# Sisukord

## 1. Jõulutuled (`tuled`)

1 sekund 10 punkti

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

Selles ülesandes on esimene trikk, et tulede põlemashoidmise kulud on antud sentides, aga Juta päevane eelarve eurodes. Arvutuste lihtsustamiseks oleks hea teisendada kõik suurused samadesse ühikutesse. Selleks võib kas $X$ ja $Y$ väärtused mõlemad 100-ga jagada (siis arvutame edasi eurodes) või $N$ väärtuse 100-ga korrutada (siis arvutaksime edasi sentides).

Ülesande sisu juurde minnes peaks olema üsna selge, et maksimaalse tulede põlemisaja saamiseks on vaja võimalikult palju kasutada odavamat elektrit. Kuna pole ette teada, kas odavam on päevane või õhtune elekter, tuleb programm kirjutada nii, et see töötaks õigesti mõlemal juhul.

Oletame, et odavam on päevane elekter ($X < Y$). Siis on $N$ euro eest võimalik tulesid põlemas hoida $N/X$ tundi. Juta koolist koju jõudmise ajal on päevase elektrihinna kehtivust jäänud veel $18 - 14 = 4$ tundi. Kui $N/X < 4$ (ehk $N < 4 \cdot X$), siis saab Jutal raha otsa enne kui õhtune elektrihind kehtima hakkab ja seega vastus ongi $N/X$.

Kui $N > 4 \cdot X$, siis on Jutal õhtuse elektrihinna kehtimahakkamise ajaks ära kulunud $4 \cdot X$ eurot ja alles veel $N - 4 \cdot X$ eurot, mille eest ta saab tulesid õhtuse elektrihinna kehtimise ajal veel $(N - 4 \cdot X)/Y$ tundi lisaks põlemas hoida. Seega siis on vastus $4 + (N - 4 \cdot X)/Y$.

Kui õhtune elekter on päevasest odavam, siis saame arutada analoogiliselt. Õhtuse hinnaga oleks võimalik tulesid põlemas hoida $N/Y$ tundi. Enne Juta magamaminekut jõuab õhtune hind kehida 5 tundi. Seega, kui $N < 5 \cdot Y$, siis on vastus $N/Y$, vastasel juhul aga $5 + (N - 5 \cdot Y)/X$.

Veel viimane erijuht on võimalus, et Jutal on piisavalt raha, et ta saaks tulesid põlemas hoida isegi rohem kui oma kojujõudmise ja magamaminemise vahelised 9 tundi. Ülesande tingimuste kohaselt ta seda ei tee ja sel juhul on vastus ikkagi 9.

Pythoni programmina võib eelneva loogika kirja panna näiteks nii:

```
# loeme sisendandmed
X = int(input())
Y = int(input())
N = int(input()) * 100 # eelarve eurodest sentideks

# leiame ja väljastame vastuse
if X*4 + Y*5 < N: # raha jätkub kojujõudmisest magamaminekuni
  print(9)
elif X < Y: # päevane elekter on odavam, kasutame seda maksimaalselt
  if X*4 > N: # kogu raha kulub päevasel ajal ära
    print(N/X)
  else: # raha jätkub kõigiks 4 päevaseks tunniks
    # leiame, kui palju õhtuks üle jääb
    varu = N - X*4
    print(4 + varu/Y)
else: # õhtune elekter on odavam, kasutame seda maksimaalselt
  if Y*5 > N: # kogu raha kulub õhtusel ajal ära
    print(N/Y)
  else: # raha jätkub kõigiks 5 õhtuseks tunniks
    # leiame, kui palju päevaks üle jääb
    varu = N - Y*5
    print(5 + varu/X)
```

C++ kasutajad võiks kirjutada näiteks nii:

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

int main() {
  // loeme sisendandmed
  double x, y, n;
  cin >> x >> y >> n;
  x /= 100; y /= 100; // X ja Y sentidest eurodeks

