stdin.readline())

# eesnimi-perenimi paarid
ep = [sys.stdin.readline().split()
  for i in range(n)]

# eesnimi-kasutajanimi paarid
for i in range(n):
  e1, k1 = sys.stdin.readline().split()
  # leiame eesnimele vastava perenime
  # väljastame selle koos kasutajanimega
  for e2, p2 in ep:
    if e1 == e2:
      sys.stdout.write(
        '{} {}\n'.format(p2, k1))
```

`sol0.cpp` (osaline)

```cpp
// osalejate arv
int n; cin >> n;

// eesnimi-perenimi paarid
vector<string> en(n), pn(n);
for (int i = 0; i < n; ++i) {
  cin >> en[i] >> pn[i];
}

// eesnimi-kasutajanimi paarid
for (int i = 0; i < n; ++i) {
  string e, k; cin >> e >> k;
  // leiame eesnimele vastava perenime
  // väljastame selle koos kasutajanimega
  for (int j = 0; j < n; ++j)
    if (en[j] == e)
      cout << pn[j] << ' ' << k << '\n';
}
```

Kui tahta efektiivsemat algoritmi, mis suudaks sekundiga töödelda ka mõnekümne või mõnesaja tuhande reaga nimekirju, on selleks päris mitu võimalust. Üks võimalus on nimekirjast eesnime järgi õige rea leidmine kiiremaks teha nii, et sorteerime selle nimekirja kohe andmete sisselugemise järel ära ja siis kasutame nimekirja kõigi ridade läbivaatamise asemel kahendotsingut.

Kahendotsing alustab nimekirjas sobiva rea otsimist nimekirja keskelt. Kui keskmine element on otsitavast suurem, siis teame, et nimekirja kogu teine pool (kus on veel suurema väärtusega elemendid) on kindlasti otsitavast väärtusest suurem ning saame terve selle teise poole kohe kõrvale jätta ja otsida edasi ainult nimekirja esimesest poolest. Kui keskmine element on otsitavast

väiksem, saama sama loogika järgi jätta kohe kõrvale nimekirja esimese poole. Allesjäänud poole hulgas otsimist jätkates vaatame jälle selle keskmist elementi. Niimoodi kulub tuhande elemendi hulgast õige leidmiseks ainult 10 ja näiteks miljoni elemendi hulgast õige leidmiseks ainult 20 võrdlemist. Kahendotsingu algoritm pole väga keeruline ja seda pole kuigi raske ka ise programmeerida, aga nii C++ kui Pythoni programmeerijate jaoks on see olemas keele standardteegis ning failides `sol1.cpp` ja `sol1.py` toodud näidislahendused kasutavad seda.

`sol1.py`

```
import sys
from bisect import bisect_left

# osalejate arv
n = int(sys.stdin.readline())

# eesnimi-perenimi paarid
ep = [sys.stdin.readline().split()
  for i in range(n)]

# sorteerime eesnime järgi,
# et saaks kasutada kahendotsimist
ep.sort()

# eesnimi-kasutajanimi paarid
for i in range(n):
  e, k = sys.stdin.readline().split()
  # leiame eesnimele vastava perenime
  j = bisect_left(ep, [e])
  # väljastame selle koos kasutajanimega
  sys.stdout.write(
    '{} {}\n'.format(ep[j][1], k))
```

`sol1.cpp` (osaline)

```
// osalejate arv
int n; cin >> n;

// eesnimi-perenimi paarid
vector<pair<string, string>> ep(n);
for (int i = 0; i < n; ++i)
  cin >> ep[i].first >> ep[i].second;

// sorteerime eesnime järgi,
// et saaks kasutada kahendotsimist
sort(ep.begin(), ep.end());

// eesnimi-kasutajanimi paarid
for (int i = 0; i < n; ++i) {
  string e, k; cin >> e >> k;
  // leiame eesnimele vastava perenime
  auto j = lower_bound(ep.begin(), ep.end(),
    make_pair(e, string("")));
  // väljastame selle koos kasutajanimega
  cout << j->second << ' ' << k << '\n';
}
```

Teine võimalus eesnimi-kasutajanimi vastavuste hulgast õige leidmist kiirendada on panna need vastavused just selleks mõeldud andmestruktuuri. Sellise andmestruktuuri nimi C++ standardteegis on `map` (ingliskeelsest sõnast *mapping*, mis tähendab vastavust) ja Pythoni omas `dict` (ingliskeelsest sõnast *dictionary*, mis tähendab sõnastikku). Neid andmestruktuure kasutavad lahendused on toodud failides `sol2.cpp` ja `sol2.py`. Sarnased andmestruktuurid on olemas ka paljude teiste keelte standardteekides, mõnikord ka *associative array*, *associative container* või *hash table* nimede all.

Eelmistest natuke erinev idee on sorteerida eesnimede järgi mõlemad nimekirjad. Siis peavad nende kahe sorteeritud nimekirja vastavad read käima sama kasutaja kohta ja me võime kõik perenimi-kasutajanimi paarid leida ja väljastada nimekirju paralleelselt läbides. Nii ongi tehtud failides `sol3.cpp` ja `sol3.py` toodud lahendustes.

`sol2.py`

```
import sys

# osalejate arv
n = int(sys.stdin.readline())

# eesnimi->perenimi vastavus
ep = dict() # tühi sõnastik
for i in range(n):
  e, p = sys.stdin.readline().split()
  ep[e] = p

# eesnimi-kasutajanimi paarid
for i in range(n):
  e, k = sys.stdin.readline().split()
  # leiame ja väljastame eesnimele vastava
  # perenime koos kasutajanimega
  sys.stdout.write(
    '{} {}\n'.format(ep[e], k))
```

`sol2.cpp` (osaline)

```
// osalejate arv
int n; cin >> n;

// eesnimi->perenimi vastavus
map<string, string> ep; // tühi sõnastik
for (int i = 0; i < n; ++i) {
  string e, p; cin >> e >> p;
  ep[e] = p;
}

// eesnimi-kasutajanimi paarid
for (int i = 0; i < n; ++i) {
  string e, k; cin >> e >> k;
  // leiame ja väljastame eesnimele vastava
  // perenime koos kasutajanimega
  cout << ep[e] << ' ' << k << '\n';
}
```

`sol3.py`

