Sisukord

1. Iidne kood **2**

2. Voltimine **5**

3. Elementidega kirjutamine **6**

4. Basseini täitmine **7**

5. Lillepood **8**

6. CC **9**

7. Lahtipakkimine **12**

1. Iidne kood (kood)

1 sek / 3 sek 20 punkti

Idee: Phil Labuschagne; teostus ja lahenduse selgitus: Ahto Truu

Selle ülesande lahendamiseks võiks kõigepealt joonisel oleva algoritmi oma programmikeeles välja kirjutada:

Python:

```
while n != 1:
  if n % 2 == 0:
    n = n // 2
  else
    n = 3 * n - 1
```

C++:

```
while (n != 1) {
  if (n % 2 == 0) {
    n = n / 2;
  } else {
    n = 3 * n - 1;
  }
}
```

Paar tehnilist märkust eelnevate koodinäidete kohta:

- N paarisarvulisuse kontrollimiseks leiame jäägi N jagamisest kahega ja vaatame, kas see on null. Suuremas osas tänapäeva programmikeeltes (Python, C, C++, C#, Java jne) on jäägi leidmise tehtemärk '`%`', aga mõnes võib see olla ka midagi muud (näiteks Pascalis peaks kirjutama '`if n mod 2 = 0`').
- Pythonis on kaks erinevat jagamistehet: '`/`' tähendab jagamist, mille tulemus on reaalarv ($6/2 \to 3{,}0$; $5/2 \to 2{,}5$); '`//`' tähendab jagamist, mille tulemus on täisarv (kui arvud täpselt ei jagu, ümardatakse tulemus allapoole: $5//2 \to 2$, $-5//2 \to -3$). Selles ülesandes on meil teada, et jagamisi tehakse ainult siis, kui N väärtus kahega jagub, seega annaks mõlemad tehted tulemuseks sama arvu, kuigi üks reaalarvu ja teine täisarvu tüüpi väärtusena.
- C-perekonna keeltes (Java sealhulgas) on ainult üks jagamistehe ja selle käitumine sõltub operandide tüüpidest: kui jagatav ja jagaja on mõlemad täisarvulist tüüpi, tehakse alati täisarvuline jagamine ($6/2 \to 3$; $5/2 \to 2$); kui vähemalt üks operandidest on reaalarvulist tüüpi, tehakse reaalarvuline jagamine ($6{,}0/2 \to 3{,}0$; $5/2{,}0 \to 2{,}5$). Selles ülesandes tehakse jagamisi ainult siis, kui N väärtus on paarisarv, seega on täisarvuline jagamine sobiv käitumine; kui meil oleks vaja kahe täisarvu tüüpi väärtuse vahel reaalarvulist jagamist, tuleks vähemalt üks neist enne jagamistehte tegemist reaalarvuks teisendada.

Esimese alamülesande jaoks (kus on teada, et N kindlasti jõuab väärtuseni 1) piisab, kui lisada sammude arvu loendamine:

Python:

```
# loeme sisendi
n = int(input())

# leiame vastuse
samme = 1
while n != 1:
  if n % 2 == 0:
    n = n // 2
  else:
    n = 3 * n - 1
  samme += 1

# väljastame vastuse
print(samme)
```

C++:

```
// loeme sisendi
int n;
cin >> n;

// leiame vastuse
int samme = 1;
while (n != 1) {
  if (n % 2 == 0) {
    n = n / 2;
  } else {
    n = 3 * n - 1;
  }
  samme += 1;
}

// väljastame vastuse
cout << samme << '\n';
```

Teise alamülesande jaoks on üks võimalus lisada eelmisele lahendusele kontroll, mis väljub kordusest, kui sammude arv ületab 100:

Python:

```
# loeme sisendi
n = int(input())

# leiame vastuse
samme = 1
while n != 1:
  if n % 2 == 0:
    n = n // 2
  else:
    n = 3 * n - 1
  samme += 1
  if samme > 100:
    break

# väljastame vastuse
if samme > 100:
  print(-1)
else:
  print(samme)
```

C++:

```
// loeme sisendi
int n;
cin >> n;

// leiame vastuse
int samme = 1;
while (n != 1) {
  if (n % 2 == 0) {
    n = n / 2;
  } else {
    n = 3 * n - 1;
  }
  samme += 1;
  if (samme > 100) {
    break;
  }
}

// väljastame vastuse
if (samme > 100) {
  cout << -1 << '\n';
} else {
  cout << samme << '\n';
}
```

Teine võimalus täislahenduse saamiseks on kirjutada kordus 100 sammu läbimiseks ja väljuda sellest enneaegselt, kui N saab väärtuseks 1:

Python:

```
# loeme sisendi
n = int(input())

# leiame vastuse
samme = 1
while samme <= 100:
  if n == 1:
    break
  if n % 2 == 0:
    n = n // 2
  else:
    n = 3 * n - 1
  samme += 1

# väljastame vastuse
if samme <= 100:
  print(samme)
else:
  print(-1)
```

C++:

```
// loeme sisendi
int n;
cin >> n;

// leiame vastuse
int samme = 1;
while (samme <= 100) {
  if (n == 1) {
    break;
  }
  if (n % 2 == 0) {
    n = n / 2;
  } else {
    n = 3 * n - 1;
  }
  samme += 1;
}

// väljastame vastuse
if (samme <= 100) {
  cout << samme << '\n';
} else {
  cout << -1 << '\n';
}
```

Kolmas võimalus on kirjutada mõlemad tingimused korduse päisesse:

Python:

```
# loeme sisendi
n = int(input())

# leiame vastuse
samme = 1
while n != 1 and samme <= 100:
  if n % 2 == 0:
    n = n // 2
  else:
    n = 3 * n - 1
  samme += 1

# väljastame vastuse
if samme <= 100:
  print(samme)
else:
  print(-1)
```

C++:

```
// loeme sisendi
int n;
cin >> n;

// leiame vastuse
int samme = 1;
while (n != 1 and samme <= 100) {
  if (n % 2 == 0) {
    n = n / 2;
  } else {
    n = 3 * n - 1;
  }
  samme += 1;
}

// väljastame vastuse
if (samme <= 100) {
  cout << samme << '\n';
} else {
  cout << -1 << '\n';
}
```

Pythonis on veel võimalus kasutada selle keele spetsiifilist `for-else` konstruktsiooni, kus `for`-korduse juurde lisatud `else`-plokis olevad käsud täidetakse parajasti siis, kui kordus töötas lõpuni (st sellest ei väljutud `break`-käsuga):

```
# loeme sisendi
n = int(input())

# leiame vastuse
for samme in range(100):
  if n == 1:
    # väljastame positiivse vastuse ja katkestame
    print(samme + 1)
    break
  if n % 2 == 0:
    n = n // 2
  else:
    n = 3 * n - 1
else:
  # väljastame negatiivse vastuse
  print(-1)
```

Selles lahenduses võiks '`for samme in range(100)`' asemel kirjutada ka '`for samme in range(1, 101)`'; siis saaks muutuja `samme` väärtusteks $0 \ldots 99$ asemel $1 \ldots 100$ ja saaksime vastuse väljastada käsu '`print(samme + 1)`' asemel käsuga '`print(samme)`'. Kumb variant on lihtsam ja selgem, sõltub kõige rohkem sellest, kui palju on vaja muutuja `samme` väärtustega mingeid arvutusi teha.