  // leiame ja väljastame vastuse
  if (4 * x + 5 * y < n) { // raha jätkub kojujõudmisest magamaminekuni
    cout << 9 << '\n';
  } else if (x < y) { // päevane elekter on odavam, kasutame seda maksimaalselt
    if (4 * x > n) { // kogu raha kulub päevasel ajal ära
      cout << n / x << '\n';
    } else { // raha jätkub kõigiks 4 päevaseks tunniks
      // leiame, kui palju õhtuks üle jääb
      double varu = n - 4 * x;
      cout << 4 + varu / y << '\n';
    }
  } else { // õhtune elekter on odavam, kasutame seda maksimaalselt
    if (5 * y > n) { // kogu raha kulub õhtusel ajal ära
      cout << n / y << '\n';
    } else { // raha jätkub kõigiks 5 õhtuseks tunniks
      // leiame, kui palju päevaks üle jääb
      double varu = n - 5 * y;
      cout << 5 + varu / x << '\n';
    }
  }
}
```

C++ kasutajatel tuleb lisaks tähelepanu pöörata andmetüüpidele. Kuigi sisendandmed on kõik täisarvud, võivad vahetulemused ja vastused olla ka murdarvud. C++ reeglid ütlevad, et kui `A` ja `B` on mõlemad täisarvutüüpi, siis tehakse avaldises `A/B` täisarvuline jagamine. See tähendab, et vastus on ka alati täisarvutüüpi; kui matemaatiliselt oleks jagatis mingi murdav, siis visatakse murdosa lihtsalt ära. Selle vältmiseks on siin ülesandes kõige lihtsam defineerida kõik muutujad kohe reaalarvutüüpi, nagu oli tehtud ka võistluse serveris antud lahenduse mallis. C++ keeles on `double` topelttäpsusega reaalarvutüüp; kui ei ole head põhjust teha midagi muud, võiks kõigi reaalavuliste väärtuste hoidmiseks kasutada seda tüüpi. Lisainfot C++ jagamistehte riskide kohta on ka ülesande "Paberi voltimine" lahenduse seletuses.

## 2. Sokid (`sokid`)

1 sekund 20 punkti

*Idee: Tähvend Uustalu, teostus: Birgit Veldi, lahenduse selgitus: Birgit Veldi ja Ahto Truu*

Juku tahab võimalikult vähe sokke üle värvida. Seega oleks tal mõistlik alustuseks ühte värvi sokkidest võimalikult palju paare moodustada. Selleks võiks kokku lugeda, kui palju igat värvi sokke on. Siis on meil ühe värvi kohta kaks võimalust:

- kui seda värvi sokke on paarisarv, saame need omavahel paarideks jaotada;
- kui seda värvi sokke on paaritu arv, saame kõik peale ühe omavahel paarideks jaotada.

Nii jääb meil igast sellisest värvist, mida oli paaritu arv, täpselt üks sokk üle. Kui neid ülejäänud sokke on paaritu arv, siis me neid kuidagi paarideks jaotada ei saa ning väljastada tuleb $-1$. (Seda võib tegelikult ka kohe algul kontrollida: kui sokkide koguarv $N$ on paaritu, siis neid paarideks jaotada pole võimalik.) Kui ülejäänud sokke on aga paarisarv, siis saame need ükskõik kuidas paarideks jaotada ning igas saadud paaris ühe soki oma paarilisega sobivaks üle värvida.

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

Python:

```python
# loeme sokkide arvu
n = int(input())

# sokkide statistika värvide kaupa
sokid = {}
for i in range(n):
  varv = input().strip()
  if varv in sokid:
    # oleme seda värvi juba näinud,
    # suurendame loendurit
    sokid[varv] += 1
  else:
    # esimene seda värvi sokk,
    # alustame uue lenduriga 1-st
    sokid[varv] = 1

# leiame üksikute sokkide arvu
yksi = 0
for arv in sokid.values():
  # paaritu arv lisab ühe üksiku soki
  yksi += arv % 2

# väljastame vastuse
if yksi % 2 > 0:
  # üksikuid on kokku paaritu arv,
  # lahendust ei ole
  print(-1)
else:
  # peab pooled üksikud üle värvima
  print(yksi // 2)
```

C++:

```cpp
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main() {
  // loeme sokkide arvu
  int n;
  cin >> n;