```
import sys

# osalejate arv
n = int(sys.stdin.readline())

# eesnimi-perenimi paarid
# sorteeritud eesnime järgi
ep = sorted(
  [sys.stdin.readline().split()
    for i in range(n)])

# eesnimi-kasutajanimi paarid
# sorteeritud eesnime järgi
ek = sorted(
  [sys.stdin.readline().split()
    for i in range(n)])

# läbime nimekirjad paralleelselt
for i in range(n):
  # väljastame kohakuti oleva
  # perenimi-kasutajanimi paari
  sys.stdout.write(
    '{} {}\n'.format(ep[i][1], ek[i][1]))
```

`sol3.cpp` (osaline)

```
// osalejate arv
int n; cin >> n;

// eesnimi-perenimi paarid
vector<pair<string, string>> ep;
for (int i = 0; i < n; ++i) {
  string e, p; cin >> e >> p;
  ep.push_back(make_pair(e, p));
}
// sorteerime eesnime järgi
sort(ep.begin(), ep.end());

// eesnimi-kasutajanimi paarid
vector<pair<string, string>> ek;
for (int i = 0; i < n; ++i) {
  string e, k; cin >> e >> k;
  ek.push_back(make_pair(e, k));
}
// sorteerime eesnime järgi
sort(ek.begin(), ek.end());

// läbime nimekirjad paralleelselt
for (int i = 0; i < n; ++i)
  // väljastame kohakuti oleva
  // perenimi-kasutajanimi paari
  cout << ep[i].second << ' '
    << ek[i].second << '\n';
```

2. Olümpiaaditulemused (tul)

30 punkti

Idee ja teostus: Rao Zvorovski, lahenduse selgitus: Ahto Truu

Selle ülesande lahendamiseks on kõige lihtsam kasutada *assotsiatiivse massiivi* andmestruktuuri, mille nimi C++ standardteegis on `map` ja Pythoni omas `dict`. Nii saame igale kasutajanimele seada vastavusse selle kasutaja esitatud lahenduste assotsiatiivse massiivi ja igale lahendusele omakorda selle punktisumma. Kui mõni kasutaja on ühele ülesandele esianud mitu lahendust, võime kohe andmete sisselugemise käigus jätta alles ainult maksimaalse punktisumma.

C++ `map` struktuuri senitundmatu indeksi poole pöördumisel luuakse automaatselt uus element, mis saab algväärtuseks vastava andmetüübi vaikeväärtuse. Näiteks allolevas programmis loob `scores[uname]` automaatselt kasutaja `uname` jaoks tühja massiivi ja `scores[uname][tname]` seab ülesande `tname` punktisumma algväärtuseks nulli. Pythoni `dict` annaks selle asemel veateate, seega tuleb Pythoni lahenduses indeksi olemasolu kontrollida ja vajadusel element ise algväärtustada. C++ `map` struktuuri läbimisel külastame elemente nende indeksite järjekorras. Pythnoni `dict` läbimisel pole elementide järjekord määratud, mistõttu peame Pythoni lahenduses tulemused eraldi sammuna sorteerima.

`sol1.py`

```python
import sys
from operator import itemgetter

n = int(sys.stdin.readline())

# loeme sisendi
# iga kasutaja jaoks ja iga ülesande kohta
# jätame meelde ainult parima skoori
scores = dict()
for i in range(n):
  u, t, s = sys.stdin.readline().split()
  if u not in scores:
    scores[u] = dict()
  if t not in scores[u]:
    scores[u][t] = 0
  scores[u][t] = max(scores[u][t], int(s))

# käime läbi kõik kasutajad
# ja iga kasutaja kõik ülesanded
# arvutame iga kasutaja lõpptulemuse
totals = dict()
for u in scores:
  if u not in totals:
    totals[u] = 0
  for t in scores[u]:
    totals[u] += scores[u][t]

# teeme tulemuse loendiks ja sorteerime
ranking = list(totals.items())
ranking.sort(key = itemgetter(1),
  reverse = True)

# väljastame
for u, s in ranking:
  sys.stdout.write('{} {}\n'.format(u, s))
```

`sol1.cpp` (osaline)

```cpp
int n; cin >> n;

// iga kasutaja jaoks ja iga ülesande kohta
// jätame meelde ainult parima skoori
map<string, map<string, int>> scores;
for (int i = 0; i < n; ++i) {
  string uname, tname; int score;
  cin >> uname >> tname >> score;
  if (scores[uname][tname] < score)
    scores[uname][tname] = score;
}

// käime läbi kõik kasutajad
// ja iga kasutaja kõik ülesanded
// arvutame iga kasutaja lõpptulemuse
// pingerea saamiseks hoiame nimi->tulemus
// vastavuse asemel tulemus->nimed vastavust
// tulemuste kahanemise järjekorras
map<int, set<string>, greater<int>> ranking;
for (auto & i1 : scores) {
  const string & uname = i1.first;
  const auto & tasks = i1.second;
  int total = 0;
  for (auto & i2 : tasks)
    total += i2.second;
  ranking[total].insert(uname);
}

// väljastame kasutajad tulemuste kaupa
for (auto & i1 : ranking) {
  const int score = i1.first;
  const auto & names = i1.second;
  for (auto & i2 : names)
    cout << i2 << ' ' << score << '\n';
}
```

Teine võimalus (eriti neile, kes `map` ja `dict` omadusi ei tea), on lahendada ülesanne sorteerimise abil. Kolmikute (kasutaja, ülesanne, punktid) sorteerimise tulemusena saame nimekirja, mis on sorteeritud kasutaja järgi, iga kasutaja esitused omavahel ülesannete järgi ja ühe kasutaja ühe ülesande esitused omavahel punktide järgi. Sellise nimekirja ühekordse läbimisega saame koostada kõigi kasutajate punktisummade nimekirja järgmiselt:

- kui vaadeldava kolmiku kasutaja erineb eelmisest, lisame tulemuste nimekirja uue kasutaja ja tema esimese skoori;
- kui kasutaja on sama, aga ülesanne erineb, liidame viimati lisatud kasutajale uue ülesande eest saadud skoori;
- kui ka ülesanne on sama, liidame kasutajale selle ja eelmise rea skooride vahe (sest just nii palju ta oma tulemust selle esitusega parandas eelmisega võrreldes).

Kui oleme kõigi kasutajate skoorid kokku saanud, sorteerime tulemuse skooride järgi ja väljastame. Kood polegi pikem kui eelmise lahenduse oma, aga seda on natuke raskem välja mõelda ja selles on ka rohkem ohukohti, kus on võimalik teha vigu, mis igas testis ei avaldu.

`sol2.py`

