Jaguvus

Olgu $a, b \in \mathbb{Z}$. Kui leidub selline $k \in \mathbb{Z}$, et $ak = b$, siis öeldakse, et **arv** a **jagab arvu** b (ehk **arv** b **jagub arvuga** a), ja kirjutatakse $a|b$.

Jaguvuse omadusi:

1. Kui $a|b$ ja $b \neq 0$, siis $|a| \leqslant |b|$. Järelikult, kui $a|b$ ja $b|a$, siis $|a| = |b|$.
2. Kui $a|b$ ja $b|c$, siis $a|c$.
3. Kui $a|b$ ja $a|c$, siis $a|(b + c)$.
4. Mistahes $k \in \mathbb{Z}$ korral: kui $a|b$, siis $a|(kb)$.
5. Mistahes $k_1, k_2, \ldots, k_n \in \mathbb{Z}$ korral: kui $a|b_1$, $a|b_2$, $\ldots$, $a|b_n$, siis $a|(k_1b_1 + k_2b_2 + \cdots + k_nb_n)$. See omadus järeldub kahest eelmisest.
6. Kui $a|b$ ja $c|d$, siis $ac|bd$.

Algarvud

Positiivset täisarvu n, millel on täpselt kaks erinevat tegurit (1 ja n ise), nimetatakse **algarvuks**. Algarvud on 2, 3, 5, 7, 11, $\ldots$ ja neid on lõpmata palju, seejuures 2 on ainus paaris algarv. Ühest suuremat mittealgarvu nimetatakse kordarvuks. Arve 0 ja 1 ei loeta ei alg- ega kordarvudeks.

Mistahes $n \in \mathbb{Z}^+$ on üheselt esitatav algarvude astmete korrutisena kujul $n = 2^{m_2}3^{m_3}5^{m_5}7^{m_7}\ldots$ (meenutame, et aste astendajaga 0 on 1).

Kui n on kordarv, peab ta jaguma mingi algarvuga, mis ei ületa $\sqrt{n}$. Seega, veendumaks, et n on algarv, piisab kontrollida tema mittejaguvust arvudega, mille ruut ei ületa n. Sellise kontrolli ajaline keerukus on $O(\sqrt{n})$.

Kui algarvulisust on vaja kontrollida paljude arvude jaoks, võib olla efektiivsem arvutada välja kõik algarvud, mille ruut ei ületa kontrollitavate arvude ülempiiri N, ning seejärel kontrollida iga arvu n jaguvust mitte enam kõigi tema ruutjuurt mitte ületavate arvudega, vaid ainult selliste algarvudega. On tõestatud, et arvust n väiksemaid algarve on ligikaudu $\frac{n}{\ln n}$, mis teeb iga arvu algarvulisuse kontrolli keerukuseks $O(\frac{\sqrt{n}}{\ln n})$. See on eelnevast kiirem, kuigi alguses on algarvude leidmiseks vaja teha lisatööd.

Järjestikuste algarvude leidmiseks paigutame kõigepealt loendisse arvu 2. Seejärel kontrollime iga järgneva arvu jaguvust nende varem leitud algarvudega, mille ruut ei ületa vaadeldavat kandidaati. Kui arv ei jagu ühegagi neist, on ta algarv ja me lisame ta loendi lõppu.

Suurim ühistegur ja vähim ühiskordne

Mistahes $a, b \in \mathbb{Z}$ korral defineerime SÜT(a, b) kui suurima $d \in \mathbb{N}$, mille korral $d|a$ ja $d|b$ ning VÜK(a, b) kui vähima $m \in \mathbb{N}$, mille korral $a|m$ ja $b|m$.