  // sokkide statistika värvide kaupa
  map<string, int> sokid;
  for (int i = 1; i <= n; ++i) {
    string varv;
    cin >> varv;
    // uue värviga sokid[varv]
    // poole pöördumine paneb selle
    // algväärtuseks automaatselt 0
    sokid[varv] += 1;
  }

  // leiame üksikute sokkide arvu
  int yksi = 0;
  for (auto [varv, arv] : sokid) {
    // paaritu arv lisab ühe üksiku
    yksi += arv % 2;
  }

  // väljastame vastuse
  if (yksi % 2 > 0){
    // kui üksikuid on kokku paaritu
    // arv, siis lahendust ei ole
    cout << -1 << '\n';
  } else{
    // muidu peab pooled üle värvima
    cout << yksi / 2 << '\n';
  }
}
```

Pythoni kasutajatel on võimalik lugeda värvide statistika kokku ka standardteegi Counter objekti abil:

```
from collections import Counter

n = int(input())
loe = Counter(input().strip() for _ in range(n))

if n % 2 == 1:
  print(-1)
else:
  v = sum(c % 2 for c in loe.values())
  print(v // 2)
```

Veel üks võimalus on pidada hulgatüüpi muutujas jooksvalt arvet ainult paariliseta sokkide üle:

Python:

```
n = int(input())

yksi = set()
for i in range(n):
  varv = input().strip()
  if varv in yksi:
    yksi.remove(varv)
  else:
    yksi.add(varv)

if n % 2 > 0:
  print(-1)
else:
  print(len(yksi) // 2)
```

C++:

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

int main() {
  int n;
  cin >> n;

  set<string> yksi;
  for (int i = 1; i <= n; ++i) {
    string varv;
    cin >> varv;
    if (yksi.contains(varv)) {
      yksi.erase(varv);
    } else {
      yksi.insert(varv);
    }
  }

  if (n % 2 > 0){
    cout << -1 << '\n';
  } else{
    cout << yksi.size() / 2 << '\n';
  }
}
```

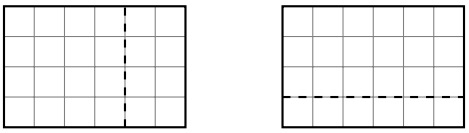
## 3. Paberi voltimine (volt)

1 sekund 30 punkti

*Idee ja teostus: Targo Tennisberg, lahenduse selgitus: Ahto Truu*

Nagu tekstis juba vihjatud, on selles ülesandes kaks põhimõtteliselt erinevat juhtu: kas murdejoon on paralleelne ruutude külgede või diagonaalidega.

Ruutude külgedega paralleelne joon omakorda võib olla horisontaalne või vertikaalne:

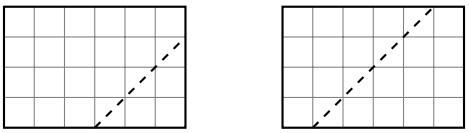


Mõlemal juhul määrab vastuse suurem neist kahest ristkülikust, milleks murdejoon algse paberi jagab.

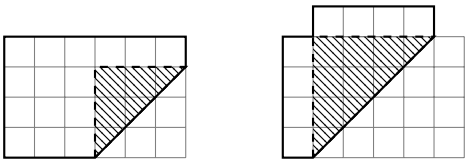
Vertikaalse murdejoone tunneme ära sellest, et $X_1 = X_2$ (lisaks teame, et $Y_1$ ja $Y_2$ on sel juhul kindlasti kas 0 ja $N$ või $N$ ja 0). Siis on tulemuse kõrgus $N$ ja laius $\max(X_1, M - X_1)$.

Sarnaselt on horisontaalse murdejoone tunnuseks $Y_1 = Y_2$ ning tulemuseks saadava ristküliku laius $M$ ja kõrgus $\max(Y_1, N - Y_1)$.

Diagonaalse murdejoone puhul võivad selle otspunktid olla kas naaberkülgedel (alloleval joonisel vasakul) või vastaskülgedel (alloleval joonisel paremal):



Võib tunduda, et siin on palju võimalikke variante, mida peaks kõiki eraldi käsitlema, aga osutub, et vastus on kõigil neil juhtudel leitav täpselt sama meetodiga!