```python
import sys
from operator import itemgetter

n = int(sys.stdin.readline())
submissions = [(u, t, int(s)) for u, t, s in
  (sys.stdin.readline().split() for i in
    range(n))]

# iga osaleja punktid kõigi ülesannete eest
submissions.sort()
ranking = []
ue, te, se = None, None, 0
for ui, ti, si in submissions:
  if ui != ue: # uus osaleja
    ranking.append((ui, si))
  elif ti != te: # uus ülesanne
    ur, sr = ranking[-1];
    ranking[-1] = (ur, sr + si)
  else: # sama ülesande parem tulemus
    ur, sr = ranking[-1];
    ranking[-1] = (ur, sr + si - se)
  ue, te, se = ui, ti, si

# sorteerime skooride järgi ja väljastame
ranking.sort(key = itemgetter(1),
  reverse = True)
for ur, sr in ranking:
  sys.stdout.write('{} {}\n'.format(ur, sr))
```

`sol2.cpp` (osaline)

```cpp
int n; cin >> n;

vector<tuple<string, string, int>> subs(n);
for (auto & s : subs)
  cin >> get<0>(s) >> get<1>(s) >> get<2>(s);

// iga osaleja punktid kõigi ülesannete eest
sort(subs.begin(), subs.end());
vector<tuple<int, string>> ranking;
string ue, te; int se = 0;
for (auto & s : subs) {
  string ui = get<0>(s), ti = get<1>(s);
  int si = get<2>(s);
  if (ui != ue) { // uus kasutaja
    ranking.push_back(make_tuple(si, ui));
  } else if (ti != te) { // uus ülesanne
    get<0>(ranking.back()) += si;
  } else { // sama ülesande parem tulemus
    get<0>(ranking.back()) += si - se;
  }
  ue = ui; te = ti; se = si;
}

// sorteerime skooride järgi ja väljastame
sort(ranking.begin(), ranking.end());
for (int i = ranking.size() - 1; i >= 0; --i) {
  auto & r = ranking[i];
  cout << get<1>(r) << ' ' << get<0>(r) << '\n';
}
```

Naiivne lahendus oleks käia läbi kõigi esituste nimekiri ja koostada loetelud kõigist esinenud kasutajanimedest ja kõigist esinenud ülesannete nimedest; siis saab iga kasutajanime jaoks käia läbi kõik ülesandenimed ja otsida kõigi esituste hulgast välja selle paari maksimaalne skoor. Aga selline lahendus peab läbi vaatama kuni 10 000 osalejat, iga osaleja jaoks kuni 10 ülesannet ja siis iga paari jaoks kuni 200 000 esitatud lahendust. Selleks kuluks kokku kuni 20 miljardit operatsiooni, mis ei mahu ajalimiidi sisse.

3. Harjutamine (har)

40 punkti

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

Esimene võimalus selle ülesande lahendamiseks on lugeda päevade kaupa lahendatud ülesannete arvud X_i massiivi (kõrvalolevates programminäidetes `xx[]`) ja hakata selle põhjal päringutele ükshaaval vastama.

Failides `sol0a.cpp` ja `sol0a.py` toodud naiivne lahendus vaatab iga Y_j jaoks läbi kõik paarid (a, b), kus $1 \leqslant a \leqslant b \leqslant N$, ja arvutab igaühe jaoks eraldi summa $X_a + \ldots + X_b$. Kuna paare (a, b) on umbes $N^2/2$ ja iga paari jaoks on vaja kokku liita keskmiselt $N/3$ elementi, kulub igale päringule vastamiseks umbes $N^3/6$ operatsiooni ja M päringu peale kokku suurusjärgus $N^3 \cdot M$ operatsiooni (lühemalt ütleme selle kohta, et lahenduse keerukus on $\mathcal{O}(N^3 \cdot M)$).

Parema lahenduse saamiseks paneme tähele, et iga a väärtuse juures saame igale järgmisele b väärtusele vastava summa eelmisest summast sellele X_b lisamisega, ehk üheainsa liitmisega. Failides `sol0b.cpp` ja `sol0b.py` toodud lahendused saavutavad sellega töömahu $\mathcal{O}(N^2 \cdot M)$. Kuna kõik X_i väärtused on mittenegatiivsed, teame lisaks ka seda, et b väärtuse kasvades summa kunagi ei kahane ja seega võime summa kasvamisel üle Y_j selle a väärtuse uurimise kohe lõpetada.

Sama loogikat saame rakendada ka eelmiselt a väärtuselt järgmisele liikumisel: b suurendamisega liiga suureks kasvanud summa $X_a + \ldots + X_b$ vähendamiseks peame suurendama a väärtust ja niimoodi uuendatud (a, b) paarile vastava summa saame leida eelmisest summast X_a lahutamisega. Failides `sol0c.cpp` ja `sol0c.py` toodud lahendused kasutavadki sellist "libiseva akna" tehnikat ja saavutavad keerukuse $\mathcal{O}(N \cdot M)$.

`sol0a.py` (osaline)

```
def saab(y):
  for a in range(n):
    for b in range(a, n):
      if sum(xx[a : b + 1]) == y:
        return True
  return False
```

`sol0b.py` (osaline)

```
def saab(y):
  for a in range(n):
    s = 0
    for b in range(a, n):
      s += xx[b]
      if s == y:
        return True
      if s > y:
        break
  return False
```

`sol0c.py` (osaline)