2. Voltimine (`volt`)

1 sek / 3 sek 30 punkti

Idee: Rocco Liiva; teostus ja lahenduse selgitus: Madis Roosma

Ülesande lahendus põhineb tähelepanekul, et x ja y teljed on ülesandes sõltumatud. Teisisõnu võime me kõigepealt keskenduda ülesande lahendamisele x telje jaoks ja seejärel analoogselt lahendada ülesande y telje jaoks.

Vaatame siis esialgu, mis juhtub x teljega. Algne paber katab lõiku $[0, X]$ ja meie soovime, et ta kataks vaid lõiku $[X_1, X_2]$.

Hoiame jooksvalt meeles vahemiku $[L, R]$, mida paber hetkel katab, ja arvutame selle lõigu keskpunkti $M = (L + R)/2$.

- Kui $L = X_1$ ja $R = X_2$, oleme soovitud vahemiku saavutanud.
- Kui $X_2 \leqslant M$, asub sihtvahemik täielikult vasakus pooles. Voldime parema poole vasakule (`p`) ja võtame $R = M$.
- Kui $X_1 \geqslant M$, asub sihtvahemik täielikult paremas pooles. Voldime vasaku poole paremale (`v`) ja võtame $L = M$.
- Vastasel juhul ulatub sihtvahemik mõlemale poole keskpunkti ning lahendus puudub.

Kasutame analoogset algoritmi ka y telje voltimiseks.

- Kui $L = Y_1$ ja $R = Y_2$, oleme soovitud vahemiku saavutanud.
- Kui $Y_2 \leqslant M$, asub sihtvahemik täielikult alumises pooles. Voldime ülemise poole alla (`u`) ja võtame $R = M$.
- Kui $Y_1 \geqslant M$, asub sihtvahemik täielikult ülemises pooles. Voldime alumise poole üles (`a`) ja võtame $L = M$.
- Vastasel juhul ulatub sihtvahemik mõlemale poole keskpunkti ning lahendus puudub.

Vastuse saamiseks piisab mõlema telje tulemuste kokku panemisest. Kui vähemalt ühte telge polnud võimalik lahendada, siis väljastame -1, muidu loeme kokku, mitu voltimist me tegime, väljastame voltimiste arvu ja siis välja arvutatud voltimiste jadad. Algoritmi keerukus on $O(\log(X)+\log(Y))$: iga voltimise järel on vastava külje pikkus kaks korda väiksem, mis tähendab, et me teeme ülimalt $\log(X)$ voltimist x teljel ja ülimalt $\log(Y)$ voltimist y teljel.

3. Elementidega kirjutamine (`elem`)

1 sek / 3 sek 40 punkti

Idee, teostus ja lahenduse selgitus: Targo Tennisberg

Lahenduse idee

Esimeses alamülesandes on kõik tähised ühetähelised. Piisab kontrollist, kas sõna iga tähe jaoks on sobiv element olemas. Kui on, siis on vastuseks sõna pikkus ja suurte tähtedega kirjutatud sõna.

Üldjuhul võib tekkida mõte valida alati pikim sobiv elemenditähis. See ei tööta: tähiste `A`, `Ab` ja `Bc` korral jätab sõna `abc` alguses `Ab` valimine järele võimatu lõpu `c`, kuigi esitus `ABc` on võimalik. Lühikeste sõnade korral saame rekursiivselt proovida kõiki lubatud ühe- ja kahetähelisi valikuid ning jätta meelde parima esituse. Pikkade sõnade jaoks on selline läbivaatus aga liiga aeglane, sest samu sõnaosi lahendatakse korduvalt.

Efektiivselt töötav idee on kasutada dünaamilist planeerimist. Olgu sõna pikkus L ja selle tähtede indeksid $0, \ldots, L-1$. Tähistame D_i-ga vähimat elementide arvu, millega saab esitada indeksist i algava sõna lõpuosa. Tühja lõpu jaoks $D_L = 0$; võimatut esitust tähistame väärtusega ∞.

Arvutame väärtused paremalt vasakule. Algul võtame $D_i \leftarrow \infty$. Seejärel vaatame läbi kaks võimalust:

- Kui täht w_i on lubatud ühetäheline tähis, võtame $D_i \leftarrow \min(D_i, 1 + D_{i+1})$.
- Kui $i+1 < L$ ja tähepaar $w_i w_{i+1}$ on lubatud kahetäheline tähis, võtame $D_i \leftarrow \min(D_i, 1 + D_{i+2})$.

Võimatu jätkuga valikuid ei kasutata.

Iga esitus peab algama ühega neist kahest valikust. Kuna lühemate lõpuosade parimad vastused on juba teada, annab nende võimaluste miinimum ka praeguse lõpuosa parima vastuse. Kui $D_0 = \infty$, väljastame `-1`; muidu on D_0 otsitav miinimum.

Vastuse taastamine ja keerukus

Esituse taastamiseks alustame indeksist $i = 0$. Kui ühetäheline tähis on lubatud ja $D_i = 1 + D_{i+1}$, väljastame selle ning liigume ühe tähe võrra edasi. Vastasel juhul kasutame kahetähelist tähist ja liigume kaks tähte edasi. Viimane valik peab olema võimalik ja optimaalne, sest praegune lõpuosa on esitatav ning ühetäheline valik optimaalset tulemust ei anna. Tähise esimese tähe väljastame suurena ja teise, kui see leidub, väikesena. Kui mõlemad valikud annavad miinimumi, sobib ükskõik kumb neist.

Lubatud tähiseid saab hoida kahes tõeväärtuste massiivis: ühes on 26 ühetähelise ja teises $26 \cdot 26$ kahetähelise tähise olemasolu. Nii võtab iga valiku kontrollimine konstantse aja. Ühe sõna töötlemine võtab $O(L)$ aega ja $O(L)$ mälu. Kui tähiseid on N ja sõnade kogupikkus on S, on kogu ajakulu $O(N + S)$.

4. Basseini täitmine (bass)

1 sek / 2 sek 60 punkti

Idee, teostus ja lahenduse selgitus: Tähvend Uustalu

Et basseini sügavus on igas suunas monotoonne ja vett hakatakse pumpama sügavaimast punktist, siis teame, et veetase basseinis on igal pool sama. Ajahetkel, kus veega on kaetud k ruutu, kasvab veetase kiirusega $\frac{1}{k}$ millimeetrit sekundis.

Vaatleme näitena esimest ülesande tekstis olevat näidet. Mõõdame veetaset sügavaima ruudu põhjast.

- Alguses katab vesi vaid ruutu $(2;4)$. Veetase kasvab kiirusega $1\ \frac{\text{mm}}{\text{s}}$.
- Ajahetkel $t = 3$ kattuvad veega ruudud $(2;3)$ ja $(1;4)$. Veetase hakkab kasvama kiirusega $\frac{1}{3}\ \frac{\text{mm}}{\text{s}}$.
- Ajahetkel $t = 3 + 3 = 6$ kattub veega ruut $(1;3)$. Veetase hakkab kasvama kiirusega $\frac{1}{4}\ \frac{\text{mm}}{\text{s}}$.
- Ajahetkel $t = 6 + 4 = 10$ kattub veega ruut $(2;2)$. Veetase hakkab kasvama kiirusega $\frac{1}{5}\ \frac{\text{mm}}{\text{s}}$.
- Ajahetkel $t = 10 + 5 = 15$ kattuvad veega ülejäänud kolm ruutu. Veetase hakkab kasvama kiirusega $\frac{1}{8}\ \frac{\text{mm}}{\text{s}}$.