SÜT ja VÜK leidmine on lihtne, kui arvud on esitatud algtegurite korrutisena.
Olgu $a = 2^{a_2}3^{a_3}5^{a_5}\ldots$ ja $b = 2^{b_2}3^{b_3}5^{b_5}\ldots$, siis
SÜT$(a, b) = 2^{\min(a_2,b_2)}3^{\min(a_3,b_3)}5^{\min(a_5,b_5)}\ldots$ ja
VÜK$(a, b) = 2^{\max(a_2,b_2)}3^{\max(a_3,b_3)}5^{\max(a_5,b_5)}\ldots$

SÜT ja VÜK omadusi:

1. Mistahes $a \in \mathbb{Z}$ korral: $\text{SÜT}(a, 0) = |a|$, $\text{VÜK}(a, 0) = 0$.
2. Mistahes $a, b \in \mathbb{Z}$ korral: $\text{SÜT}(a, b) \cdot \text{VÜK}(a, b) = |ab|$.
3. Mistahes $a, b \in \mathbb{Z}$, $m \in \mathbb{N}$ korral: $\text{SÜT}(ma, mb) = m\text{SÜT}(a, b)$.
4. Mistahes $a, b \in \mathbb{Z}$, $m \in \mathbb{N}$ korral: $\text{VÜK}(ma, mb) = m\text{VÜK}(a, b)$.
5. Mistahes $a, b \in \mathbb{Z}$ korral: $\text{SÜT}(a, b) = \text{SÜT}(a \pm b, b)$.
6. Mistahes $a, b \in \mathbb{Z}$ korral: $\text{SÜT}(a, b) = \text{SÜT}(a \bmod b, b)$.
7. Kui $a|bc$ ja $\text{SÜT}(a, b) = 1$, siis $a|c$.
8. Kui $a_i \in \mathbb{N}$ on paarikaupa ühistegurita ja $a_1 a_2 \dots a_n = b^k$, siis leiduvad $b_1, b_2, \dots, b_n \in \mathbb{N}$, mille korral $a_1 = b_1^k$, $a_2 = b_2^k$, $\dots$, $a_n = b_n^k$. (Näiteks: kui a_i on paarikaupa ühistegurita ja $a_1 a_2 \dots a_n$ on täisruut, siis on iga a_i täisruut.)
9. Mistahes $a \in \mathbb{Z}$ korral: $\text{SÜT}(a, a + 1) = 1$.
10. Mistahes $a, b \in \mathbb{Z}$ korral: leiduvad sellised $u, v \in \mathbb{Z}$, et $\text{SÜT}(a, b) = au + bv$.
11. Mistahes $a, b, n \in \mathbb{Z}$ korral on võrrandil $ax + by = n$ täisarvuline lahend parajasti siis, kui $\text{SÜT}(a, b)|n$. Siis on võrrandi kõik lahendid kujul $(x, y) = (x_0 + t \cdot \frac{b}{d}, y_0 - t \cdot \frac{a}{d})$, kus (x_0, y_0) on selle võrrandi mistahes erilahend, $d = \text{SÜT}(a, b)$ ja $t \in \mathbb{Z}$.

Omadusel 6 põhineb **Eukleidese algoritm** SÜT leidmiseks, mis on kiirem kui algteguriteks lahutuse kasutamine. Kui on antud arvud a ja b, kus $a \geqslant b > 0$, siis asendame nad vastavalt arvudega b ja $a \bmod b$. Jätkame seda protsessi, kuni üks arvudest muutub nulliks, mis kindlasti kunagi juhtub, sest minimaalne arv väheneb igal sammul. Seejuures arvude suurim ühistegur ei muutu. Lõpuks, kui teine arv võrdub nulliga, rakendame omadust 1.

Kahe arvu VÜK on kõige lihtsam leida omaduse 2 abil, leides eelnevalt arvude SÜT Eukleidese algoritmiga.

Jäägiga jagamine ja kongruentsid

Mistahes $a \in \mathbb{Z}$, $m \in \mathbb{Z}^+$ jaoks leiduvad üheselt määratud täisarvud q ja r, mille korral $a = qm + r$ ja $0 \leqslant r < m$, kusjuures $r = 0$ parajasti siis kui $m|a$. Arvu r nimetatakse **jäägiks** a jagamisel m-ga ja tähistatakse $a \bmod m$. Arvu q ehk jagatise $\frac{a}{m}$ täisosa tähistatakse $a \operatorname{div} m$.