```
def saab(y):
  s = 0
  a, b = 0, 0
  while b < n:
    while s < y and b < n:
      s += xx[b]
      b += 1
    while s > y:
      s -= xx[a]
      a += 1
    if s == y:
      return True
  return False
```

Teine võimalus on leida kõigepealt kõik summad, mida üldse on võimalik saavutada, ja siis kontrollida iga päringu jaoks, kas selles küsitav Y_j on saavutatavate summade hulgas. Kuna erinevaid (a, b) paare on umbes $N^2/2$, ei saa ka saavutatavaid summasid rohkem olla (küll aga võib neid olla vähem, kui mitu erinevat lõiku annavad tulemuseks sama summa). Summade endi arvutamise keerukus võib olla kas $\mathcal{O}(N^3)$, nagu failides `sol1a.cpp` ja `sol1a.py`, või $\mathcal{O}(N^2)$, nagu failides `sol1b.cpp` ja `sol1b.py`. Kasutatava hulgatüübi realisatsioonist sõltub, kas elemendi lisamine ja elemendi esinemise kontrollimine kulutavad hulga suurusest sõltumatu konstantse aja (sellised on Pythoni `set` ja C++ `unordered_set`) või kulub N-elemendilise hulgaga tehtavale tehtele $\mathcal{O}(\log N)$ operatsiooni (selline on C++ `set`, mille puhul on tagatud, et hulga läbimisel vaatle-

me elemente nende suuruse järjekorras; kuna meil selles ülesandes ei ole vaja elemente kindlas järjekorras läbi vaadata, pole ka põhjust sellist halvema tööajaga andmestruktuuri kasutada). Õige hulgatüübi kasutamisel on lahenduste kogutööaeg (kõigi summade hulga leidmine ja selle põhjal päringutele vastamine) vastavalt $\mathcal{O}(N^3 + M)$ või $\mathcal{O}(N^2 + M)$, vale hulgatüübi korral aga vastavalt $\mathcal{O}(N^3 \cdot \log N + M \cdot \log N)$ või $\mathcal{O}(N^2 \cdot \log N + M \cdot \log N)$.

`sol1a.py` (osaline)

```
ss = set()
for a in range(n):
  for b in range(a, n):
    ss.add(sum(xx[a : b + 1]))

for y in yy:
  if y in ss:
    print('JAH')
  else:
    print('EI')
```

`sol1b.py` (osaline)

```
ss = set()
for a in range(n):
  s = 0
  for b in range(a, n):
    s += xx[b]
    ss.add(s)

for y in yy:
  if y in ss:
    print('JAH')
  else:
    print('EI')
```

4. Katapult-karussell (`kk`) 50 punkti

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

Lihtsuse mõttes lahutame arvust u ja kõikidest x_i väärtustest 1 ja eeldame vaikselt, et kõiki arvutusi tehakse mooduli n järgi. Paneme tähele, et kui keegi asub enne positsioonil u, siis pärast "hüpet" parameetritega $\langle x_i, y_i \rangle$ asub ta positsioonil $y_i \cdot u + x_i$. Niisiis võime ülesande sõnastada ümber järgnevalt:

> Vaatleme teisendust
>
> $$f : u \mapsto y_S \cdot (y_{S-1} \cdot (\cdots (y_1 \cdot u + x_1) \cdots) + x_{S-1}) + x_S.$$
>
> Arvuta $f^K(u)$.

Alamülesanne 3

Paneme tähele, et

$$y_2 \cdot (y_1 \cdot u + x_1) + x_2 = (y_2 \cdot y_1) \cdot u + (y_2 \cdot x_1 + x_2),$$

$$y_3 \cdot ((y_2 \cdot y_1) \cdot u + (y_2 \cdot x_1 + x_2)) + x_3 = (y_3 \cdot y_2 \cdot y_1) \cdot u + (y_3 \cdot y_2 \cdot x_1 + y_3 \cdot x_2 + x_3)$$

jne. Igal sammul jääb avaldis kujule $p \cdot u + q$. Kordajad p ja q võime välja arvutada, rakendades järjest teisendusi $p \mapsto p \cdot y_i$ ja $q \mapsto y_i \cdot q + x_i$.

Kui oleme p ja q lõpuks välja arvutanud, võime arvule u teisendust $p \cdot u + q$ lihtsalt K korda järjest rakendada. Keerukus tuleb $\mathcal{O}(S + K)$. Et $S \leqslant 10^6$ ja $K \leqslant 10^7$, mahub see ajalimiidi piiresse.

Tehniline komplikatsioon on, et siin on tarvis arvutada $(p \cdot u + q) \bmod n$, kus arvud p, q, u ja n võivad kõik olla ülimalt 2^{63}. Vahevastused keele C++ 64-bitiste täisarvude (`long long`) sisse ei

mahu. Et arvutused jääksid korrektsed, võib kasutada näiteks andmetüüpi `__int128`. Pythonis võivad täisarvud olla kuitahes suured, seega probleeme ei teki. Siiski on ka Pythonis soovitatav võtta vastus igal sammul mooduli n järgi: kuigi keel toetab kuitahes suuri täisarve, hakkavad need väga kiiresti väga palju mälu kulutama ja arvutused muutuvad väga aeglaseks.

Täislahendus

Tähelepanek 1. *Kui meil on lineaarfunktsioonid $f(u) = x \cdot u + y$ ja $g(u) = a \cdot u + b$, siis nende kompositsioon $g(f(u))$ on ka lineaarfunktsioon.*

Tõestus. Tõepoolest,

$$g(f(u)) = a \cdot f(u) + b = a \cdot (x \cdot u + y) + b = (a \cdot x) \cdot u + (a \cdot y + b).$$

□

Seda tähelepanekut me sisuliselt eelmises punktis juba ka kasutasime.

Eelmises alamülesandes leidsime, et meil on lineaarteisendus $f(u) = p \cdot u + q$ ja vaja arvutada $f^K(u)$. Lahendusidee on järgmine:

- kui K on paaris, siis $f^K(u) = f^{K/2}(f^{K/2}(u))$. Kui teame lineaarfunktsiooni $f^{K/2}$ kordajaid, saame f^K kordajad ka välja arvutada.
- kui K on paaritu, siis $f^K(u) = f(f^{(K-1)/2}(f^{(K-1)/2}(u)))$. Kui teame lineaarfunktsiooni $f^{(K-1)/2}$ kordajaid, saame f^K kordajad ka välja arvutada.

Seda lahendusideed realiseerib järgmine rekursiivne funktsioon.