Samal põhimõttel saame välja arvutada ajahetke, mil iga ruut veega kattub. Sorteerime ruudud sügavuse järgi kahanevalt; olgu nende sügavused $d_1, d_2, \ldots, d_{NM}$. Siis sügavuse järjekorras $k+1$. ruut kattub veega hetkel $t_{k+1} = t_k + k \cdot (d_k - d_{k-1})$, ning alates sellest hetkest hakkab veetase kasvama kiirusega $\frac{1}{t_{k+1}}\ \frac{\text{mm}}{\text{s}}$. Hoiame selle ajahetkede nimekirja meeles; teame, et see on kasvav.

Kui meile esitatakse päring ajahetke T kohta, peame esiteks leidma veetaseme, mis basseinis sel hetkel oli. Selleks leiame oma nimekirjast suurima t_k nii, et $t_k \leqslant T$. Siis teame, et ajahetkel T kasvas veetase kiirusega $\frac{1}{k}\ \frac{\text{mm}}{\text{s}}$. Ajahetkel t_k oli veetase $d_1 - d_k$, järelikult hetkel T on veetase $d_1 - d_k + \frac{T - t_k}{k}$. Nüüd piisab meil lahutada sellest vastusest päritava ruudu $(X;Y)$ kõrgus ja vastus ongi käes.

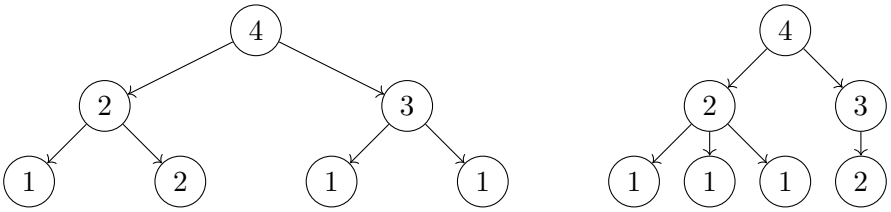
Kui me iga T kohta vaataksime käsitsi nimekirjast suurima t_k nii, et $t_k \leqslant T$, siis kuluks meil iga päringu peale $O(NM)$ aega, mis on liiga aeglane. Efektiivne viis seda teha on kasutada kahendotsingut (mille jaoks läheb vaid $O(\log(NM))$ aega). Seejuures ei ole tarvis kahendotsingut ise implementeerida: keeles C++ teeb töö ära funktsioon `std::upper_bound` või `std::lower_bound`; Pythonis `bisect.bisect_left` või `bisect.bisect_right`.

5. Lillepood (`lill`)

1 sek / 3 sek 100 punkti

Idee: Sona Kravtšenko; teostus: Andres Alumets; lahenduse selgitus: Tähvend Uustalu

Lihtsaim viis seda ülesannet lahendada on *räsimine* (ingl *hashing*): arvutada iga alampuu kohta välja mingi selline väärtus, mis sarnaste alampuude korral oleks sama, aga erinevate alampuude korral väga suure tõenäosusega erinev. Seejuures tuleb arvestada ülesande tekstis oleva tingimusega: tippude järjekord ega asetus ei loe, loeb vaid tase. Näiteks alloleval joonisel olevate puude räsid peaksid tulema võrdsed. Tippudes olevad arvud tähistavad õie tüüpi.



Üks võimalik räsimise viis on järgmine. Valime välja, näiteks juhuslikult, arvud b ja $c_1, c_2, \ldots, c_{10^5}$. Siis tipu u alampuu räsi $H(u)$ on summa $b^k c_{F_v}$ üle kõikide u alamate v, kus k on tipu v sügavus tipu u suhtes. Et arvud liiga suureks ei lähe, võtame kogu asja mingi suure täisarvulise mooduli järgi.

Näiteks kui võtta $c_1 = 123$, $c_2 = 45$, $c_3 = 67$ ja $c_4 = 89$, siis ülaloleva puu räsi tuleb

$$\begin{aligned} &89 && \text{(1. tase)}\\ &+45 \cdot b + 67 \cdot b && \text{(2. tase)}\\ &+123 \cdot b^2 + 45 \cdot b^2 + 123 \cdot b^2 + 123 \cdot b^2. && \text{(3. tase)} \end{aligned}$$

On võimaik veenduda, et sellel räsil on ülesande tekstis soovitud omadused: alamale v vastavas liidetavas $b^k c_{F_v}$ ei ole "informatsiooni" selle kohta, mis järjekorras ta vastaval tasemel on, ega ka selle kohta, millised tipud tema ja u vahel on. Et liidetavate järjekord ei mõjuta summat, siis ei mõjuta ka tippude järjekord räsi, nagu vaja.

Iga alampuu räsi efektiivseks arvutamiseks paneme tähele, et tipu u alampuu räsi on

$$H(u) = c_{F_u} + b \cdot (H(v_1) + H(v_2) + \cdots + H(v_k)),$$

kus $v_1, v_2, \ldots, v_k$ on tipu u lapsed. Need väärtused saame $O(1)$ ajas mööda puud "alt üles" liikudes välja arvutada.

Ülesande päringutele vastamiseks piisab nüüd arvutada iga küsimuse kohta "küsimuse räsi" $\sum b^j c_{T_{i,j,k}}$ ja kontrollida, kas puu alampuude seas on selline, millel on sama räsi.

6. CC (cc)

5 sek / 10 sek 100 punkti

Idee, teostus ja lahenduse selgitus: Jaagup Tamme

Vaatame erinevaid lahendusideid alustades lihtsamatest ja toome välja nende plussid ja miinused.

Alati jätmine

Kõige lihtsam ja riskivabam strateegia on pärast vooru alguses saadud esimest kaarti kohe valida jätmine ja voor lõpetada. Nii ei pea lõhki minemist üldse kartma ja igas voorus teenime punkte. Küll aga teenime igas voorus nii üsna vähe punkte, umbes 8 ja iga mängu võitmiseks kulub ligikaudu 25 vooru.

Alati võtmine, näiteks 7 kaardini

Kui alati jätmine tõi igas voorus vähe punkte, siis teine äärmus oleks alati võtta. On selge, et kuna unikaalseid kaarte on kokku 14, siis võttes voorus rohkem kui 14 kaarti on lõhki minemine kindel. Kuid kuna pakk on segatud juhuslikult, siis on väga väike tõenäosus saada järjest 14 erinevat kaarti.

Natuke mõistlikum oleks voorus proovida saada kokku 7 kaarti. Selgituseks, ülesande idee tuli kaardimängust *Flip 7*, kus on rohkem erikaarte ning eraldi reegel, et kui voorus kogutakse 7 numbrikaarti, siis voor lõppeb ja saab 15 punkti vooru eest lisaks. Kuid tõenäosus saada voorus 7 erinevat kaarti on umbes 10%. Seega keskmiselt üheksas voorus kümnest läheksime lõhki, kuid ühes voorus saaksime umbes 40–60 punkti. Ehk siis tänu suurele hulgale lõhkiminekutele saaksime voorus keskmiselt 4–6 punkti, mis on halvem kui alati jätmine.