Keeles Pascal on jäägi leidmise ja täisarvulise jagamise tähisteks samuti `mod` ja `div`, keeltes C, C++ ja Java aga `%` ja `/` ning Pythonis `%` ja `//`. See annab ka mugava võimaluse jaguvuse testimiseks programmis: tuleb vaid kontrollida, kas $a \bmod m = 0$.

Väärib märkimist, et paljud programmikeeled defineerivad negatiivsete operandide korral jäägi ja jagatise teisti kui matemaatikud: näiteks C (C99 standardi järgi) ja Java ümardavad jagatise alati 0 (mitte $-\infty$) suunas, mistõttu jääk on alati jagatavaga samamärgiline; C++ standard ja vanemad C standardid ei määra jäägi märki, kui jagatav või jagaja on negatiivne, kuid kompilaatorid valivad tavaliselt jagatava märgi; ka Pascali standardi järgi peab `div` alati ümardama 0 suunas, aga tehe $a \bmod b$ pole negatiivse b korral üldse lubatud (kuigi praktikas valdav enamus kompilaatoreid seda siiski lubab). Pythonis seevastu on jääk sama märgiga kui jagaja. Üldiselt oleks parem vältida nende tehete kasutamisel negatiivseid arve.

Näide: olgu antud positiivsete arvude a, b ja m puhul vaja leida $(a - b) \bmod m$, kusjuures pole teada, kumb arvudest a ja b on teisest suurem. Siis allpool esitatud jääkide omadustest järeldub,

et igas keeles on ohutu kasutada avaldist "$((a \bmod m) - (b \bmod m) + m) \bmod m$", kuna jäägi võtmine toimub siin ainult positiivsest arvust.

Et jääkide omadusi oleks mugavam kirjeldada, on matemaatikas võetud kasutusele **kongruentsi** mõiste. Kui täisarvud a ja b annavad arvuga $m > 0$ jagades sama jäägi, siis kirjutatakse seda lühidalt $a \equiv b \pmod{m}$ ja öeldakse, et a ja b on **kongruentsed mooduli m järgi**.

Kongruentside (jääkide) omadusi:

1. Kui $a \equiv b \pmod{m}$ ja $b \equiv c \pmod{m}$, siis $a \equiv c \pmod{m}$.
2. Kui $a \equiv b \pmod{m}$ ja $c \equiv d \pmod{m}$, siis $a \pm c \equiv b \pm d \pmod{m}$ ja $ac \equiv bd \pmod{m}$.
3. Kui $a \equiv b \pmod{m}$ ja n on positiivne täisarv, siis $a^n \equiv b^n \pmod{m}$.
4. Olgu $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$. Siis $f(x \bmod m) \bmod m = f(x) \bmod m$.
5. Väited $a \equiv b \pmod{m}$, $a - b \equiv 0 \pmod{m}$, $m|(a-b)$ on samaväärsed.
6. Mistahes $k \in \mathbb{Z}^+$ korral: $a \equiv b \pmod{m}$ parajasti siis, kui $ka \equiv kb \pmod{km}$.
7. Mistahes $k \in \mathbb{Z}$ korral: kui $a \equiv b \pmod{m}$, siis $ka \equiv kb \pmod{m}$.
8. Kui $ak \equiv bk \pmod{m}$, siis $a \equiv b \pmod{\frac{m}{\mathrm{SÜT}(m,k)}}$.
9. Kui $a \equiv b \pmod{m_1}$, $a \equiv b \pmod{m_2}$, $\ldots$, $a \equiv b \pmod{m_n}$, siis $a \equiv b \pmod{\mathrm{VÜK}(m_1,m_2,\ldots,m_n)}$.
10. Kui $a \equiv b \pmod{m}$ ja $d|m$, siis $a \equiv b \pmod{d}$.

Positsioonilised arvusüsteemid

Mistahes $b \in \mathbb{N}$, $b > 1$ korral kasutatakse b-süsteemis arvude märkimiseks numbreid $0, 1, \ldots, b-1$. Arvu $\overline{a_n \ldots a_1 a_0}$ väärtus on $a_n b^n + \cdots + a_1 b^1 + a_0 b^0$.