```
// arvutab f^t kordajad, kus f(u) = a * u + b
function calc(int a, int b, int t):
    if t == 0:
        return (1, 0) // ühikteisendus
    if t mod 2 == 0:
        (x, y) = calc(a, b, t / 2)
        return (x**2, x * y + y)
    if t mod 2 == 1:
        (x, y) = calc(a, b, (t - 1) / 2)
        return (a * x**2, a * x * y + a * y + b)
```

Vastuse annab meile siis `calc(p, q, K)` kutsumine. Iga kutsega läheb parameeter t vähemalt kaks korda väiksemaks, aga jääb mittenegatiivseks täisarvuks. Seega kutsutakse funktsiooni `calc` välja $\mathcal{O}(\log K)$ korda. Keerukus on seega $\mathcal{O}(S + \log K)$.

Maatriksitest

Elegantne ja üldine viis selle lahenduse kirjapanekuks on maatriksite kasutamine. *Maatriksiks* nimetatakse matemaatikas $n \times m$ tabelit, mille igasse veergu on kirjutatud arv. Näiteks:

$$\begin{pmatrix} 5 & 1 & 0.3 \\ -1 & 0 & 777 \end{pmatrix} \qquad \begin{pmatrix} 2 & 1 \\ 0 & 0 \\ 324 & \pi \\ \pi^\pi & \sqrt{2} \end{pmatrix} \tag{33}$$

Lihtsuse mõttes piirdume edaspidi 2×2 ruutmaatriksitega, kuid samad põhimõtted jäävad kehtima mistahes mõõtmete korral.

Maatriksite *korrutamine* käib järgmise eeskirja järgi:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00} \cdot b_{00} + a_{01} \cdot b_{10} & a_{00} \cdot b_{01} + a_{01} \cdot b_{11} \\ a_{10} \cdot b_{00} + a_{11} \cdot b_{10} & a_{10} \cdot b_{01} + a_{11} \cdot b_{11} \end{pmatrix}.$$

Maatriksite korrutamisel on veel järgmine tore omadus: see on *assotsiatiivne.* See tähendab, et kui A, B ja C on maatriksid, siis $(A \cdot B) \cdot C = A \cdot (B \cdot C)$.

Tuleme tagasi ülesande juurde. Paneme tähele, et

$$\begin{pmatrix} p & q \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} u & 0 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} p \cdot u + q & 0 \\ 1 & 0 \end{pmatrix}$$

Tähistades $P = \begin{pmatrix} p & q \\ 0 & 1 \end{pmatrix}$ näeme, et meil on vaja arvutada

$$P^K \cdot \begin{pmatrix} u & 0 \\ 1 & 0 \end{pmatrix}.$$

Maatriksite korrutamise assotsiatiivsusest tuleneb, et me ei pea parempoolset maatriksit tobedalt K korda vasakult maatriksiga P korrutama, vaid võime kõigepealt arvutada välja P^K ja siis maatriksid kokku korrutada.

Kuidas aga kiiresti arvutada P^K? Seda saame teha nn "kiire astendamise" meetodil. Paneme tähele, et kui t on paaris, siis $P^t = P^{t/2} \cdot P^{t/2}$. Võime kirjutada rekursiivse funktsiooni (mis on sisuliselt sama, mis kirjeldatud ülal).

```
function exp(matrix P, int t):
    if t == 0:
        return I
    if t mod 2 == 0:
        half = exp(P, t / 2)
        return half * half
    if t mode 2 == 1:
        half = exp(p, (t - 1) / 2)
        return P * half * half
```

Iga kutsega läheb t vähemalt 2 korda väiksemaks. Kui selle funktsiooni abil arvutada P^K, kutsutakse funktsiooni `exp` välja $\mathcal{O}(\log K)$ korda. Keerukus on seega $\mathcal{O}(S + \log K)$.

Samal meetodil saab lahendada paljusid teisi ülesandeid. Klassikaline näide: defineeritud on mingi jada, kus iga liige sõltub mõnedest eelmistest lineaarselt. Näiteks jada, kus $a_1, \ldots, a_4 = 1$ ja $i \geqslant 5$ korral

$$a_i = 5 \cdot a_{i-1} + 3 \cdot a_{i-3} - a_{i-4}.$$

Ülesanne on arvutada selle jada K-s liige, kus $K \leqslant 10^{18}$. Saab defineerida maatriksi P nii, et P teisendab vektori $(a_{i-4}, a_{i-3}, a_{i-2}, a_{i-1})$ vektoriks $(a_{i-3}, a_{i-2}, a_{i-1}, a_i)$ ja arvutada ülaltoodud meetodil P^K.

5. Maagilised taimed (`taim`) 60 punkti

Idee, teostus ja lahenduse selgitus: Tähvend Uustalu

Alamülesanded 1 ja 3

Alamülesannetes 1 ja 3 ei oodata meilt taimede järjekorra valimist: piisab, kui me suudame taimede kasvamise protsessi efektiivselt simuleerida.

Joonistame graafi, mille tippudeks on paarid (p, h) iga $1 \leqslant i \leqslant N$ ja $1 \leqslant h \leqslant K$ korral. Lisaks nendele lisame veel tipu $\top$. Nüüd joonistame servad:

1. iga $1 \leqslant i \leqslant N$ ja $1 \leqslant h < K$ korral suunatud serva $(i, h) \xrightarrow{1} (i, h+1)$
2. iga $1 \leqslant i \leqslant N$ korral suunatud serva $(i, K) \xrightarrow{0} \top$.
3. iga tingimuse $\langle u, a, v, b \rangle$ korral suunatud serva $(v, b) \xrightarrow{1} (u, a)$, kui $u < v$, või serva $(v, b) \xrightarrow{0} (u, a)$, kui $u > v$.

Siin tähistab kirjutis $e \xrightarrow{w} f$ suunatud serva tipust e tippu f kaaluga (ehk ka pikkusega) w.

Tipp (p, h) tähistab sündmust "taim p kasvab h meetri kõrguseks". Tipp $\top$ tähistab sündmust "kõik taimed on kasvanud pikkuseni K". Serv $(p_1, h_1) \xrightarrow{w} (p_2, h_2)$ tähistab piirangut, et sündmus (p_2, h_2) juhtub vähemalt w päeva hiljem kui sündmus (p_1, h_1).

Näiteks sisendi

