Sisukord

Juta teekond **2**

Numbrid **4**

Erinevused **7**

Sipelgas **9**

Hinded **11**

Segane väljund **14**

1. Juta teekond (`teed`)

1 sekund 10 punkti

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

Üks võimalus selle ülesande lahendamiseks on leida kõik võimalikud arvujadad, mille Juta võib oma teekonnal üles kirjutada, ja siis kontrollida, kas sisend on üks neist. See pole väga raske, sest kokku on üldse ainult kaheksa võimalikku jada. Siiski on selline käsitsi kõigi variantide välja kirjutamine üsna tüütu. See on ka natuke veaohtlik, sest kui ühes neist variantidest trükiviga teha, siis saame programmi, mis enamasti töötab küll õigesti, aga vahel harva siiski ka valesti.

Parem võimalus on koostada lahendus teelõikude kaupa:

1. Esimesel teelõigul võib Juta minna kas ülemist või alumist haru mööda ja kirjutada üles vastavalt kas arvu 1 või arvu 3. Järelikult, kui sisendi esimesel real on mingi muu arv, siis ei saa see olla Juta teekonna üleskirjutus.
2. Ka teisel teelõigul võib Juta minna kas ülemist või alumist haru mööda ja siis kirjutada üles vastavalt kas arvu 6 või arvu 8. Järelikult, kui sisendi teisel real on mingi muu arv, siis ei saa see olla Juta teekonna üleskirjutus.
3. Kolmandal teelõigul võib Juta samuti minna kas ülemist või alumist haru mööda ja siis kirjutada üles vastavalt kas arvu 2 või arvu 5. Järelikult, kui sisendi kolmandal real on mingi muu arv, siis ei saa see olla Juta teekonna üleskirjutus.
4. Kui üheski kolmest eelmisest punktist vastuolu ei tekkinud, siis on võimalik, et sisendis olev arvujada on saadud Juta teekonna üleskirjutusena.

Eelnev loogika näeb keeltes Python ja C++ programmeerituna välja umbes selline:

Python:

```
arv = int(input())
if arv != 1 and arv != 3:
  print("EI")
  exit()

arv = int(input())
if arv != 6 and arv != 8:
  print("EI")
  exit()

arv = int(input())
if arv != 2 and arv != 5:
  print("EI")
  exit()

print("JAH")
```

C++:

```
#include <iostream>
using namespace std;

int main() {
  int arv;

  cin >> arv;
  if (arv != 1 and arv != 3) {
    cout << "EI\n";
    return 0;
  }

  cin >> arv;
  if (arv != 6 and arv != 8) {
    cout << "EI\n";
    return 0;
  }

  cin >> arv;
  if (arv != 2 and arv != 5) {
    cout << "EI\n";
    return 0;
  }

  cout << "JAH\n";
}
```

Pythonis võib tingimuse '`arv != 1 and arv != 3`' asemel kirjutada ka '`arv not in [1, 3]`'. Kui võimalikke väärtusi on rohkem kui kaks, siis on teine variant lühem ja selgem.

Soovi korral võiks kõigi ridade kohta käivad tingimused koguda kokku ja kirjutada ühe suurema loogilise avaldisena, Pythonis näiteks nii:

```
arv1 = int(input())
arv2 = int(input())
arv3 = int(input())

if (arv1 == 1 or arv1 == 3) and \
    (arv2 == 6 or arv2 == 8) and \
    (arv3 == 2 or arv3 == 5):
  print("JAH")
else:
  print("EI")
```

See siiski pigem ei ole hea stiil, sest selliseid suuri avaldisi on keerulisem lugeda, mis teeb raskemaks nende tähenduse mõistmise ja õigsuse kontrollimise.