Tasub tähele panna, et b-süsteemis on arvu a viimaseks numbriks $a \bmod b$. Viimased k numbrit kujutavad aga b-süsteemis arvu $a \bmod b^k$. Seega saab arvu viimaste numbrite kohta käivad ülesanded taandada jääkidega arvutamise ülesanneteks.

Arvu $\overline{0{,}a_1 a_2 \ldots a_n}$ väärtus on $a_1 b^{-1} + a_2 b^{-2} + \cdots + a_n b^{-n}$.

Arvuti mälus hoitakse arve kahendsüsteemis. Märgita täisarvutüüpe (mida Javas küll polegi) hoitakse nii, nagu arvata võikski: 8-bitine muutuja bittidega $\overline{00000110}$ esitab väärtust $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6$. Märgiga täisarvutüübi puhul esitatakse mittenegatiivseid arve samamoodi, kuid negatiivsete arvudega on esmapilgul keerulisem: näiteks 8-bitist arvu -1 talletatakse mälus kujul $\overline{11111111}$, mis 8-bitise märgita täisarvuna tõlgendades oleks 255, arvu -2 esitatakse kujul $\overline{11111110}$, mis vastab märgita täisarvule 254, -3 on $\overline{11111101}$, mis vastab märgita täisarvule 253 jne. Asi saab selgeks, kui neid väärtusi vaadelda jääkidena 2^n järgi: nimelt $-1 \equiv 255 \pmod{2^8}$, $-2 \equiv 254 \pmod{2^8}$ jne.

Tasub tähele panna, et paljudes keeltes toimub liitmine, lahutamine ja korrutamine n-bitiste (nii märgiga kui märgita) täisarvudega *modulo* 2^n: ka ületäitumise korral annab tulemus arvuga 2^n jagades sama jäägi, mille annaks õige vastus. Täisarvude mäluesitust arvestades tuleb nii ka välja, et liitmine, lahutamine ja korrutamine (kuid mitte jagamine!) toimuvad märgiga täisarvude korral täpselt samamoodi kui märgita täisarvude korral.

Suurte arvudega arvutamine

Vahel on vaja kasutada suuremaid täisarve kui arvuti riistvara ja programmikeel vaikimisi võimaldavad. Siis võib suurte arvude esitamiseks luua oma andmestruktuuri ning kirjutada alamprogrammid nendega tehete sooritamiseks. Kui arvutis hoitakse arve tavaliselt kahendsüsteemis, siis suuri täisarve on sisestamise ja väljastamise lihtsustamiseks mugavam hoida kümnendsüsteemis.

Lihtsaim variant: naturaalarvu $a = \overline{a_n \dots a_1 a_0}$ hoiame massiivis `a[0..n]`. Paneme tähele, et arvu hoiame "tagurpidi", ühelised alguses — see teeb palju mugavamaks tehete sooritamise.

Naturaalarvu n teisendus 10-süsteemis massiiviesitusse:

```
for i := 0 to max:
  a[i] := n mod 10
  n := n div 10
```

Väljastamine:

```
for k := max downto 0:
  if a[k] > 0:
    break
--- väljastamist alustame esimesest mittenullist
for i := k downto 0:
  write(a[i])
```

Kahe suure arvu liitmine, $c := a + b$:

```
x := 0 --- ülekanne eelmisest järgust
for i := 0 to max:
  x := x + a[i] + b[i]
  c[i] := x mod 10
  x := x div 10
```

Suure arvu korrutamine tavalisega, $a := a * b$:

```
x := 0 --- ülekanne eelmisest järgust
for i := 0 to max:
  x := x + a[i] * b
  a[i] := x mod 10
  x := x div 10
```

Suure arvu jagamine tavalisega, $a := a$ div b:

```
x := 0 --- ülekanne eelmisest järgust
for i := max downto 0:
  x := x * 10 + a[i]
  a[i] := x div b --- alati tuleb a[i] < 10
  x := x mod b
--- nüüd x = a mod b
```

Suurte arvude omavahel jagamine on nii tülikas, et sellest tuleks olümpiaadil kindlasti hoiduda, kui vähegi võimalik.