Kui murdejoone otspunktid on naaberkülgedel, siis saab kahekordselt kaetud täisnurkse kolmnurga kujuline ala, mille laius on $|X_1 - X_2|$ ja kõrgus $|Y_1 - Y_2|$ (viirutatud ala ülaloleval joonisel vasakul) ja tulemuseks saadava kujundi pindala on algsest paberist selle võrra väiksem.

Ülesande arvatavasti raskeim osa on märgata, et ka vastaskülgedel olevate otspuktidega murdjoone korral saab kahekordselt kaetud samasugune kolmnurk (ülaloleval joonisel paremal).

Kõiki neid tähelepanekuid kombineerides on lahenduse sisuline osa lõpuks 3-realine:

```
def area(m, n, x1, y1, x2, y2):
  if x1 == x2: return max(m - x1, x1) * n
  if y1 == y2: return max(n - y1, y1) * m
  return m * n - abs((x1 - x2) * (y1 - y2) / 2)
```

Tehniline detail C++ kasutajatele: kui `x1` ja `x2` on täisarvutüüpi, siis on seda ka `x1-x2`; kui `x1-x2` ja `y1-y2` on täisarvutüüpi, siis on seda ka `(x1-x2)*(y1-y2)`; ja siis on ka jagamine `(x1-x2)*(y1-y2)/2` täisarvuline ning viskab vastuse murdosa minema. Selle vältimiseks võib:

- kirjutada jagaja reaalarvuna: `(x1-x2)*(y1-y2)/2.0` või
- teisendada jagatava enne jagamist reaalarvuks: `double((x1-x2)*(y1-y2))/2`.

Teisel juhul on sulgude paigutus väga oluline! Avaldis `double((x1-x2)*(y1-y2)/2)` teeb kõigepealt täsarvulise jagamise (koos murdosa äraviskamisega) ja teisendab siis selle tulemuse reaalarvutüüpi; juba minema visatud murdosa see enam tagasi ei too.

## 4. Koogid (kook)

1 sekund 40 punkti

*Idee ja teostus: Kregor Ööbik, lahenduse selgitus: Ahto Truu*

Peaks olema üsna ilmne, et kui lahendus leidub, siis peavad suuremad koogid minema rohkemate päkapikkudega laudadele. Kõige lihtsam seda saavutada on sorteerida nii koogid kui lauad suuruste järjekorda. Kuna pärast on vaja teada ka laudade ja kookide numbreid, siis on parem lihtsalt suuruste asemel hoida ja sorteerida (suurus, indeks) paare:

```
p = [int(x) for x in input().split()]
p = sorted((x, i) for i, x in enumerate(p))
```

Selleks, et kontrollida, kas kõik päkapikud saavad võrdse hulga kooki, võib võrrelda näiteks kõiki teisi (kook, laud) paare esimese (kook, laud) paariga. Ümardamisvigade vältimiseks on parem tingimuse

$$\frac{M_a}{P_a} = \frac{M_b}{P_b}$$

asemel kontrollida tingimust

$$M_a \cdot P_b = M_b \cdot P_a \ ,$$

sest siis on kõik arvutused täisarvudega:

```
saab = True
for i in range(1, n):
  if m[i][0] * p[0][0] != m[0][0] * p[i][0]:
    saab = False
```

Kui lahendus leidub, siis on kookide laudadele jaotuse leidmiseks üks võimalus käia kookide ja laudade jadad paralleelselt läbi ja panna iga koogi indeksi kohta kirja selle laua indeks:

```
a = [-1] * n
for i in range(n):
  a[m[i][1]] = p[i][1] + 1
print(*a)
```

Teine võimalus on koguda kokku kookide indeksid (kookide suuruse järjekorras) ja laudade indeksid (laudade suuruse järjekorras) ja sorteerida need paarid omakorda kookide indeksite järgi:

```
a = [(m[i][1], p[i][1]) for i in range(n)]
print(*(p + 1 for m, p in sorted(a)))
```

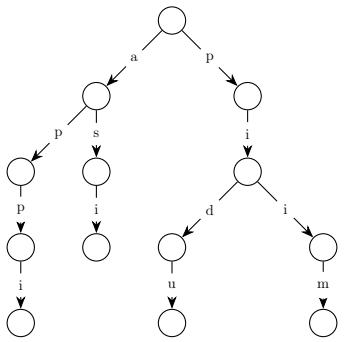
## 5. Kingituste andmebaas (baas)