```
4 4 5
4 4 3 5
3 4 4 3
2 2 3 4
2 5 1 5
```

põhjal joonistatud graaf on illustreeritud joonisel 1.

Tegemist on suunatud tsükliteta graafiga — kui kuskil oleks suunatud tsükkel, siis ei oleks üldse võimalik, et taimed kasvavad kõik K meetri kõrguseks, mis on vastuolu ülesande eeldusega: *On garanteeritud, et kõikides testides on võimalik taimi nii istutada, et ülimalt* 10^{18} *päevaga kasvavad nad kõik vähemalt* K *meetri kõrguseks.*

Tähistagu $\mathrm{dp}[e]$ päeva, mil sündmus e juhtub. On selge, et siis

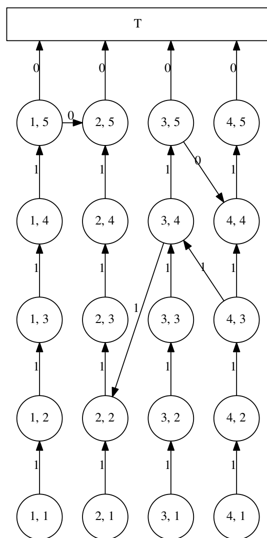
$$\mathrm{dp}[u, 1] = u. \tag{1}$$

Muudel juhtudel paneme tähele, et

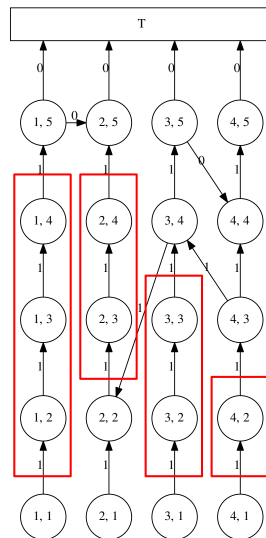
$$\mathrm{dp}[e] = \max_i (w_i + \mathrm{dp}[f_i]), \tag{2}$$

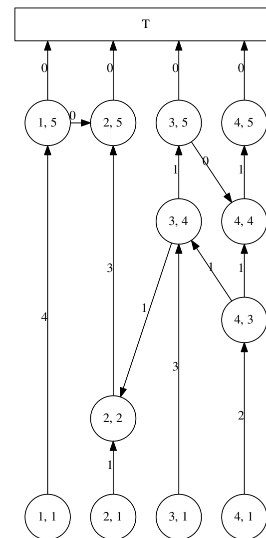
kus maksimum on võetud üle kõikide servade $f_i \xrightarrow{w_i} e$. Tõepoolest: sündmus e juhtub esimesel päeval, mil kõik eeldused selleks on täidetud. Vastus on siis $\mathrm{dp}[\top]$. Arvutame iga e korral $\mathrm{dp}[e]$ välja. Selleks järjestame tipud nii, et kõik servad on suunatud väiksema järjekorranumbriga tipult suurema järjekorranumbriga tipule (nn topoloogiline järjestus, *topological sort*). Nüüd arvutame iga e jaoks valemi põhjal $\mathrm{dp}[e]$ välja. Tingimusi on $\mathcal{O}(N \cdot K + M)$ tükki, seega kõigi $\mathrm{dp}[e]$ arvutamiseks kulub $\mathcal{O}(N \cdot K + M)$ aega.

Kolmanda alamülesande lahendamiseks paneme tähele, et meil on palju tippe, kust on ainult üks serv sisse ja üks serv välja. Võime sellised "vertikaalsed plokid" eemaldada ja panna asemele



Joonis 1: Sõltuvuste graaf





Joonis 2: Ebavajalike tippude eemaldamine

suurema kaaluga serva (vt. joonist 2). Nüüd jääb alles $\mathcal{O}(M+N)$ tippu (alles jäävad ainult iga veeru alumised tipud, ülemised tipud ja need tipud, mis on seotud mingi ülesande tingimusega) ja $\mathcal{O}(M+N)$ serva (lisaks ülesande tingimustele võib igast tipust veel minna üks serv üles). Graafi ehitamine võtab (sõltuvalt täpsest realisatsioonist) aega tõenäoliselt $\mathcal{O}((M+N) cdot \log M)$, tabeli dp täitmine $\mathcal{O}(M+N)$. Keerukus on seega $\mathcal{O}((M+N)\cdot\log M)$.

Täislahendus

Tähistagu T_i iga i korral päeva, mil taime i istutame. Strateegia on järgmine: analüüsime, kuidas sõltub lõppvastus arvudest $T_1, \ldots, T_n$.

Tähistame sümboliga $\operatorname{maxd}(e, f)$ pikimat teekonda tippude e ja f vahel (eeldusel, et see leidub). Defineerime dp nii, nagu eelmistes alamülesannetes, aga valem (1) asendub valemiga

$$\operatorname{dp}[u, 1] = T_u. \tag{3}$$

Tähelepanek 2. *Arv* $\operatorname{dp}[e]$ *avaldub kujul*

$$\operatorname{dp}[e] = \max_u(\operatorname{maxd}((u, 1), e) + T_u). \tag{4}$$

Siin on maksimum võetud üle selliste u ($1 \leqslant u \leqslant N$), et teekond $(u, 1) \rightsquigarrow e$ leidub.

Tõestus. Anname tippudele sellised järjekorranumbrid, et tipud $(u, 1)$ saavad igaüks järjekorranumbri u ja et kõik servad on suunatud väiksema järjekorranumbriga tipu poolt suurema poole. Kasutame matemaatilist induktsiooni.

- *Baas.* Vaatleme tippu järjekorranumbriga $k \leqslant N$, s.t. tippu $(k, 1)$. Siis

  $$\operatorname{dp}[(k, 1)] = T_k = \operatorname{maxd}((k, 1), (k, 1)) + T_k.$$

  See on nõutud kujul, sest ainuke tipp $(u, 1)$, kust teekond tippu $(k, 1)$ leidub, on see, kus $u = k$.

- *Samm.* Vaatleme tippu e järjekorranumbriga $k > N$ ja eeldame, et kõikide väiksema järjekorranumbriga tippude korral valem (4) kehtib. Siis

$$\begin{aligned}\mathrm{dp}[e] &= \max_i(w_i + \mathrm{dp}[f_i]) = \\ &= \max_i(w_i + \max_{u_i}(\mathrm{maxd}((u_i,1),f_i) + T_{u_i})) = \\ &= \max_{i,u_i}(w_i + \mathrm{maxd}((u_i,1),f_i) + T_{u_i}).\end{aligned}$$

See maksimum koosneb hulgast liidetavatest kujul $a_v + T_u$, kusjuures mõne T_u korral võib olla mitu liidetavat. Iga u korral võime alles jätta vaid selle liikme, kus a_v on kõige suurem. Avaldise (4) liige $x + T_u$ kujuneb niisiis järgnevalt.

$$x + T_u = \max_i(w_i + \mathrm{maxd}((u,1),f_i) + T_u),$$

kus maksimum on võetud üle selliste i, et teekond $(u,1) \rightsquigarrow f_i$ leidub. Seega $x = \max_i(w_i + \mathrm{maxd}((u,1),f_i))$, mis on aga täpselt pikim teekond $(u,1) \rightsquigarrow e$.

Selguse mõttes teeme sellesama arvutuse läbi konkreetsete arvudega. Oletame, et tippu e läheb kolm serva $f_1 \xrightarrow{5} e$, $f_2 \xrightarrow{2} e$, $f_3 \xrightarrow{3} e$ ja et

$$\begin{aligned}\mathrm{dp}[f_1] &= \max\{3 + T_1, 5 + T_2, 1 + T_3\}, \\ \mathrm{dp}[f_2] &= \max\{3 + T_2, 5 + T_3\}, \\ \mathrm{dp}[f_3] &= \max\{6 + T_1, 7 + T_4\}.\end{aligned}$$

Siis