Keerulisemate avaldiste kirjutamisel peab silmas pidama ka tehete järjekorda. Eelolevas näites on sulud `or`-tehete ümber vajalikud, sest üldiselt on kõigis programmikeeltes `and`-tehte prioriteet kõrgem ja ilma sulgudeta avaldist

```
arv1 == 1 or arv1 == 3 and arv2 == 6 or arv2 == 8 and arv3 == 2 or arv3 == 5
```

tõlgendab kompilaator, nagu see oleks

```
arv1 == 1 or (arv1 == 3 and arv2 == 6) or (arv2 == 8 and arv3 == 2) or arv3 == 5
```

mille tähendus on muidugi hoopis teine.

2. Numbrid (numbrid)

1 sek / 3 sek 20 punkti

Idee ja teostus: Heno Ivanov, lahenduse selgitus: Ahto Truu

> Antud hulk arve. Leida kõik arvud, mis jäävad mingi kahe antud arvu vahele, kuid ei kuulu antud arvude hulka. Arvude väärtused on kuni 10^8, aga ei antud arve ega otsitavaid arve pole üle 50 000.

Selles ülesandes on ilmne lahendus vaadata läbi kõik arvud alates vähimast antud arvust kuni suurima antud arvuni ja kontrollida igaühe kohta, kas see esineb antud arvude hulgas. Arvude väärtused on kuni 10^8, mis võib tekitada kartuse, et kõigi võimaluste läbivaatus kestab liiga kaua. Aga kui on teada, et antud arve pole üle 50 000 ja nende vahele jäävaid pole ka üle 50 000, siis see tähendab, et üheski sisendis ei saa vähima ja suurima antud arvu vahe olla üle 100 000.

Selle idee naiivne realisatsioon Pythonis:

```
n = int(input())
soovid = list(map(int, input().split()))

vastus = []
for number in range(min(soovid), max(soovid)):
  if number not in soovid:
    vastus.append(number)

print(len(vastus))
print(*vastus)
```

ja C++'s:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> soovid;
  for (int i = 0; i < n; ++i) {
    int soov;
    cin >> soov;
    soovid.push_back(soov);
  }

  vector<int> vastus;
  const int min_soov = *min_element(soovid.begin(), soovid.end());
  const int max_soov = *max_element(soovid.begin(), soovid.end());
  for (int number = min_soov; number < max_soov; ++number) {
    if (find(soovid.begin(), soovid.end(), number) == soovid.end()) {
      vastus.push_back(number);
    }
  }

  cout << vastus.size() << '\n';
  for (auto number : vastus) {
    cout << number << ' ';
  }
  cout << '\n';
}
```

Nende lahenduste miinus on, et need hoiavad soovide nimekirja loendina, kus elemendi olemasolu kontroll vajab loendi läbivaatamist, milleks kulub halvimal juhul kuni N sammu. Kõigi võimalike arvude kontrolliks kulub seega $(\max - \min) \cdot N$ sammu. Selles ülesandes olid küll sisendi ja väljundi piirid ning ajalimiidid nii valitud, et ka selline naiivne lahendus võis maksimumpunktid teenida, aga üldiselt jääb see suuremate andmemahtude korral aeglaseks.

Üks võimalus lahendust kiirendada on võtta soovide nimekirja hoidmiseks loenditüübi asemel kasutusele hulgatüüp, milles elemendi olemasolu kontroll on palju efektiivsem. Pythonis piisab selleks, kui

```
soovid = list(map(int, input().split()))
```

asemel kirjutada

```
soovid = set(map(int, input().split()))
```

C++ kasutajatel on vaja teha kolm asendust. Esiteks peab välja vahetama soovide nimekirja andmetüübi, milleks tuleb

```
vector<int> soovid;
```

asemel kirjutada

```
set<int> soovid;
```

Teiseks tuleb elementide nimekirja lisamine

```
soovid.push_back(soov);
```

asemel kirjutada kujul

```
soovid.insert(soov);
```

Kõige tähtsam on, et elemendi olemasolu tingimuseks peab

```
if (find(soovid.begin(), soovid.end(), number)== soovid.end())
```

asemel kirjutama