5 sek / 10 sek 60 punkti

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

Olgu antud mingi nimekiri sõnedest. *Trie'ks* nimetatakse andmestruktuuri, milles on tipp iga sõne kohta, mis on mõne nimekirjas oleva sõne prefiks. Sõnele $s$ vastavast tipust on tähega $c$ märgistatud serv sõnele $t$ vastavasse tippu, kui sõnele $s$ tähe $c$ lõppu lisamise annab sõne $t$.

Näiteks vastab nimekirjale "piim", "pidu", "asi", "appi" alltoodud trie.




Üks viis trie ja sellesse sõne lisamise realiseerimiseks on järgmine:

```
struct Node { // tipp
  map<char, Node> next;
};

void add_word(Node & root, const string & word) {
  Node *curr = &root;
  for (auto c : word) {
    // next[c] poole pöördumine lisab uue elemendi,
    // kui seda veel olemas ei ole
    curr = &curr->next[c];
  }
}
```

Mõtleme ülesandest nii: tries on sõnele vastav tipp parajasti siis, kui ta on prefiks vähemalt ühele sõnale, millest trie moodustatud on. Kui me ehitame iga lapse kingisoovidest trie, siis otsime pikimat sõnet, millele vastav tipp leidub igas tries. Teisisõnu otsime sügavaimat tippu kõikide triede ühisosas.

Kahe trie ühisosa saab leida näiteks nii:

```
// märk '&' tüübi järel tähendab, et funktsioon saab argumendiks antud
// struktuuri aadressi ja andmetest koopiat ei tehta
Node intersect(Node & a, Node & b) {
  Node res;
  for (auto & [c, _] : a.next) {
    // käime läbi kõik tipu a alluvad; kui tipul b on ka sama tähega alluv,
    // siis arvutame rekursiivselt nende alampuude ühisosa ja lisame selle
    // ühisosa-triesse
    if (b.next.contains(c)) {
      res.next[c] = intersect(a.next[c], b.next[c]);
    }
  }
  return res;
}
```

Tähelepanu tuleb pöörata ka mälulimiidile. Kuigi see on antud ülesandes 512 MB — suurem kui harilik 256 MB — võib hooletu lahendus ka seda ületada. Sisendi maksimaalne maht on 64 MB ehk kingisoovide pikkuste summa võib olla kuni $64 \cdot 2^{20}$. Seega, kui iga lapse kingisoovidest ehitada trie ja kõik need tried samaaegselt mälus on, võib nendel triedel ka kokku olla $64 \cdot 2^{20}$ tippu. Kui iga tipp võtab näiteks 10 baiti, siis ongi juba 640 MB kulutatud (ja 512 MB piir ammu ületatud). Üks lahendus on alati hoida meeles globaalset ühisosa-triet, aga iga lapse triet vaid ühes iteratsioonis.

```
Node file_trie;
for (int i = 0; i < n; ++i) {
  // line_trie on deklareeritud ploki sees - see tähendab, et iga line_trie
  // kustutatakse kohe, kui iteratsioon on läbi, ja järgmiseks iteratsiooniks
  // tehakse uus tühi trie
  Node line_trie;
  for (int j = 0; j < m; ++j) {
    string word;
    cin >> word;
    add_word(line_trie, word);
  }
  if (i == 0) {
    file_trie = line_trie;
  } else {
    file_trie = intersect(file_trie, line_trie);
  }
}
```

*Alternatiivne lahendus: Marko Tsengov*

Oletame, et me teame päringuks olnud sõne pikkust $\ell$. Sellisel juhul saame tulemused rida-rea haaval läbi käia ning hoida meeles, millised võimalikud päringud eelnevate ridade põhjal olla võisid. Iga uue rea puhul saame leida päringud, mis just sellele reale oleks sobinud, ning leida kõigi seni sobinud (iga sõne vastava pikkusega prefiks) ja sellele reale sobinud päringute ühisosa. Sellise jooksva ühisosa meeles hoidmiseks sobib hästi hulga (*set*) andmestruktuur, kust saab kiiresti (keerukusega $O(1)$ või $O(\log N)$) elemente pärida ning kustudada.