3 kaarti voorus

Kui igas voorus üritada võtta täpselt 3 kaarti, siis lõhki minemise tõenäosus on umbes 25% ja keskmiselt kogume iga vooruga 18 punkti. Seega iga mängu võitmiseks kulub umbes 12 vooru. Lahendusidee miinus tuleb välja näiteks siis, kui käes on kaardid 1, 2 ja 3 või hoopis 10, 11 ja 12. Intuitiivselt esimesel juhul võiksime julgemalt järgmise kaardi võtta kui teisel juhul, kuid lahendus võtab mõlemat olukorda võrdsena.

Võtta, kui numbrikaartide summa on väiksem kui 20/25

See on ilmselt kõige lähedasem tavalise *Flip 7* mängija strateegiale. Ehk siis liidame enda ees olevad numbrikaardid kokku ja kui hetkel on summa vähem kui 20, siis võtame, muidu jätame. Programmi kirjutades on muidugi raske hinnata, mis on kõige parem konstant võtmise ja jätmise vaheliseks piiriks. Kuid parima konstandi saab katseliselt leida, varieerides konstanti ja tehes igaga neist esituse. Siiski on ka selle idee probleem liigne lihtustamine. Kui käes on kaardid 1, 2, 3, 4, 5 ja 6 või 10 ja 11, siis mõlemal juhul on summa võrdne, kuid teisel juhul on lõhkiminemise tõenäosus suurem.

Kaardi võtmise statistiline keskväärtus

Järgmine lahendusidee on statistiliselt hinnata, kas järgmist kaarti võttes oodatav skoor läheb paremaks kui hetkel on või mitte. Selle jaoks võib proovida läbi kõik kaardid, leida tekkinud käe skoor arvestades uut kaarti ja lõpuks võtta kõikide saadud käte keskmine tulemus. Kui kaarti võttes saame keskmiselt parema tulemuse kui hetkel, siis võtame kaardi, vastasel juhul jätame.

Vaatame näiteks olukorda, kus käes on kaardid 9 ja 10. Praeguse käe skoor on 19. Kõik võimalikud järgmised kaardid annavad järgmised tulemused:

Järgmine kaart	Käsi	Käe skoor	Kaartide arv algses pakis
1	1,9,10	20	1
2	2,9,10	21	2
3	3,9,10	22	3
4	4,9,10	23	4
5	5,9,10	24	5
6	6,9,10	25	6
7	7,9,10	26	7
8	8,9,10	27	8
9	9,9,10	0	9
10	9,10,10	0	10
11	9,10,11	30	11
12	9,10,12	31	12
`x2`	9,10,`x2`	38	1
`/2`	9,10,`/2`	9,5	1

Seega on järgmise kaardi võtmisel saadud käe oodatav punktisaak

$$\frac{1 \cdot 20 + 2 \cdot 21 + 3 \cdot 22 + 4 \cdot 23 + 5 \cdot 24 + 6 \cdot 25 + 7 \cdot 26 + 8 \cdot 27 + 11 \cdot 30 + 12 \cdot 31 + 1 \cdot 38 + 1 \cdot 9{,}5}{1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 11 + 12 + 1 + 1} \approx 20{,}47$$

punkti, mis on rohkem kui 19. Järelikult sellises olukorras on kasulik järgmine kaart võtta. Sellise lahendusega kulub mängu võitmiseks keskmiselt 11,5 vooru.

Kaardi võtmise statistiline keskväärtus koos kaartide lugemisega

Kuna kõik mängitud kaardid on lahendusele näha ja kaardipakk mängitakse lõpuni, siis on 79 kaardi järel võimalik täpselt öelda, mis on pakki jäänud viimane kaart ja selle järgi tuleks teha otsus, kas võtta või jätta. Kaartide lugemist saab kasutada eelmise lahenduse edasiarendusena, võttes arvesse kaardipakki järele jäänud kaardid, saame täpsemalt hinnata, kas on kasulik võtta või jätta.

Vaatame uuesti olukorda, kus käes on kaardid 9 ja 10, kuid pakki on jäänud palju üheksaid ja kümneid ning teisi kaarte on vähe. Vaatame eelmise näitega analoogilist tabelit:

Järgmine kaart	Käsi	Käe skoor	Kaartide arv pakis
1	1,9,10	20	1
2	2,9,10	21	1
3	3,9,10	22	1
4	4,9,10	23	2
5	5,9,10	24	2
6	6,9,10	25	3
7	7,9,10	26	3
8	8,9,10	27	4
9	9,9,10	0	8
10	9,10,10	0	9
11	9,10,11	30	5
12	9,10,12	31	6
`x2`	9,10,`x2`	38	1
`/2`	9,10,`/2`	9,5	1

Nüüd on järgmise kaardi võtmisel saadud käe oodatav punktisaak

$$\frac{1 \cdot 20 + 1 \cdot 21 + 1 \cdot 22 + 2 \cdot 23 + 2 \cdot 24 + 3 \cdot 25 + 3 \cdot 26 + 4 \cdot 27 + 5 \cdot 30 + 6 \cdot 31 + 1 \cdot 38 + 1 \cdot 9{,}5}{1 + 1 + 1 + 2 + 2 + 3 + 3 + 4 + 5 + 6 + 1 + 1} \approx 17{,}05,$$

mis on vähem kui hetkel käes olev punktisaak 19. Seega mõistlikum oleks selles olukorras käsi jätta.

Praktilised näpunäited

- Kui eelmiste voorude punktisumma pluss hetke käe skoor on vähemalt 200, siis tuleks jätta. Kaardimängus *Flip 7* üldiselt see päris nii lihtne ei ole.
- Keskväärtust arvutavaid lahendusi saab täiendada ka kahe või rohkema kaardi võtmise kasulikkuse tõenäosust hindama. Nii võiks lahendus muutuda täpsemaks, sest erikaardid `x2` ja `/2` muudavad ka järgmiste kaartide punktiskoori.
- Keskväärtust arvutavaid lahendusi saab täiendada, lisades mängu võitmise eest mingi sisemise boonuse, näiteks 40 punkti. Kui võidust on puudu väike arv punkte, siis võib olla kasulik viimastes voorudes riskida ning 200 punkti saavutamise boonus võiks sellele kaasa aidata.

7. Lahtipakkimine (`pakk`)

10 sek 100 punkti

Idee: Tähvend Uustalu, teostus ja lahenduse selgitus: P. Randla

Selles ülesandes on laias laastus kahte sorti sisendeid: sellised, kus sisend on võimalik võrdlemisi väikese programmi abil genereerida (ja ülesandeks jääb ainult selle programmi parameetrite leidmine ja siis "koodi golfimine"), ja sellised, kus tuleb sisend mingil (kompaktsel) kujul programmi teksti sisse kodeerida (sellisel juhul on põhiülesanne leida võimalikult efektiivne kodeerimine).

Selles ülesandes on testimiskeskkonnast eemaldatud kõik tavapärased pakkimisprogrammid ja -teegid. Ühest küljest aitab see vältida standardalgoritmidele ebavõrdse eelise andmist: kuna standardalgoritmi puhul ei pea algoritmi ennast programmi sisse kirjutama, siis tuleks lahendus selle võrra lühem. See aga tähendaks, et igasugune innovatsioon peaks ületama üsna suure künnise, et olla standardalgoritmist parem isegi siis, kui algoritm ise lahenduse hulka arvestada. Teisest küljest sunnib see osalejaid (loodetavasti) natukene rohkem pakkimise peale mõtlema.