```
if (!soovid.contains(number))
```

Üldine `find`-funktsioon töötab ka hulgatüübiga, aga ei oska selle eripärasid kasutada ja selline lahendus töötaks `set`-tüüpi andmehoidlaga isegi aeglasemalt kui `vector`-tüüpi hoidlaga.

Teine võimalus efektiivsema lahenduse saamiseks on vaadata uuesti selgituse alguses toodud ülesandepüstituse lühikokkuvõtet: "leida kõik arvud, mis jäävad mingi kahe antud arvu vahele, kuid ei kuulu antud arvude hulka". Nimelt, kui arvujada ära sorteerida, siis on järjestikuste elementide vahele jäävad "augud" (ja kõik neisse kuuluvad väärtused) lihtsalt ja efektiivselt leitavad.

Selle idee realisatsioon Pythonis:

```
n = int(input())
soovid = [int(s) for s in input().split()]

soovid.sort()

vastus = []
for i in range(1, n):
  for number in range(soovid[i - 1] + 1, soovid[i]):
    vastus.append(number)

print(len(vastus))
print(*vastus)
```

ja C++'s:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int main() {
  int n;
  cin >> n;
  vector<int> soovid(n);
  for (auto & soov : soovid) {
    cin >> soov;
  }

  sort(soovid.begin(), soovid.end());

  vector<int> vastus;
  for (int i = 1; i < n; ++i) {
    for (int number = soovid[i - 1] + 1; number < soovid[i]; ++number) {
      vastus.push_back(number);
    }
  }

  cout << vastus.size() << '\n';
  for (auto number : vastus) {
    cout << number << ' ';
  }
  cout << '\n';
}
```

3. Erinevused (erinevused)

0,5 sek / 3 sek — 30 punkti

Idee: Heno Ivanov, teostus ja lahenduse selgitus: Ahto Truu

> Antud N täisarvu $A_1, A_2, \ldots, A_N$. Leida A_i ja A_j vahede absoluutväärtuste summa üle kõigi paaride $1 \leqslant j < i \leqslant N$. On teada, et $N \leqslant 100\,00$ ja $0 \leqslant A_i \leqslant 1\,000$.

Naiivne lahendus on kõik paarid läbi vaadata ja vahede absoluutväärtused vahetult kokku liita.

Python:

```python
...
s = 0
for i in range(n):
  for j in range(i):
    s += abs(a[i] - a[j])
...
```

C++:

```cpp
...
long long s = 0;
for (int i = 0; i < n; ++i)
  for (int j = 0; j < i; ++j)
    s += std::abs(a[i] - a[j]);
...
```

C++ lahenduses tasub tähele panna muutuja `s` tüüpi: kuigi arvud A_i ja nende vahed on üsna väikesed, ei mahu vahede summa suuremates testides 32-bitisesse muutujasse ära. Tõsi, eeltoodud naiivsed lahendused on nii suurtes testides ka liiga aeglased ja sellises C++ lahenduses 32-bitise muutuja kasutamisega võistlusel punkte kaotanud ei oleks.

Küll aga on `s` õige tüüp oluline efektiivsemates lahendustes, mille leidmiseks annab vihje teise alamülesande kitsendus. Kui arvud on kasvavalt järjestatud, tekib seaduspära, et igas paaris, kus $j < i$ on ka $A_j < A_i$ ja seega läheb igas sellises paaris A_i summasse plussmärgiga. Samas igas paaris, kus $i < k$, on ka $A_i < A_k$ ja igas sellises paaris läheb A_i summasse miinusmärgiga. Seega läheb iga A_i summase plussmärgiga $i - 1$ ja miinusmärgiga $N - i$ korda. See algoritm annab õige tulemuse ka juhul, kui andmetes on korduvaid väärtusi: igas võrdsete elementide paaris arvestame ühte neist pluss- ja ja teist miinusmärgiga, täpselt nagu peab. Sellest tähelepanekust saamegi efektiivsemad lahendused:

Python:

```python
...
a.sort()
s = 0
for i in range(n):
  s += a[i] * i
  s -= a[i] * (n - 1 - i)
...
```

C++:

```cpp
...
std::sort(a.begin(), a.end());
long long s = 0;
for (int i = 0; i < n; ++i) {
  s += a[i] * i;
  s -= a[i] * (n - 1 - i);
}
...
```

Eeltoodud programminäidetes on avaldised '`s += a[i] * i`' ja '`s -= a[i] * (n - 1 - i)`' sellepärast, et nii Pythoni `list`- kui ka C++ `vector`-tüübis on indeksid $1 \ldots N$ asemel $0 \ldots N - 1$.

C++ lahenduses peame jälle mõtlema ka tüüpidele. Kui `a[i]` ja `i` on mõlemad 32-bitised muutujad, siis arvutatakse ka nende korrutis välja 32-bitisena. See, et tulemus pärast liidetakse 64-bitisele muutujale `s`, ei hoiaks korrutise liiga suure väärtuse korral ära 32-bitise vahetulemuse ületäitumist. Selles ülesandes on ka maksimaalsel juhul (kui $i = N - 1 = 99\,999$ ja $A_i = 1\,000$) korrutis veel piisavalt väike ja omistamine '`s += a[i] * i;`' seega korrektne ka 32-bitiste `a[i]` ja `i` korral. Aga kui A_i väärtuste ülempiir oleks näiteks $100\,000$, peaks ületäitumise vältimiseks hoolitsema, et korrutamine tehtaks 64-bitisena.

Ületäitumise vältimiseks valitud suhteliselt väike A_i ülempiir avab ukse ka veel ühele efektiivsele lahendusele, mille saame mõnes mõttes kahe eelmise lahenduse ideid kombineerides. Nimelt on võimalikke erinevaid tulemusi piisavalt vähe (ainult 1 001), et jõuame ajalimiiti ületamata vaadata läbi kõigi võimalike tulemuste kõik paarid (mida on $1\,001 \cdot 1\,000/2 = 500\,500$).

Sellest saamegi järgmise lahenduse idee: loendame iga võimaliku tulemuse esinemiste arvud; seejärel vaatame läbi kõik võimalikud tulemuste paarid ja liidame iga paari elmentide vahe summasse nii mitme kordselt, kui mitu sellist tulemuste paari sisendis tegelikult leidub; paaride arvu leidmiseks piisab, kui teame, mitu korda esineb sisendis paari esimene ja mitu korda teine liige: paaride arv on siis nende esinemiste arvude korrutis, sest esimese tulemuse iga esinemine moodustab paari teise tulemuse iga esinemisega. Programmikoodis näeb see välja nii:

Python:

```python
...
mitu = [0] * piir
for t in a:
  mitu[t] += 1

s = 0
for t in range(piir):
  for u in range(t):
    s += (t - u) * mitu[t] * mitu[u]
...
```

C++:

```cpp
...
std::vector<int> mitu(piir);
for (auto t : a) {
  mitu[t] += 1;
}

long long s = 0;
for (int t = 0; t < piir; ++t) {
  for (int u = 0; u < t; ++u) {
    s += (long long) (t - u) *
      mitu[t] * mitu[u];
  }
}
...
```

Taas kord andmetüüpidele mõeldes näeme, et seekord on ületäitumise vältimiseks tõesti vaja tagada, et korrutamine tehtaks 64-bitisena. Halvimal juhul, kui 100 000 elemendiga sisendis on kaks väärtust, kumbki 50 000 korda, on nende kahe esinemissageduse korrutis 2 500 000 000 ja seega 32-bitise märgiga täisarvu jaoks juba liiga suur; lisaks võib tulemuste vahe olla kuni 1 000, mis viib kolme teguri korrutise ülempiiri 2 500 000 000 000-ni. Eelvoorus kasutatud komplektis sellist testi ei olnud, aga lõppvoorus sarnases olukorras arvatavasti juba oleks.