Kui on vaja naturaalarvude asemel arvutada täisarvudega, võime neid hoida samamoodi, nõudes lisaks, et kõigi numbrite märgid oleksid samad. Kui arvutuste käigus see nõue rikutud saab, siis normaliseerime:

```
for k := n downto 0:
  if a[k] <> 0:
    break
--- arvu märgi määrab kõrgeima järgu märk
x := 0 --- ülekanne eelmisest järgust
for i := 0 to k do:
  a[i] := a[i] + x; x := 0
  while a[k] < 0 and a[i] > 0:
    a[i] := a[i] - 10; x := x + 1
  while a[k] > 0 and a[i] < 0:
    a[i] := a[i] + 10; x := x - 1
```

Kiire astendamine

Olgu meil vaja leida a^n, kus n on naturaalarv. Programmikeelte standardvahendid pakuvad tavaliselt ainult reaalarvude astendamist. Sel juhul tuleb aga ka tulemus reaalarvutüüpi ning tekib ümardamisvea oht.

Teine võimalus oleks korrutada arvu a iseendaga n korda, kuid saab läbi ka palju efektiivsemalt. Kui järjest korrutamisel tuleb teha $O(n)$ korrutamistehet, siis järgnev meetod saab hakkama $O(\log n)$ tehtega, seega töötab ülikiiresti ka juhul, kui astendaja on näiteks 10 000 000.

Vaatleme arvu n kahendesitust: $n = 2^k b_k + 2^{k-1} b_{k-1} + \cdots + 2b_1 + b_0$, kus iga b_i on 0 või 1. Siis a^n kujutab endast selliste astmete a^{2^i} korrutist, kus $b_i = 1$. Seega võib leida ainult need arvu a astmed, mille astendajaks on n-st mitte suuremad 2 astmed, ning vajalikud nende hulgast korrutada. Arvu a järjest ruutu tõstes saame $a^2, a^4, a^8, \ldots, a^{2^k}$, mille hulgas ongi kõik astmed, mida vaja.

Samuti sobib see hästi juhul, kui on vaja leida mitte aste ise, vaid jääk astme jagamisel mingi arvuga m. See kehtib tänu kongruentside omadusele 3. Näide: leiame arvu $514^{141} \bmod 100$, st selle arvu 2 viimast kümnendnumbrit.

$141 = 128 + 8 + 4 + 1 = 2^7 + 2^3 + 2^2 + 2^0$

$$
\begin{array}{llllll|llllll}
514^1 \bmod 100 & = & & = & 14 & & 514^{16} \bmod 100 & = & 56^2 \bmod 100 & = & 36 \\
514^2 \bmod 100 & = & 14^2 \bmod 100 & = & 96 & & 514^{32} \bmod 100 & = & 36^2 \bmod 100 & = & 96 \\
514^4 \bmod 100 & = & 96^2 \bmod 100 & = & 16 & & 514^{64} \bmod 100 & = & 96^2 \bmod 100 & = & 16 \\
514^8 \bmod 100 & = & 16^2 \bmod 100 & = & 56 & & 514^{128} \bmod 100 & = & 16^2 \bmod 100 & = & 56
\end{array}
$$

$514^{141} \bmod 100 = 514^{128} \cdot 514^8 \cdot 514^4 \cdot 514^1 \bmod 100 = 56 \cdot 56 \cdot 16 \cdot 14 \bmod 100 = 64$

Astendamine, $c := a^b \bmod m$ ($a \geqslant 0$, $b \geqslant 0$, $m > 0$):

```
c := 1; x := a
while b > 0:
  if b mod 2 = 1:
    c := (c * x) mod m
  x := (x * x) mod m
  b := b div 2
```

Natuke krüptoloogiast

Eelmises jaotises nägime, kuidas antud arvude s, a ja m abil leida arvu $x = s^a \bmod m$. Vaatleme nüüd pöördülesannet: antud s, m ja x järgi tuleb leida a. Et üldiselt on sobivaid a väärtusi lõpmata palju, võib lisada, et otsitakse vähimat positiivset sellist arvu a. Arvu a nimetatakse arvu x **diskreetseks logaritmiks** alusel s mooduli m järgi.