Teksti kodeerimisest üldisemalt

Teksti (või ka muude andmete) pakkimisel on üldjuhul kolm etappi:

1. Sisendi eeltöötlemine, et selle "pakkivust" suurendada. Siia võib kuuluda näiteks korduvate sümbolite tuvastamine (nn "*run-length encoding*" ehk RLE, kus kodeeritakse iga sümboliga koos ka see, kui mitu korda järjest see sümbol esineb), pikemate korduvate tekstilõikude tuvastamine (Lempeli–Zivi algoritm), Burrowsi–Wheeleri teisendus, või PNG pildiformaadi puhul reafiltrite valimine.
2. Saadud sümbolite jada modelleerimine, st selle lähendamine mingi tõenäosusjaotusega. Lihtsamate pakkimisalgoritmide puhul loetakse lihtsalt kokku, kui mitu korda iga sümbol esineb. Kavalamad algoritmid võivad aga kasutada palju keerulisemaid mudeleid, kus iga sümboli tõenäosusjaotus sõltub sellele eelnenud sümbolitest (ehk nn *kontekstist*). Modelleerimine võib olla ka *adaptiivne*: pakkimise käigus uuendatakse mudelit jooksvalt vastavalt nähtud sümbolite jadale.
3. Sümbolite jada bittideks kodeerimine, kasutades leitud tõenäosusjaotust. Seda etappi kutsutakse ka entroopia-kodeerimiseks. Klassikaline ja üsna lihtne viis seda teha on Huffmani kood, mis seab igale sümbolile vastavusse mingi bitijada viisil, mis minimeerib kokku kuluvate bittide arvu (eeldusel, et sümbolid tõepoolest esinevad kasutatava jaotusega). On tõestatud, et Huffmani kood on suisa optimaalne(!), kuid praktikas seda moodsates algoritmides siiski ei kasutata: kood on optimaalne vaid juhul, kui iga sümbol esitatakse fikseeritud bitijadana. Selgub, et tihti on parem mõned sümbolid esitada "murdarvulise koguse" bittidega. Kaks levinud meetodit selle saavutamiseks on aritmeetiline kodeerimine ja asümmeetrilised numbrisüsteemid ("ANS").

Mainime, et reaalsetes pakkimisalgoritmides on need etapid omavahel tihedalt seotud; siiski võib neid filosofeerimise mõttes eraldi arutada.

Mõned elulised näited:

- ZIP-vormingus arhiivid kasutavad "DEFLATE"-nimelist algoritmi, mis kombineerib LZ-algoritmi ja Huffmani kodeeringu.
- Pakkimisalgoritm Bzip2 kasutab RLE-d, Burrowsi–Wheeleri teisendust ja lõpuks Huffmani kodeerimist.
- XZ- ja 7z-vormingud kasutavad LZMA algoritmi, mis kasutab LZ-algoritmi, keerulisemat adaptiivset kontekstipõhist tõenäosuste modelleerimist (siin kasutatakse Markovi ahelaid, millest ka algoritmi lühendis "MA") ja aritmeetilist kodeerimist.

- Uuem algoritm Zstandard kasutab LZ-algoritmi ja nii ANS- kui Huffmani kodeerimist.
- Väga tõhus, kuid aeglane algoritmide pere "PAQ8" kasutab sümbolite modelleerimiseks mitmeid erinevaid algoritme ja ka närvivõrke (ja seda juba aastal 2005!). Entroopiakodeerimiseks kasutatakse aritmeetilist kodeerimist.
- Vähemtuntud Bzip3 vorming (mis ei oma mingit seost Bzip2 vorminguga) kasutab LZ-algoritmi, Burrowsi–Wheeleri teisendust ja aritmeetilist kodeerimist.

Antud ülesandes on põhirõhk ülaltoodud etappidest teisel ja kolmandal: sisendites `input_006.txt` ja `input_007.txt` (ja võistluse jooksul kahjuks ka sisendis `input_010.txt`) tuleb üsna suur kogus andmeid kodeerida. Teksti pakkimise sisendites (`input_001.txt` ja `input_002.txt`) sobib muidugi ka tekstile spetsiifiline eeltöötlus.

`input_001.txt`

See sisend on A. H. Tammsaare romaani "Tõde ja õigus" esimene osa. Selle pakkimiseks sobib kasutada mingit kombinatsiooni eelmainitud algoritmidest.

Parima võistluse jooksul esitatud lahenduse pikkus oli 260 487 baiti, selle esitas Mattias Joosep Tammaru. Lahendus paistab olevat aritmeetiline kodeerimine koos võrdlemisi keerulise adaptiivse kontekstimodelleerimisega. Teine koht (Kristjan Lepp 304 808 baidiga) implementeeris Bzip2 dekodeerimisalgoritmi.

`input_002.txt`

See sisend on Linuxi tuuma (v7.2) lähtekoodi fail `net/wireless/nl80211.c`.

Ka siin töötab üldine tekstipakkimisalgoritm. Võib aga täheldada, et testimiskeskkonnas leidub fail `/usr/include/linux/nl80211.h`, mis on vastava `.c`-faili *header*-fail. See ei sisalda küll kogu `.c`-faili infot, kuid kui seda faili sisendi kodeerimisel sõnastikuna kasutada, siis võib olla võimalik mõningase korduva informatsiooni kodeerimist ära hoida. Võistluse jooksul keegi seda ei proovinud.

Parima võistluse jooksul esitatud lahenduse pikkus oli 73 275 baiti, selle esitas Mattias Joosep Tammaru.

`input_003.txt`

Nagu ülesande tekstis öeldud, on see sisend reaalarv, mis esitub teatud konstantide lineaarkombinatsioonina. Täpsemini on tegu arvude $\sqrt{2}$, $\ln 2$ ja e (ümardatud) lineaarkombinatsiooniga, kus kõik kordajad on täisarvud. (Sisendi algusesse oli peidetud vihjena arvude $\sqrt{2}$, π ja e kümnendesituste paar esimest numbrit; neist π oli küll "punane heeringas".)

Tähistame sisendis antud reaalarvu muutujaga t ja otsitavad kordajad muutujatega x, y ja z (ehk siis $x\sqrt{2} + y \ln 2 + ze = t$).

Selle lineaarkombinatsiooni leidmiseks saab kasutada Lenstra–Lenstra–Lovászi võre baasitaanduse algoritmi, ehk lühidalt LLL-algoritmi. LLL-algoritmist võib (selle ülesande kontekstis) mõelda kui mustast kastist, mis võtab sisendiks täisarvude maatriksi ja tagastab kõige lühema vektori (tavalise eukleidilise kauguse mõttes), mis on võimalik saada selle maatriksi ridade lineaarkombinatsioonina.

Vaatleme maatriksit

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & \lfloor w\sqrt{2} \rfloor \\ 0 & 1 & 0 & 0 & \lfloor w \ln 2 \rfloor \\ 0 & 0 & 1 & 0 & \lfloor we \rfloor \\ 0 & 0 & 0 & 1 & \lfloor wt \rfloor \end{bmatrix},$$

kus w on mingi suur konstant (näiteks $w = 2^{256}$).