Et esimesel real on $M$ sõnet, peame korraga meeles hoidma kuni $M$ erinevat päringut, kuna ridade kaupa edasi minnes saab sobivaid päringuid ainult vähemaks jääda. Ühisosa uuendamiseks võime kõik eelnevate ridade jaoks sobinud päringud läbi käia ning kontrollida, kas need sobivad ka uuele reale. Et sel real oli niikuinii juba $M$ sõnet, mille sisselugemiseks kulus $M$ operatsiooni, pole nende kõigi kohta hulka kuuluvuse kontrollimine sellega võrreldes kuigi aeganõudev. Seega saame sobivad päringud leida kogu andmestikku vaid ühe korra läbides.

Siiski peame veel parandama asjaolu, et me ei tea $\ell$ väärtust. Märkame, et kui pakume reaalsest väärtusest suurema pikkuse, siis eelolev algoritm töötab, kuid lõpuks ei leia ühtki sobivat päringut. Samuti märkame, et kui pakume reaalsest väärtusest väiksema pikkuse, siis sama algoritm endiselt töötab, tagastades õige päringu vastava pikkusega prefiksi.

Eeltoodud omaduste põhjal saame $\ell$ väärtuse leida kahendotsingu abil või — kuna lubatud sõnede pikkus on võrdlemisi väike — ka iteratiivselt, proovides järjest suuremaid väärtusi, kuni sobivat päringut enam ei leidu.

## 6. Tasakaalus jõulusaan (saan)

2 sek / 10 sek 100 punkti

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

Alustel $A_1, A_2, \ldots, A_K$ olevate pakkide massikese on $(A_1, A_2, \ldots, A_K)/K$; saani keskpunkt on $(N+1)/2$. Seega on saan tasakaalus, kui

$$\frac{A_1 + A_2 + \cdots + A_K}{K} = \frac{N+1}{2}$$

ehk

$$A_1 + A_2 + \cdots + A_K = \frac{K(N+1)}{2}.$$

Näeme, et kui $N$ on paaris ja $K$ on paaritu, siis lahendit olla ei saa: vasak pool on täisarv, parem mitte. Muudel juhtudel leidub vähemalt mingigi tasakaalus konfiguratsioon: pooled kastid vasakusse serva, pooled paremale, kasutades vajaduse korral ka keskmist alust.

Lahendame ülesande dünaamilise plaanimise abil. Selleks on palju variante, esitame siinkohal ühe. Keskendume minimaalse käikude arvu leidmisele — võimaluste arvu loendamine tuleb sellest lahendusest loomulikul teel välja. Pakkide ümbertõstmise asemel mõtleme pakkide lisamisest ja eemaldamisest: käime alused vasakult paremale läbi; kaalume igalt aluselt paki eemaldamist (kui sellel hetkel on pakk) ja paki lisamist (kui sellel hetkel pakki ei ole). Lõpplahendis peavad lisatud pakkide ja eemaldatud pakkide arvud muidugi võrdsed olema.

Hetke olekut peegeldab kolmik $(i, j, S)$, kus:

- $i$ on läbi vaadatud aluste arv;
- $j$ on eemaldatud pakkide arvu ja lisatud pakkide arvu vahe;
- $S$ on alustel $1 \ldots i$ olevate pakkide asukohtade summa.

Tähistagu $\mathrm{dp}[i][j][S]$ minimaalset eemaldatud pakkide arvu (või samaväärselt, lisatud pakkide arvu). Alguses seame $\mathrm{dp}[0][0][0] = 0$ ja iga muu kolmiku korral $\mathrm{dp}[i][j][S] = \infty$. Nüüd arvutame tabeli varasemate väärtuste põhjal välja hilisemad väärtused.