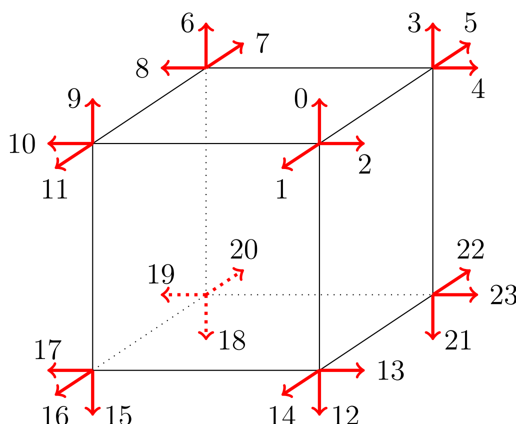
4. Sipelgas (sipelgas)

1 sekund 40 punkti

Idee ja teostus: Heno Ivanov, lahenduse selgitus: Ahto Truu

> Kuubi tipus seisev robotsipelgas liigub käsu `V` peale järgmisse tippu mööda endast vasakul olevat, käsu `P` peale mööda paremal olevat serva. Sipelga täidetud käskude jada põhjal leida lühim võimalik tee tagasi algustippu.

Ülesanne leida lühim tee peaks mõtted kohe graafidele viima. Graafiülesandena lahendamiseks peame tähele panema, et sipelga olekus on lisaks tema asukohale oluline ka tema pea suund, milleks kuubi igas tipus on kolm võimalust.



Kui nummerdame olekud nii, nagu näidatud ülaloleval joonisel, ja kirjutame iga oleku jaoks välja, millisesse olekusse sipelgas sealt kummagi käsuga edasi liigub, on juba lihtne koostada ka funktsioon, mis leiab algolekust mingite käskude täitmise järel saadud lõppoleku:

Python:

```
vasakule = [
  10, 12,  5,  1, 21,  8,
   4, 18, 11,  7, 15,  2,
  22,  0, 17, 13,  9, 20,
  16,  6, 23, 19,  3, 14]
paremale = [
   5, 10, 12,  8,  1, 21,
  11,  4, 18,  2,  7, 15,
  17, 22,  0, 20, 13,  9,
  23, 16,  6, 14, 19,  3]

def liigu(olek, sammud):
  for samm in sammud:
    if samm == 'V':
      olek = vasakule[olek]
    if samm == 'P':
      olek = paremale[olek]
  return olek
```

C++:

```
vector<int> vasakule = {
  10, 12,  5,  1, 21,  8,
   4, 18, 11,  7, 15,  2,
  22,  0, 17, 13,  9, 20,
  16,  6, 23, 19,  3, 14};
vector<int> paremale = {
   5, 10, 12,  8,  1, 21,
  11,  4, 18,  2,  7, 15,
  17, 22,  0, 20, 13,  9,
  23, 16,  6, 14, 19,  3};

int liigu(int olek, string sammud) {
  for (auto samm : sammud) {
    if (samm == 'V') {
      olek = vasakule[olek];
    }
    if (samm == 'P') {
      olek = paremale[olek];
    }
  }
  return olek;
}
```

Tagasitee võiksime muidugi leida mõne standardse lühima tee leidmise algoritmiga. Aga saab ka lihtsamalt, kui paneme tähele, et mistahes tipust mistahes teise tippu saab alati ülimalt kolme sammuga. See tähendab, et võimalikke tagasiteid pole kuigi palju ja võime kõik variandid lihtsalt programmi sisse kirjutada:

Python:

```
teed = [
  "",
  "V", "P",
  "VV", "VP", "PV", "PP",
  "VVV", "VVP", "VPV", "VPP",
  "PVV", "PVP", "PPV", "PPP"]
```