$$\begin{aligned}\mathrm{dp}[e] = \max\{&5 + \max\{3 + T_1, 5 + T_2, 1 + T_3\}, \\ &2 + \max\{3 + T_2, 5 + T_3\}, \\ &3 + \max\{6 + T_1, 7 + T_4\}\} \\ = \max\{&\max\{8 + T_1, 10 + T_2, 6 + T_3\}, \\ &\max\{5 + T_2, 7 + T_3\}, \\ &\max\{9 + T_1, 10 + T_4\}\}.\end{aligned}$$

Nüüd on meil maksimum maksimumidest. Vastus ei muutu, kui me sisemised maksimumi võtmised ära jätame ja argumendid ümber järjestame.

$$\begin{aligned}\mathrm{dp}[e] &= \max\{8 + T_1, 10 + T_2, 6 + T_3, 5 + T_2, 7 + T_3, 9 + T_1, 10 + T_4\} \\ &= \max\{9 + T_1, 8 + T_1, 10 + T_2, 5 + T_2, 7 + T_3, 6 + T_3, 10 + T_4\}.\end{aligned}$$

Paneme tähele, et liikmed $8 + T_1$, $5 + T_2$ ja $6 + T_3$ ei saa kunagi olla maksimaalsed, kuna vastavalt liikmed $9 + T_1$, $10 + T_2$ ja $7 + T_3$ on alati suuremad. Seega

$$\mathrm{dp}[e] = \max\{9 + T_1, 10 + T_2, 7 + T_3, 10 + T_4\}.$$

Kuidas kujunes siin näiteks liige $9 + T_1$? See tekkis valemiga

$$9 + T_1 = \max\{5 + 3 + T_1, 3 + 6 + T_1\} = \max\{8 + T_1, 9 + T_1\}.$$

□

Vastus dp[⊤] on siis

$$\text{dp}[\top] = \max_u(\text{maxd}((u, 1), \top) + T_u).$$

Arvud $\text{maxd}((u, 1), \top)$ võime välja arvutada graafi läbimise teel. Selleks on kasulik tähele panna, et kui pöörata kõik servad tagurpidi, on tegemist pikima teekonnaga $\top \rightsquigarrow (u, 1)$. Niisiis piisab, kui arvutame pikima teekonna tipust $\top$ igasse teise tippu. Saame arvud $D_u = \text{maxd}((u, 1), \top)$.

Ülesanne on nüüd taandunud järgnevale: meil on antud arvud $D_1, \ldots, D_n$ ja peame valima paarikaupa erinevad positiivsed täisarvud $T_1, \ldots, T_n$ nii, et maksimum $\max(D_i + T_i)$ oleks minimaalne. Siin saab kasutada lihtsat ahnet strateegiat: paneme suurimale arvudest D_i vastavusse arvu $T_i = 1$, suuruselt järgmisele $T_i = 2$ jne, kuni vähimale arvudest D_i läheb vastavusse arv n.

Veendume, et see annab vähima võimaliku maksimumi. Vaatleme mingit optimaalset paigutust, mis ei ole selline nagu meil. Seal peavad kindlasti leiduma paarid $\langle D, T \rangle$ ja $\langle D', T' \rangle$, kus $D < D'$ ja $T < T'$. Siis võime T ja T' omavahel ära vahetada ja maksimum sellega kasvada ei saa. Võime niimoodi paare vahetada, kuni saame meie defineeritud vastavuse. Maksimum võrreldes optimaalsega kasvanud ei ole, seega on ka meie paigutus optimaalne.

Lõpuks arvutame keerukuse.

- Graafi ehitamine võtab aega $\mathcal{O}((N + M) \cdot \log M)$.
- Pikimate teekondade leidmine võtab aega $\mathcal{O}(N + M)$.
- Arvudele D_i arvude T_i vastavusse panemine (mis sisaldab tõenäoliselt sorteerimist) võtab aega $\mathcal{O}(N \cdot \log N)$.

Kokku on keerukus seega $\mathcal{O}((N + M) \cdot \log M + N \cdot \log N)$. Kui eeldada, et $N = M$, saame $\mathcal{O}(N \cdot \log N)$.

6. Maagiline BF (`mbf`) — 60 punkti

Idee, teostus ja lahenduse selgitus: Sandra Schumann

Iga alamülesande jaoks lühikese üherealise programmi kirjutamine

Selle ülesande lahendamist on mõtet alustada võimalikult lühikese alamülesannet lahendava BF programmi kirjutamisest, pööramata tähelepanu ruudustikule.

1. `>[-<->]`
2. `[>+>+<<-]>+>------[-------<------->]<-[-<+>]`
3. `[[-]+>]<[[-<+>]<]`
4. `-[->[->>+<<]>[>+<<+>-]>[-<+>]<<<]>[-<+>]`
5. `>>[[-<-<+>>]<<[->+<]>[->+<]>>]<[[-<+>]<]` (Rao Zvorovski)

Lahenduste väljamõtlemiseks on kasulik tabelarvutusprogramm, milles käsitsi proovida programmi läbi jooksutada. Suuremaks hulgaks testimiseks on aga hea kirjutada oma interpretaator.

Koodi tuleks testida nagu iga teist koodijuppi: andes sellele hulga erinevaid sisendeid ja vaadates, kas kood annab nende peale ootuspärased väljundid. Sisendid on praeguses ülesandes sisendmassiivis olevad arvud ja väljund programmi lõpuks sisendmassiivi esimesse lahtrisse kirjutatud arv. See on ka see, kuidas esitatud lahendusi testiti serveris.

Näiteks katsetas testimisserver esimese alamülesande jaoks esitatud koode järgmiste arvutuskäikude peal:

- $7 - 5 = 2$
- $0 - 0 = 0$
- $200 - 200 = 0$
- $199 - 0 = 199$
- $100 - 99 = 1$
- $153 - 54 = 99$

N-käsulisest BF programmist $N \times N$ MBF ruudustiku tegemine

$N \times N$ ruudustikus mahub N-käsuline BF programm ära ühele reale. Lihtne meetod sellest ülesande tingimustele vastava ruudustiku tegemiseks on panna töötav kood nii esimesele reale kui ka esimesse veergu ja seejärel täita ülejäänud ruudustik `>`-sümbolitega, eesmärgiga liigutada kursor kaugele paremale, et ülejäänud kood ei mõjutaks kuidagi esimesse lahtrisse kirjutatut. Selleks, et ruudustik õige tulemuse annaks, peab mõnikord mängima ka sulgudega: muidu võib tekkida näiteks rohkem sulgevaid sulge (`]`) kui avanevaid (`[`). Lisasulge on kõige parem paigutada diagonaalile, siis mõjutavad nad mõlemat pidi kirjutatud koodi samamoodi.

Näiteks eelnevalt toodud teise alamülesande lahendusele vastav ruudustik oleks järgmine:

```
[>+>+<<-]>+>------[-------<------->]<-[-<+>]
>[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
+>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
+>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
+>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
+>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
]>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

Sellised lahendused on ebaefektiivsed, kuid saavad juba päris mitu punkti.

Ruudustiku küljepikkuse optimeerimine

Üherealist lahendust on aga võimalik ka tihedamasse ruutu kokku keerata. Selleks on kasulik proovida hoida kogu kood ruudustiku peadiagonaali suhtes sümmeetriline: siis saab koodi optimeerida mõlemas suunas samamoodi.

Ruudustiku suuruse kaks korda vähendamiseks saab koodi esimese rea jagada kahe rea peale. Selleks, et ruudustik püsiks sümmeetriline, tuleb seda teha nii, et esimese rea teine märk ja teise rea esimene märk oleks samad. Näiteks `>>[[-<-<+>>]<<[->+<]>[->+<]>>]<[[-<+>]<]` saab kahe rea peale suruda järgmiselt:

```
>>[[-<-<+>>]<<[->+<]
>[->+<]>>]<[[-<+>]<]
```

Jätkata saab ka koodirea kolmeks või isegi neljaks jagamisega. See kõik vähendab kiiresti N suurust, aga iga uus rida toob juurde probleeme: sellisel viisil tegutsedes saab ridu poolitada ainult nii, et tulpadesse juba kirjutatud sümbolid oleksid koodis õige koha peal. Näiteks neljanda rea jaoks tähendab see, et pärast kolmanda rea lõppu koodis olevad kolm järgmist sümbolit peavad vastavalt kattuma esimese, teise ja kolmanda rea neljandate sümbolitega, ainult siis püsib ruudustik sümmeetriline. Nagu arvata võib, on selline kokkusattumus üpris ebatõenäoline.

Koodi on aga võimalik kohati pikendada, et "kokkusattumusi"õigetesse kohtadesse sättida. Näiteks ei tee liigutus `><` funktsionaalselt mitte midagi, aga nihutab edasist koodi kahe märgi võrra ja seega võib aidata mõningaid märke õigetesse kohtadesse liigutada. Sama kehtib `+-` kohta. Ka `<>` ja `-+` võivad samasse kategooriasse langeda, kuid mitte alati.

Eespool toodud näited pikendavad koodi alati paarisarvu märkide võrra. Mõnikord on aga koodi vaja pikendada paaritu arvu märkide võrra. Kui teha `+` või `-` operatsioone lahtritel, millest me ei hooli, siis saab koodi pikendada ka paaritu arvu märkide võrra. Samuti võib vabalt lisada `-` kohe

`]` järele, sest tsükkel saab lõppeda ainult siis, kui mäluviit on parasjagu nullväärtusega lahtril. Kasulik võib olla ka `[` lisamne kohe `]` järele, sest see alustab tsüklit, millesse kindlasti ei siseneta. Selline tsükkel on vaja mingil hetkel ka lõpetada vastava `]`-suluga, et programmi tööga edasi minna, kuid vahele võib sisestada peaaegu mistahes märke.

Optimeerimistehnikaid on teisigi, kokkuvõttes oleneb kõik sellest, missugust sümbolite paigutust on antud hetkel vaja koodis saavutada.

Head lahendused

Alamülesandele 1 on optimaalne N väärtus 3:

```
>[-
<->
><]
```

Alamülesandele 2 on olemas vähemalt lahendus suurusega $N = 11$:

```
[>+>+<<-]>+
>------[---
+-----<---<
>---->]<-[+
+---<+>]>>>
<-->+>>>>>[
<-<]>>>>>>>
-[-<]>>>>>[
]--->>>>>>>
>--[>>>>>>]
+-<+>[>[>]]
```

Alamülesandele 3 on olemas lahendus suurusega $N = 7$:

```
[[-]+>]
[]<><[[
-<+>][-
]>><<[-
+<]<<>]
>[[[>]>
][--]>]
```

Alamülesandele 4 on olemas lahendus $N = 11$ jaoks (autor Andres Alumets):

```
-<-[->[>>+<
<-+-]>[>+<+
-+-<+>-]-->
[-<+>]<<<][
-]+><<<-[--
>>>]<<+-->>
```

```
[[-<<+>>]][
>>]<-->[[--
>+-<[-][---
+<-]->]---]
<+>[->[--]]
```

Alamülesandele 5 on olemas lahendus, kus $N = 9$ (autor Andres Alumets):

```
>[><>]--<
[[-<->>+<
>-+<]>[->
<<<+>>][-
>-]><<<<>
]>>><>>>>
->[]<>>>>
-+-[<>>>>
<<>->>>>]
```