Nagu enne nägime, on astendamisülesanne mooduli järgi lihtsalt ja kiiresti lahenduv. Samas pöördoperatsiooni kohta ei ole teada efektiivsemat algoritmi kui a kõigi võimaluste läbivaatus. Kui m valida piisavalt suur (ning algarv), siis võtab diskreetse logaritmi leidmine arvutil miljoneid aastaid, samas kui astendamisülesanne samade arvudega lahendub endiselt kiiresti. Sellistel arvuteoreetilistel ülesannetel, mis on efektiivselt lahenduvad, nende pöördülesanne aga mitte, põhineb suur osa tänapäeva krüptoloogiast.

Oletame, et kaks inimest, Alice ja Bob, soovivad vahetada salajast informatsiooni, mis oleks kättesaamatu sidekanali pealtkuulajale. Tavaline moodus selleks on kasutada šifreerimist. Selleks on vaja võtit — täisarvu, mida teaksid mõlemad osapooled, kuid mitte keegi teine, ning mille abil saab sõnumi teksti šifreerida ja dešifreerida. Kui neil pole võtit omavahel varem kokku lepitud, saavad nad alguses (enne võtme kokkuleppimist) saata ainult kaitsmata sõnumeid ja siis on oht, et pealtkuulaja saab võtme teada, kui üks püüab seda teisele saata. **Diffie-Hellmani võtmevahetusalgoritm** põhineb diskreetse logaritmi ülesande keerukusel ning lubab Alice'il ja Bobil võtme nii kokku leppida, et nad võtit ennast läbi sidekanali ei saada.

Eelnevalt on fikseeritud hästi suur (näiteks 300-kohaline) algarv p ning mingi arv s, viimane võib olla ka väike arv, nt 2 või 3. Alice valib mingi arvu a (oma privaatvõtme) ja saadab Bobile arvu $x = s^a \bmod p$. Bob valib endale privaatvõtme b ja saadab Alice'ile arvu $y = s^b \bmod p$. Arvud x ja y arvutatakse muidugi kiire astendamise algoritmiga.

Paneme tähele, et

$$y^a \equiv (s^b)^a = (s^a)^b \equiv x^b \pmod{p}.$$

Seega $y^a \bmod p = x^b \bmod p$ ning see arv w saabki võtmeks. Paneme tähele, et nii Alice (kes teab arve a ja y) kui ka Bob (kes teab arve b ja x) saavad selle arvu kiire astendamisega välja arvutada. Samas oletatav pealtkuulaja teab küll arve x, y, p ja s, kuid arvude a ja b väljaarvutamiseks peaks ta lahendama diskreetse logaritmi ülesande, millega ta piisavalt suure p korral tõenäoliselt hakkama ei saa. Ei ole teada ka ühtegi meetodit arvu w arvutamiseks ainult x ja y abil, mis oleks kiirem kui a ja b leidmine.

Teiseks raskesti lahenduvaks ülesandeks, mis krüptoloogias kasutust leiab, on kordarvu algteguriteks lahutamine. Kui p ja q on kaks (suurt) algarvu, siis nende korrutamine on teostatav algoritmiga, mis on polünomiaalse keerukusega algarvude bittide arvu suhtes. Kui aga on antud arv pq ja ülesandeks leida selle algtegurid, siis kõik teadaolevad algoritmid vajavad selleks nende algarvude väärtusega võrreldavat sammude arvu. Sellel ülesandel põhineb laialdaselt kasutatav **Rivesti-Shamiri-Adlemani šifreerimisalgoritm**.

Kasutusel on kaks võtit: avalik ja salajane. Avaliku võtmega šifreeritud teksti saab dešifreerida vaid salajase võtme abil. Seejuures on avaliku võtme põhiosaks suur täisarv, mis on kahe algarvu korrutis. Šifreeringu turvalisus põhineb sellel, et hoolimata selle võtme avalikkusest (mis on vajalik, et igaüks saaks võtme omanikule salastatud sõnumit saata) ei suuda ükski kõrvaline isik leida neid kahte algarvu. Muide, šifreerimine ise kujutab siin endast astendamist: kui sõnumi tekst on täisarv m, siis šifreerides esitatakse see kujul $m^k \bmod pq$, kus k on samuti avaliku võtme juurde kuuluv arv.