C++:

```
vector<string> teed = {
  "",
  "V", "P",
  "VV", "VP", "PV", "PP",
  "VVV", "VVP", "VPV", "VPP",
  "PVV", "PVP", "PPV", "PPP"};
```

Õigupoolest saaks ka nendest nimekirjadest veel mitmeid variante välja jätta. Näiteks peaks olema lihtne näha, et mistahes olekust alustades jõuame käsujadade `"VP"` ja `"PV"` järel alati samasse tippu (kuigi mitte samasse olekusse). Aga siiski on need nimekirjad juba praegu piisavalt lühikesed, et on triviaalne kõik variandid läbi proovida ja esimene lähtetippu tagasi viiv variant vastusena välja trükkida:

Python:

```
_ = int(input())
tee = input().strip()
olek = liigu(0, tee)

for tee in teed:
  if liigu(olek, tee) in [0, 1, 2]:
    print(len(tee))
    print(tee)
    break
```

C++:

```
int main() {
  int n;
  cin >> n;
  string tee;
  cin >> tee;

  int olek = liigu(0, tee);
  for (auto tee : teed) {
    int lopp = liigu(olek, tee);
    if (lopp == 0 or lopp == 1 or
        lopp == 2) {
      cout << tee.length() << '\n';
      cout << tee << '\n';
      break;
    }
  }
}
```

5. Hinded (`hinded`)

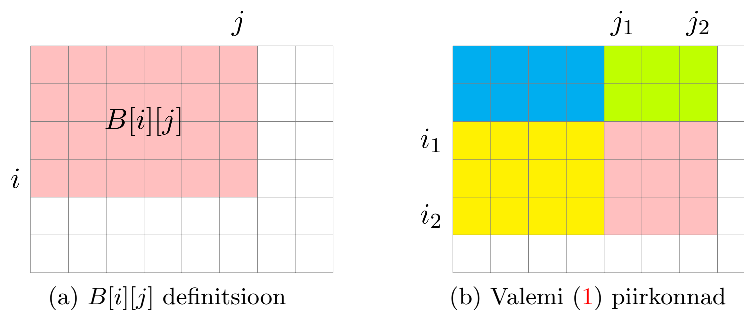
1 sek / 5 sek 60 punkti

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

Lahutame igast lahtrist arvu 50. Nüüd taandub ülesanne kujule

> On antud ruudustik A. Igasse lahtrisse on kirjutatud üks arv. Leia ruudustikus selline ristkülik, mille summa on kõikide ristkülikute seas vähim võimalik.

Selle ülesande lahendamiseks on kõigepealt hea teada, kuidas kiiresti arvutada alamristkülikute summasid. Teeme seda nn osasummade meetodi abil (ingl k. *prefix sums*): arvutame välja teise maatriksi B nii, et $B[i][j]$ on maatriksi A kõigi nende lahtrite summa, mis jäävad lahtrist (i, j) vasakule ja üles (ehk nende lahtrite, mille rea number on ülimalt i ja veeru number ülimalt j).



(a) $B[i][j]$ definitsioon (b) Valemi (1) piirkonnad

Joonis 1: Osasummad ja nende kasutamine

Nüüd on ristküliku $A[i_1 \dots i_2][j_1 \dots j_2]$ summa leitav valemiga

$$B[i_2][j_2] - B[i_1 - 1][j_2] - B[i_2][j_1 - 1] + B[i_1 - 1][j_1 - 1]. \qquad (1)$$

Tõepoolest:

- Lahter, mis asub ristküliku sees (joonisel 1b punane piirkond), on esindatud ainult liidetavas $B[i_2][j_2]$.
- Lahter, mille rea number jääb i_1 ja i_2 vahele, kuid veeru number on väiksem kui j_1 (joonisel kollane piirkond), on esindatud liidetavates $B[i_2][j_2]$ ja $B[i_2][j_1 - 1]$. Nende märgid on valemis (1) vastupidised, seega need lahtrid summasse ei panusta.
- Lahter, mille veeru number jääb j_1 ja j_2 vahele, kuid rea number on väiksem kui i_1 (joonisel roheline piirkond), on esindatud liidetavates $B[i_2][j_2]$ ja $B[i_1 - 1][j_2]$. Nende märgid on valemis (1) vastupidised, seega need lahtrid summasse ei panusta.
- Lahter, mille rea number on väiksem kui i_1 ja veeru number väiksem kui j_1 (joonisel sinine piirkond) on esindatud kõigis neljas liidetavas. Et kahel liidetaval on valemis (1) plussmärk ja kahel miinusmärk, siis ka need lahtrid summasse ei panusta.
- Ülejäänud lahtrid ei panusta ühtegi liidetavasse.

Teisisõnu, kui valemis (1) kõik B-d A-de summana lahti kirjutada, siis taanduvad välja kõik liidetavad, välja arvatud need, mis on ristküliku $A[i_1 \dots i_2][j_1 \dots j_2]$ sees. Niisiis, kui meil on selline maatriks B välja arvutatud, saame mistahes ristküliku elementide summa arvutada konstantse ($O(1)$) ajaga.

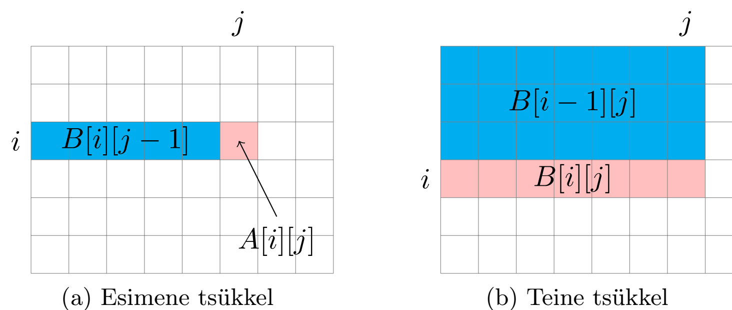
Maatriksi B saab välja arvutada alloleva pseudokoodi abil:

```
// B on (n + 1) x (n + 1) maatriks, alguses täidetud nullidega
// eeldame, et A elemendid on indekseeritud A[1..n][1..n]
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
    B[i][j] = B[i][j - 1] + A[i][j];

for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
    B[i][j] += B[i - 1][j];
```

Veendume selle korrektsuses.

1. Hetkel, kui omistame `B[i][j] = B[i][j - 1] + A[i][j]`, on (joonis 2a) $B[i][j-1]$ ristküliku $A[i][1\dots j-1]$ summa. Seega $B[i][j-1] + A[i][j]$ on ristküliku $A[i][1\dots j]$ summa.
2. Hetkel, kui omistame `B[i][j] += B[i - 1][j]`, on (joonis 2b) $B[i][j]$ ristküliku $A[i][1\dots j]$ summa, $B[i-1][j]$ aga juba $A[1\dots i-1][1\dots j]$ summa. Seega pärast tehte tegemist on $B[i][j]$ ristküliku $A[1\dots i][1\dots j]$ summa.



(a) Esimene tsükkel

(b) Teine tsükkel

Joonis 2: Osasummade arvutamine

Nii on võimalik maatriks B arvutada välja $O(n^2)$ ajaga. Pärast välja arvutamist saab igale päringule vastata $O(1)$ ajaga. See annab meile ka automaatselt teise alamülesande lahenduse: käime läbi kõik nelikud i_1, i_2, j_1, j_2, arvutame valemi (1) abil välja iga ristküliku $A[i_1 \dots i_2][j_1 \dots j_2]$ summa ja võtame nendest summadest miinimumi.