Kui võtame selle maatriksi ridadest lineaarkombinatsiooni, kus iga rida on vastavalt kordajatega $x, y, z, -1$, saame vektori $(x, y, z, -1, E)$, kus $E = x\lfloor w\sqrt{2} \rfloor + y\lfloor w \ln 2 \rfloor + z\lfloor we \rfloor - \lfloor wt \rfloor$. Märkame, et kui E avaldises ümardamised ära jätta, siis kehtiks $E = 0$. Seega võiks loota, et E on "üsna väike". Seega vektor $(x, y, z, -1, E)$ on maatriksi M ridade lineaarkombinatsioon, ja üsna lühike. Valides w piisavalt suure, võib ka loota, et see on lausa kõige lühem võimalik vektor, ehk et LLL-algoritm tagastab selle vektori.

Selle lahenduse saab realiseerida näiteks vabavaralise arvutimatemaatikasüsteemi SageMath abil:

```
tgt = RealNumber(open("input_003.txt").read().strip())
w = 2^256
M = matrix([
  [1, 0, 0, 0, floor(w * sqrt(2))],
  [0, 1, 0, 0, floor(w * ln(2))],
  [0, 0, 1, 0, floor(w * e)],
  [0, 0, 0, 1, floor(w * tgt)],
])

print(M.LLL()[0])
```

See kood väljastab vektori (1522251764430814662, −4704692201478199620, 407704246355058087, 1, 3286591480521701456), ja võib kontrollida, et nende kordajatega saab tõepoolest sisendi rekonstrueerida. (Pane tähele, et ülaltoodud arutelus on vektori neljas komponent −1, mitte 1, seega tuleb siinsed kordajad võtta vastasmärgiga.)

Mainime, et ei ole tarvis kohe ette ära arvata, milliseid konstante ülesandes kasutati. Kogu ülaltoodud arutelu võib läbi teha ka rohkem kui kolme reaalarvulise konstandi jaoks; siis tuleb LLL-ist saadavasse vektorisse lihtsalt hulk nulle (mis vastavad nendele konstantidele, mida antud kombinatsioonis ei kasutatud).

Esitatavas programmis endas on kõige mõistlikum kasutada Pythoni standardteegi `decimal` moodulit:

```
# LANGUAGE: PyPy
from decimal import*
c=getcontext()
c.prec=3000
print(c.ln(2)*0x414a6ca176a89944-c.sqrt(8)*0xa900ff50ece1ee3
  -c.exp(1)*0x5a874da19695da7)
```

Siin viimast reapoolitust ei tasu muidugi programmi sisse jätta; see on toodud lihtsalt selleks, et tekst leheküljele ära mahuks. Lisaks pane tähele, et $\sqrt{2}$ kordaja juhtus olema paarisarv, seega saab arvu $\sqrt{2}$ asendada arvuga $\sqrt{8}$ ja vastava kordaja kahega jagada.

Selle lahenduse pikkus on 136 baiti; võistluse jooksul parim esitatud lahendus oli 137 baiti. Siinne lahendus põhineb Karl Markus Konstabeli ja Mattias Joosep Tammaru lahenduste ideedel.

input_004.txt

Selles sisendis on (nagu ülesande tekstis mainitud) tegu lineaarrekurrentsiga. Kui on teada kor-

dajad $a_1, \ldots, a_k$, siis peab programmis kodeerima ainult need kordajad ja jada esimesed k arvu, kõik järgnevad saab rekurrentsi abil välja arvutada. Kuna selle sisendi jaoks on moodul ette antud (ja algarv), siis saab kordajate leidmiseks rakendada Berlekampi–Massey algoritmi:

```
from sage.matrix.berlekamp_massey import berlekamp_massey
nums = open("input_004.txt").read().split()
F = GF(10^9+7)
v = [F(x) for x in nums[:50]]
bm = berlekamp_massey(v) # tagastab kordajad polünoomkujul
taps = companion_matrix(bm).T[-1] # annab kordajad otse kasutataval kujul
print(taps)
```

Edasi jääb küsimus, kuidas need kordajad võimalikult kompaktselt programmi kodeerida. Parim žüriile teadaolev lahendus on järgnev:

```
# LANGUAGE: PyPy
p=10**9+7
n=int("4dogl3wabzn0faub0eyjcoj61h2qltpg2dsxsyqpevua6n02xwib9ijh2h",36)
L=[]
exec("L+=sum(map(int.__mul__,L[-5:],L),n)%p,;n//=p;"*32773)
print(*L[5:],end=" ")
```

Siin kasutatakse tavalise `for`- või `while`-tsükli asemel ära `exec` funktsiooni ja sõnede kordamist korrutamismärgi abil. See on loomulikult tohutult ebaefektiivne (Pythoni/PyPy kompilaator peab selle võrra 32 773 korda rohkem tööd tegema), kuid on mõned baidid lühem kui vastav `for`-tsükkel.

Nii kordajad a_i kui ka jada esimesed 5 liiget on kodeeritud ühte täisarvu '`n`'. "Tsükli" esimese 10 iteratsiooni jooksul loetakse arve arvust `n` (mingil määral küll segatud massiivi `L` olemasolevate elementide poolt) ja lisatakse neid massiivi `L`. Pärast kümnendat iteratsiooni on `n` = 0, massiivi `L` esimesed 5 liiget on kordajad ja edasi implementeerib tsükkel lihtsalt soovitud rekurrentsi.

Lisaks mainime, et loendile saab elemente lõppu lisada käsuga `l+=elem,` (mida Python loeb kui `l+=(elem,)`, st üheliikmelise enniku lisamine), mis on samaväärne käsuga `l.append(elem)`, kuid on märksa lühem.

Ülaltoodud lahenduse pikkus on 164 baiti; selle idee pärineb Karl Markus Konstabeli lahendusest. Parim võistluse jooksul esitatud lahendus oli 171 baiti pikk.

`input_005.txt`

Sarnane eelmise sisendiga, kuid nüüd on ka moodul tundmatu. Mooduli leidmiseks saab rakendada Joan F. Boyari artiklis "*Inferring Sequences Produced by Pseudo-Random Number Generators*" kirjeldatud algoritmi.

Tegelikult läheb artiklis kirjeldatud algoritmist vaja vaid esimest sammu, mis leiab mooduli mingi kordse: seda sammu on võimalik korrata sisendjada mitme erineva osa peal, et saada mitu erinevat mooduli kordset; nendest suurima ühisteguri võtmisel saamegi algse mooduli.

Edasi selgub, et moodul ei ole algarv ja seega ei saa Berlekampi–Massey algoritmi otse rakendada. Kuna aga moodul on võrdlemisi väike (kõigest 64-bitine), siis saab selle hõlpsalt ära tegurdada ja iga teguri järgi eraldi Berlekampi–Massey algoritmi rakendada. Seejärel tuleb saadud vastused kombineerida hiina jäägiteoreemiga.

Need sammud on implementeeritud failis `crack_lfsr_1.sage`.