Ülesandeid

1. Panna kongruentside eeltoodud omadused kirja, kasutades tähistusi x div y ja x mod y.
2. Tasand on värvitud malelaua moodi mustadeks ja valgeteks ruutudeks. Ühe ruudu laius ja kõrgus on täisarv a, ruutude servad on koordinaattelgedega paralleelsed ning koordinaatide alguspunkt on ühe musta ruudu vasakus alumises nurgas. Antud on tasandi punkti koordinaadid (täisarvud). Leida, kas see punkt asub mõne ruudu serval või tipus, ning kui ei, siis mis värvi ruudus ta asub.
3. R rida teksti tuleb paigutada V veergu nii, et veergude kõrgused erineksid ülimalt ühe võrra ning pikemad veerud asuksid vasakul. Leida i-ndas veerus olevate ridade arv.
4. Keeltes C ja C++ on funktsioon `clock()`, mis tagastab programmi käivitamise algusest kulunud protsessori tööaja; ajaühikute arv sekundis on konstant `CLOCKS_PER_SEC`. Järgmine funktsioon peaks töötama (nt mingile ülesandele järjest paremaid lahendeid otsima) umbes etteantud arvu sekundeid. Mis tal viga on ning kuidas teda paremaks muuta?

   ```
   void toota(int mituSekundit) {
     clock_t lopp = clock() + mituSekundit * CLOCKS_PER_SEC;
     while (clock() < lopp) teeNatukeTood();
   }
   ```

   Vihje: funktsiooni `clock()` väärtus võib üle joosta (nt Linuxis on `clock_t` tihti 32-bitine täisarv ja `CLOCKS_PER_SEC` = 1 000 000). Eeldame, et parameeter `mituSekundit` on väike.
5. Mitmel viisil saab antud täisarvu n väljendada järjestikuste naturaalarvude summana? (Näiteks, kui $n = 9$, siis $9 = 4 + 5 = 2 + 3 + 4$ — kokku 2 viisil.)
6. Kirjutada programm, mis leiab 10 000 esimest algarvu.
7. Kirjutada programm, mis lahutab antud täisarvu n algteguriteks. (Vihje: selleks ei ole vaja eelnevalt algarve leida.)
8. Kirjutada programm, mis loeb suvalise pikkusega kahendarvu ja leiab selle arvu 7-ga jagamisel tekkiva jäägi. Kogu programmis tohib kasutada ainult üht täisarvu tüüpi muutujat (ja lisaks üht sümboltüüpi muutujat kahendarvu numbrite sisselugemiseks).
9. Kirjutada programm, mis loeb täisarvu esituse a-süsteemis ja leiab selle arvu esituse b-süsteemis. Raskem variant: kogu programmis tohib kasutada ainult kolme täisarvu tüüpi muutujat (ja lisaks kaht sõnetüüpi muutujat — üht antud arvu a-süsteemi ja teist b-süsteemi esituse jaoks).
10. Kirjutada programm, mis leiab antud reaalarvu $0 < r < 1$ esituse b-süsteemis.
11. Balansseeritud 3-süsteemis märgitakse arve "numbrite" $+$, $-$ ja 0 abil ning arvu $\overline{++0-0}$ väärtus on $(+1) \cdot 3^4 + (+1) \cdot 3^3 + 0 \cdot 3^2 + (-1) \cdot 3^1 + 0 \cdot 3^0$. Kirjutada programm, mis leiab antud täisarvu a esituse balansseeritud 3-süsteemis.
12. Kirjutada programm suurte naturaalarvude võrdlemiseks.
13. Kirjutada programm suurte naturaalarvude korrutamiseks.
14. Kirjutada programm suurte naturaalarvude jagamiseks.
15. Kirjutada programm suurte täisarvude liitmiseks.
16. Kirjutada programm suurte täisarvude korrutamiseks.
17. Kirjutada programm suure täisarvu jagamiseks tavalise täisarvuga.
18. Kirjutada programm suure naturaalarvu astendamiseks tavalise naturaalarvuga, kasutades kiiret astendamist.