Saab aga ka paremini. Oletame, et oleme otsustanud i_1, i_2 ja j_2 väärtused. Peame leidma sellise $j_1 \leqslant j_2$, mis minimeerib summat

$$B[i_2][j_2] - B[i_1 - 1][j_2] - B[i_2][j_1 - 1] + B[i_1 - 1][j_1 - 1].$$

Liidetavad $B[i_2][j_2]$ ja $-B[i_1 - 1][j_2]$ ei sõltu j_1 väärtusest. Seega võime sama hästi otsida sellist $j_1 \leqslant j_2$, mis minimeerib summat

$$-B[i_2][j_1 - 1] + B[i_1 - 1][j_1 - 1].$$

See summa aga ei sõltu jälle kuidagi j_2 väärtusest. See viib meid täislahenduseni. Käime läbi kõik i_1 ja i_2 paarid. Iga paari kohta käime läbi kõik j-d kasvavas järjestuses. Iga j korral jätame meelde $-B[i_2][j-1] + B[i_1][j-1]$ väärtuse. Seejärel valime juba meelde jäetud väärtuste seast vähima ja liidame sellele $B[i_2][j] - B[i_1 - 1][j]$:

```
int ans = INF;
for (int i1 = 1; i1 <= n; i1++) {
  for (int i2 = i1; i2 <= n; i2++) {
    int best = INF;
    for (int j = 1; j <= n; j++) {
      best = min(best, -B[i2][j - 1] + B[i1 - 1][j - 1];
      ans = min(ans, B[i2][j] - B[i1 - 1][j] + best);
    }
  }
}
```

6. Segane väljund (`segane`)

3 sek / 10 sek 60 punkti

Idee: Heno Ivanov, teostus ja lahenduse selgitus: Marko Tsengov

> N võrdset sõne pikkusega L on omavahel teadmata viisil põimitud. Taasta tulemuse S põhjal kõik võimalikud esialgsed sõned.

Vaatame algul lihtsuse huvides juhte, kus $N = 2$. Naiivne lahendus on määrata igale S märgile, kas ta on esimeses või teises sõnes, ning seejärel kontrollida, kas saadud sõned on võrdsed. Selle efektiivne implementatsioon lahendab ka esimese alamülesande keerukusega $O\left(\binom{2L}{L-1}\right)$.

Täislahenduse saame dünaamilise planeerimise abil. Hoiame olekuid (s, v), kus s märgib pikima sõne hetkest olekut ning v ülejäänud $N-1$ sõne pikkuseid (märkame, et kui $N = 2$, siis on selleks vaid üks arv). Algseks olekuks on $(\texttt{""}, [0, 0, ..., 0])$, märkides, et üheski sõnes pole ühtki märki. Nõuame, et sõnede pikkused oleksid mittekasvavalt järjestatud.

Teisendame neid olekuid, läbides sõne S märgihaaval. Peame kaaluma kahte eri liiki teisendusi:

1. Lisame praeguse märgi sõnele s. Seda on mõttekas teha vaid siis, kui $|s| < L$.
2. Suurendame mõne v elemendi väärtust ühe võrra. Seejuures peab hetkene märk võrduma suurendatavale elemendile vastava märgiga sõnes s (kuna muidu poleks sõned lõpuks võrdsed), samuti ei tohi ühtki väärtust suurendada üle eelmise v elemendi väärtuse (esimese elemendi puhul üle $|s|$), et mittekasvamise tingimus oleks rahuldatud.

Teostades iga oleku puhul kõiki võimalikke teisendusi ning vältides duplikaatide ilmnemist, leiab selline algoritm S läbise lõpuks parajasti kõik võimalikud algsed sõned (v väärtused on kõik L). Algoritmi keerukuse hindamiseks märkame, et võimalikke olekuid v jaoks on $O(L^N)$, s võimalused on aga rohkem piiratud, kuna mitmete samade märkide puhul tekivad identsed s, eri märkide puhul aga v väärtusi palju erinevalt suurendada ei saa, nõudes märgi lisamist sõnesse s. Katsetades näib algoritmi keerukus olevat ligikaudu $O(N \cdot L^N)$.