- Iga $i = 1, \ldots, n$ korral:
  - Iga $j = -n, \ldots, n$ korral:
    - Iga $S = 0, \ldots, \frac{n(n+1)}{2}$ korral:
      - Kui alusel $i$ on pakk:
        - Seame $\mathrm{dp}[i][j][S+i] \leftarrow \min(\mathrm{dp}[i][j][S+i], \mathrm{dp}[i-1][j][S])$. (Kaalume mitte millegi tegemist ehk paki alusele jätmist. See viiks meid olekust $(i-1, j, S)$ olekusse $(i, j, S+i)$: ühtegi pakki ei eemaldatud ega lisatud, aga pakkide asukohtade summa, mis arvestab nüüd ka alusega $i$, kasvab $i$ võrra. Miinimumi võtame sellepärast, et tabelis võib juba ees olla parem lahend sama oleku jaoks.)
        - Seame $\mathrm{dp}[i][j+1][S] \leftarrow \min(\mathrm{dp}[i][j+1][S], \mathrm{dp}[i-1][j][S])$. (Kaalume aluselt paki eemaldamist; see viiks meid olekust $(i-1, j, S)$ olekusse $(i, j+1, S)$ — eemaldatud pakkide arv kasvab ühe võrra, asukohtade summa jääb samaks.)
      - Kui alusel $i$ pakki pole:
        - Seame $\mathrm{dp}[i][j][S] \leftarrow \min(\mathrm{dp}[i][j][S], \mathrm{dp}[i-1][j][S]$ (kaalume mitte millegi tegemist).
        - Seame $\mathrm{dp}[i][j-1][S+i] \leftarrow \min(\mathrm{dp}[i][j-1][S+i], \mathrm{dp}[i-1][j][S]+1)$ (kaalume alusele paki panemist).

Lõppvastus on tabeli lahtris $\mathrm{dp}[N][0][K(N+1)/2]$. Selline lahendus muidu töötab, aga siin on mõned nüansid.

Esiteks on $91 \times 181 \times 4096$ tabel päris suur. Mälulimiiti mahub see küll, aga nii suure `array` või `vector` loomine võib ebaõnnestuda. Selle probleemi lahendamiseks piisab tähele panna, et meil ei ole vaja tervet tabelit meeles hoida. Vaja on ainult praeguse $i$ ja $i-1$ tabelit.

Teiseks on vaja tabelis kasutada negatiivseid indekseid. Selleks on mitu moodust; žürii lahenduses on see lahendatud nii:

```
template<typename T>
class OffsetArray {
  int offset;
  vector<T> vec;

public:
  OffsetArray (int _size, int _offset, T init) :
    offset(_offset), vec(_size, init) { }

  T& operator[] (int index) {
    return vec[offset + index];
  }
};
```

Siis `OffsetArray<int>(n, m, 0)` loob massiivilaadse toote, mis on alguses täidetud nullidega ja milles on kasutatavad indeksid $-m \ldots n-m-1$.

Esimene pool ülesandest — tõstmiste arvu minimeerimine — on seega lahendatud keerukusega $O(N^4)$. Siia tuleb nüüd lisada võimaluste loendamine. Selleks peame iga oleku kohta meeles, mitu võimalust on minimaalse sammude arvuga selle oleku juurde jõuda. Teeme teise tabeli: $\mathrm{ways}[i][j][S]$ loendab, mitu võimalust $\mathrm{dp}[i][j][S]$ käiguga olekusse $(i, j, S)$ jõudmiseks on juba leitud. Seda tabelit uuendame koos tabeliga dp. Oletame näiteks, et alusel $i$ on pakk ja kaalume mitte millegi tegemist. Ülalolev algoritm teeb siis $\mathrm{dp}[i][j][S+i] \leftarrow \min(\mathrm{dp}[i][j][S+i], \mathrm{dp}[i-1][j][S])$. Enne seda uuendame tabelit ways:

- kui $\mathrm{dp}[i-1][j][S] < \mathrm{dp}[i][j][S+i]$, siis **seame** $\mathrm{ways}[i][j][S+i]$ väärtuseks $\mathrm{ways}[i-1][j][S]$;
- kui $\mathrm{dp}[i-1][j][S] = \mathrm{dp}[i][j][S+i]$, siis **liidame** $\mathrm{ways}[i][j][S+i]$ väärtusele $\mathrm{ways}[i-1][j][S]$;
- kui $\mathrm{dp}[i-1][j][S] > \mathrm{dp}[i][j][S+i]$, siis ei tee me midagi.