Leitud kordajate programmi kodeerimiseks läheb siin aga natuke teistsuguseid trikke vaja:

```
# LANGUAGE: PyPy
#coding:l1
p,*L=memoryview(bytes('\x9b\x33\x21\x6e\xd9\xf9\x7b\xf0\x18\x80\x15\x08\x3a
   \x69\x7e\x76\xdd\x9d\x35\xb7\xd8\x54\xf0\x97\x7a\x90\x89\x10\x01\x01\x83
   \xb3\x32\xd7\x89\x78\x3c\xed\x58\x26\xb6\xc3\x31\x38\xce\xbd\x9c\x6f\x98
   \x68\xaf\xef\x82\x01\xdb\x42\xa6\xfe\x3b\x8e\xbd\xa0\x61\x20\xf1\xf0\xd4
   \x8e\xce\x81\xdc\x02\xdd\xa3\x6a\xcd\x91\x30\x88\xa8\x55\x78\x84\x52\x72
   \x22\x86\x37',"l1")).cast("Q")
exec("L+=sum(map(int.__mul__,L[-5:],L))%p,;"*16379)
print(*L[5:],end=" ")
```

Siin on `bytes`-i argument toodud Pythoni `\x`-käsklustega, kuid tegelikus programmis on need baidid lähtekoodis "toorelt". Kuna sellised baidijadad ei ole UTF-8 kodeeringus lubatud, siis tavaliselt annaks Python selle peale veateate. Programmi alguses on aga kommentaar `#coding:l1`, mistõttu loeb Python seda faili "Latin-1" kodeeringus. Selles kodeeringus on lubatud kõik baidid (peale `\x00`-baidi), ja iga bait vastab sama indeksiga Unicode'i tähemärgile. `memoryview().cast()` abil saab need baidid üsna hõlpsalt teisendada täisarvude massiiviks, millest loetakse siis välja moodul `p` ja kordajad / esialgsed jada elemendid `L`. Edasi toimib lahendus täpselt samamoodi kui eelmises sisendis.

Lahenduse pikkuseks on 214 baiti; sellise lahenduse esitas võistluse ajal Mattias Joosep Tammaru.

input_006.txt

Selles sisendis on tegu teatud suunatud graafi naabrusmaatriksiga. Sisendit uurides võib märgata, et antud graaf on transitiivne: kui leidub serv tippude u ja v vahel, ja leidub serv tippude v ja w vahel, siis leidub ka serv tippude u ja w vahel. See tähendab, et ei ole vaja programmis kodeerida kõiki graafi servi, vaid piisab neist, mis ei ole saadavad teiste servade ja transitiivsuse kaudu. Sellist teisendust nimetatakse graafi transitiivseks reduktsiooniks.

Seega taandub ülesanne ligikaudu 1600 tipupaari kodeerimisele, millest on võimalik transitiivse sulundi arvutamisega taastada algne graaf.

Kuna ei ole oluline, mis järjekorras graafi servi hoiame, siis võime servade loendi sorteerida (kõigepealt algustipu järgi, viigi korral ka lõpptipu järgi). Siis saame servi kodeerida n-ö delta-kodeeringuga: teisendame iga serva täisarvuks (näiteks $x \cdot 1300 + y$) ja kodeerime järjestikuste arvude erinevused.

Selle kodeerimise realiseerimiseks on erinevaid võimalusi. Žürii lahendus (failis `sol_graph.py`, mille moodustas `crack_graph.py`) kasutab servade jaoks ANS-kodeeringut. Selle pikkuseks 2651 baiti. Selle sisendi jaoks ei ole see aga optimaalne: paremini sobib mõni lihtsamini dekodeeritav delta-kood.

Parim esitatud lahendus oli 2550 baiti; selle esitas Mattias Joosep Tammaru. (Selles lahenduses kodeeriti (δ mod 512) otse 9 bitina, $\lfloor \delta/512 \rfloor$ aga unaarse kodeeringuga.)

input_007.txt

Tegu on PPM-vormingus pildiga (vt Joonis 1).

Siin saab rakendada üldiseid pakkimisalgoritme, kuid kasulik oleks pildi kahemõõtmelisust ära kasutada.

Joonis 1: Seitsmes sisend.

Üks võimalik lahendus kasutab adaptiivset ANS-kodeerimist Pythoni suurtel täisarvudel:

```
# LANGUAGE: PyPy
#coding:l1
# Dekodeerime oma suure täisarvu:
N = 0
for ch in r"""<DATA>""":
    # Pythonile ei meeldi, kui programmi lähtetekstis 0x00-baidid on. Lisaks
    # teisendab Python toored 0x0D-baidid ('\r') 0x0A-baitideks, seega ei saa ka
    # neid kodeerimisel otse kasutada.
    # Kasutame aga kõiki muid baite.
    N = N * 254 + ord(ch) - 1 - (ch>"\r")

output = bytearray(b"P6\n512 512\n255\n")
# mudel[k][0]: Kui mitu heledat pikslit kontekstis k nähtud on?
# mudel[k][1]: Kui mitu korda konteksti k üldse nähtud on?
# Esialgu eeldame, et igas kontekstis on üks hele ja üks tume piksel.
mudel = [(1,2)] * 416
# eelmiste pikslite värvid:
e = [0] * 1025
# iga piksli jaoks:
for n in range(8**6):
    y, x = divmod(n, 512)
    # kaugus pildi servast:
    kaugus = min(x, y, 511 - x, 511 - y)
    if 47 < kaugus < 80:
        # selles vahemikus on kõik pikslid valged.
        v = 0
    else:
        kontekst = 32 * [
            # Leiame, millises pildi osas hetkel oleme.
            # välimise serva jaoks: leiame, kas me oleme horisontaalses või
            # vertikaalses osas, ja sinises või mustas osas
            (min(x, 511 - x) == kaugus) + 2 * (kaugus > 15),
            # mujal: kas oleme "I" triibu juures või sellest kummalgi pool,
            # ja kas oleme "E" ülemise serva kohal või "E" alumise serva all
            ((x > 239) + (x > 271)) * 3 + (y > 176) + (y > 335) + 4
          ][kaugus > 47
        # Leiame, mis värvi olid naaberpikslid:
        #   üleval,   vasakul,    üleval+vasak, üleval+parem, 2 rida üleval
        ] + e[-512] + 2 * e[-1] + 4 * e[-513] + 8 * e[-511] + 16 * e[-1024]

        # loeme ühe biti ANS koodijadast:
        p,q = mudel[kontekst]
        v = N%q >= p
        N = N//q * [p,q-p][v] + N%q - p*v
```

```
            # uuendame oma mudelit:
            mudel[kontekst] = p+1-v, q+1
        # Salvestame leitud piksli ja lisame selle väljundisse:
        e += v,
        # 0 = valge, 1 = must v.a. sinises ümbritsevas triibus, kus 1 = sinine
        output += [b"\xff" * 3, b"\0" * 3, b" @\x80"][v + v * (15 < kaugus < 48)]
open(1, "wb").write(output)
```

Siin tuleb `<DATA>` asendada 7697-baidise sõnega, mis kodeerib kogu pildi informatsiooni, ja kood minimeerida (eemaldada üleliigsed tühikud ja kommentaarid). Seda teeb fail `sol_pilt_gen.py`. Kokku tuleb lahenduse pikkuseks 8162 baiti.

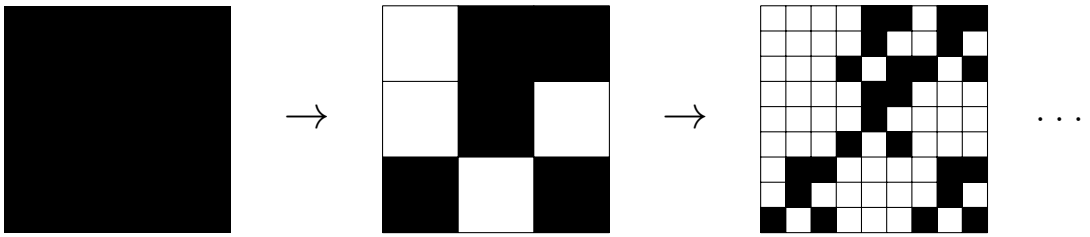
See lahendus on võrdlemisi aeglane: iga biti dekodeerimiseks tuleb teha jagamistehe täisarvuga `N`, mis ise sisaldab kõigi bittide dekodeerimiseks vajalikku infot. Seega on tegu $O(n^2)$ keerukusega. Päriselus kasutatavad ANS-(de)kodeerijad kasutavad sümbolite tõenäosuste nimetajatena alati kahe astmeid ja loevad arvu `N` bitikaupa. Sellisel juhul ei ole vaja suurte arvudega tehteid teha, kuid sümbolite tõenäosuste ümardamise tõttu langeb kodeerimise efektiivsus. Siiski, kuna nimetajaks võib võtta näiteks 2^{16} või 2^{20}, on tavaliselt efektiivsuse langus kaduvväike. Mainime, et sama olukord tekib ka aritmeetilistes kodeerijates.

Parim võistluse jooksul esitatud lahendus oli 8468 baiti; selle esitas Kristjan Lepp. Siinne lahendus kasutab Kristjani leitud kontekste, kuid efektiivsema ANS-kodeeringuga.

input_009.txt

`input_008.txt` ja `input_009.txt` on genereeritud peaaegu sama loogikaga; alustame siin üheksandast. Mõlemad sisendid on fraktalid (täpsemalt lõplik lähend fraktalile).

`input_009.txt` on moodustatud sarnaselt Sierpiński kolmnurgaga või täpsemalt Sierpiński vaibaga: alustatud on ühest ruudust, see on jagatud 3×3 alamruuduks, nendest alamruutudest neli tükki on ära kustutatud ja ülejäänud viies sama protsessi korratud.



Joonis 2: `input_009.txt` konstruktsioon.

Kui see struktuur ära tabada, siis jääb üle vaid sama protsess koodis implementeerida. Sisendi suurus on 729×729, mis vastab protsessi 6 korda rakendamisele.

Parim esitatud lahendus on järgnev:

```
# LANGUAGE: PyPy
for i in range(729):
    s,d="#.";exec("s=(d+s*2,d+s+d,s+d+s)[i%3];d*=3;i//=3;"*6);print(s)
```

(Esitatud programmis tuleb jällegi reavahetus ära kustutada.)

Selle esitas Kristjan Lepp ja selle pikkuseks on 88 baiti.

input_008.txt

See sisend on moodustatud täpselt sama moodi, nagu `input_009.txt`, kuid 'AND'-tehte asemel on kasutatud 'XOR'-tehet (s.t. rekursiivses protsessis on osade ruutude värv ära vahetatud, selle asemel et need ruudud kustutada).

```
# LANGUAGE: PyPy
for i in range(729):
    a,b=".#"
    exec("u=a+b;a,b=((u+b,b+a+a),(u+a,b+u),(b+u,u+a))[i%3];i//=3;"*6)
    print(a)
```

Selle esitas Kristjan Lepp ja selle pikkuseks on 103 baiti.

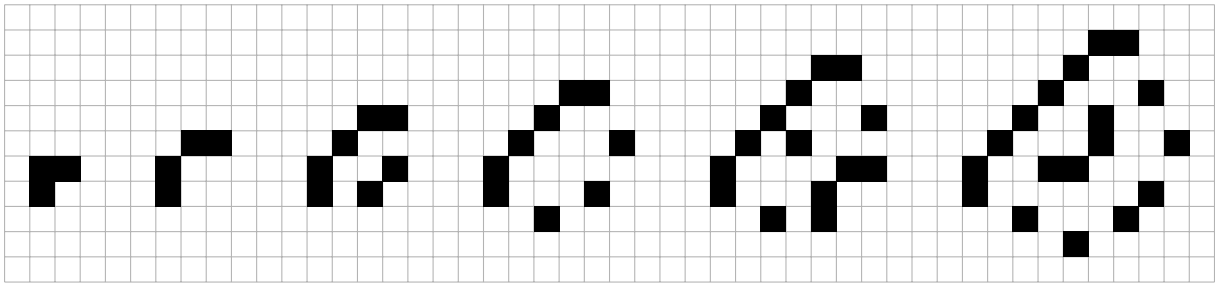
Alternatiivne lahendus (samuti 103 baiti):

```
# LANGUAGE: PyPy
R=range(729)
for i in R:
 print("".join(".#"[sum(342>>i//3**k%3*3+j//3**k%3for k in R[:6])%2]for j in R))
```

Selle autor on Mattias Joosep Tammaru.

input_010.txt

See sisend kujutab rakkautomaadi "*Seeds*" (vaata ka siit) ühe lihtsa mustri arengut. See automaat on võrdlemisi lihtsa reeglistikuga: rakk "sünnib", kui ta ei ole elus ja tal on täpselt kaks elus naabrit. Igal muul juhul rakk "sureb" (või jääb surnuks).



Joonis 3: *Seeds*'i areng.

Antud sisend on sama mustri areng pärast 499 sammu. Nurka on lisatud ka paar "ostsillaatorit", mis võiksid aidata tuvastada, millise automaadiga on tegu.

Žüriil on lahendus pikkusega 284 baiti:

```
# LANGUAGE: PyPy
R = range(1024)
# 0x0101010101....010101, 1024 korda
K = 2**8192 // 255
# iga rida ühe täisarvuna. iga 8 bitti tähistab ühte rakku: 0 = "surnud",
# 1 = "elus", 2..255 aritmeetika jaoks reserveeritud.
themap = [0]*1024
# esialgne muster:
themap[511] = 257 << 4088
themap[512] = 1 << 4096
for _ in "x"*499:  # See for-tsükkel tuleb eelmainitud 'exec'-häkiks teisendada
    # Arvutame terve rea uue oleku korraga,
    # ja terve mustri uue oleku ühe listikomprehensiooniga
```

```
    themap = [
        # K^ on tegelikult viimane samm, aga on siin ühe baidi kokku hoidmiseks
        K ^
        # loeme kokku iga raku ülemised ja alumised naabrid:
        (((tmp := sum(themap[y-1:y+2]))
            # loeme kokku vasakud ja paremad naabrid
            *257 + (tmp>>8)  # (tmp*257 = (tmp<<8) + tmp)
        # teame, et baidid on kõik 0..9, otsime kahtesid
        ^ 2*K)
        # nüüd on baidid 0..11, otsime nulle
        + 15*K
        # baidid 15..26, otsime 15
        >> 4 & K
        # 1 kui oli mittenull, 0 kui oli null, st oli kaks naabrit
        | themap[y]
        # 0 ainult siis, kui oli 2 naabrit ja ei olnud elus, muidu 1
        ) # ^K (tehtud algusepoole)
        # vastavalt 1 ja 0
    for y in R]

# lisame nurgas oleva vihjemustri:
# siin on tegelikult vaja ainult 140 iteratsiooni, aga rohkem ei tee kahju.
for i in R:
    themap[2+i//35] |= (0xa080000e225400e2a007112a100001410 >> i & 1) << i%35*8

for row in themap:
    # teisendame iga rea otse ASCII-ks:
    # 0 -> 46 = '.',
    # 1 -> 46-11 = 35 = '#'
    print((K*46 - row*11).to_bytes(1024).decode())
```

284-baidise programmi saamiseks tuleb see kood lasta läbi `uglify.py` skriptist.

Parim võistluse ajal esitatud lahendus oli 22 980 baiti.