3.1 Tsükkel

KORDUV TEGEVUS

Kui püüda arvuti võimalikke plusse välja tuua, siis üheks oluliseks neist on kahtlemata võime mingeid tegevusi kiiresti ja korduvalt sooritada. Nii saab teha arvutusi, midagi andmetest otsida, erinevaid variante läbi vaadata jpm. Arvutid on läinud järjest kiiremaks ja nii saavad paljud asjad tehtud praktiliselt silmapilkselt. Samas on ülesandeid, mille lahendamiseks kulub ikkagi rohkem aega kui tahaks, isegi kui mitu arvutit korraga ülesannet lahendama panna. Näiteks ilmaennustus on selline keeruline ülesanne.

Kui tahta mingeid asju korduvalt teha, siis võivad ju programmid väga pikaks minna? Näiteks kui tahame, et programm väljastaks ekraanile viis korda üksteise alla "Tere!", siis kõlbaks selline programm.

```
print("Tere!")
print("Tere!")
print("Tere!")
print("Tere!")
print("Tere!")
```

Saja korra jaoks tuleks siis programm vastavalt pikem. Tegelikult on programmeerimiskeeltes olemas head võimalused selliste korduste lühemaks esituseks. Kõigepealt püütakse aru saada, mis on see korduv tegevus, mis päris samasugusena (või kindlate reeglite järgi muudetuna) tuleb ikka ja jälle teha. Eelmises näites oli selleks rida `print("Tere!")`. Teise asjana tuleb läbi mõelda, mitu korda me tahame seda tegevust teha. See võib olla meil ette teada, aga võib sõltuda ka mingitest välistest asjaoludest, näiteks kasutaja poolt antud vastusest. Näiteks pole mõtet parooli uuesti küsida, kui juba õige on sisestatud.

WHILE-TSÜKKEL

Korduvaid tegevusi realiseeritakse tsüklite abil. Vastavaid vahendeid võib konkreetses programmeerimiskeeles olla mitmeid. Näiteks Pythonis on olemas `while`-tsükkel ja `for`-tsükkel. Meie alustame eelkontrolliga tsüklist, mille põhimõte on teatud mõttes sarnane valikulausega. Sellist tsüklit kutsutaksegi `while`-tsükliks, sest reeglina on programmeerimiskeeltes just võtmesõna `while` selles tähenduses kasutusel. Erinevus `if`-lausest on selles, et pärast seda, kui tsükli sisus olevad laused on täidetud, minnakse uuesti tingimust kontrollima. Kui tingimus ikka veel kehtib, siis täidetakse sisu edasi jne. Kui mingil hetkel tingimust kontrollides see enam ei kehti, siis lõpetatakse tsükli täitmine. Tsükli sisus olevad laused peavad olema taandatud sarnaselt `if`-lause kehas olevatele lausetele.

Eelkontrolliga tsükli plokkskeem näeb välja selline:

Tsükli jätkamistingimus on (nagu ka `if`-lause tingimus) tõeväärtustüüpi. Kui tingimus on täidetud (tingimusavaldise väärtus on tõene), siis minnakse tsükli sisu täitma. Kui aga pole täidetud, siis minnakse tsüklist välja.

Tavaliselt on tingimus esitatud võrdlemisena, aga võib näiteks olla ka lihtsalt tõeväärtus `True`. Või hoopis tõeväärtus `False`. See viimane on küll üsna mõttetu: nii karm "piirivalvur", et kunagi kedagi edasi ei lubata. Jätkamistingimuse `True` puhul on tegemist lõpmatu tsükliga, sest tingimusavaldis on alati tõene. Teoreetiliselt jääbki see tsükkel igavesti tööle. Praktiliselt siiski ilmselt pannakse arvuti millalgi kinni, toimub elektrikatkestus vms. Kui me nüüd Pythonis meelega või kogemata sellise programmi teeme, mis igavesti tööle jääb, siis ei ole meil katkestamiseks siiski vaja arvutit kinni panna. Nimelt saame Thonnys programmi töö katkestada Stop-märgiga nupu või klahvikombinatsiooni *Ctrl + F2* abil. (Keskkonnas IDLE katkestatakse programmi töö *Ctrl + C* abil.) Tegelikult saab lõpmatut tsüklit kasutada ka päris sihipäraselt sellises olukorras, kus tuleb näiteks mingit sündmust aktiivselt oodata. Sellisel juhul on tsüklist väljasaamine teisiti organiseeritud.

Ülesanne

Kui while-tsükli jätkamistingimuseks on avaldis 3 > 5, siis

Vali tsükli sisu ei täideta kunagi

Vali tsükli sisu täidetakse teoreetiliselt lõpmatu arv kordi

Vali on tegemist vigase programmiga

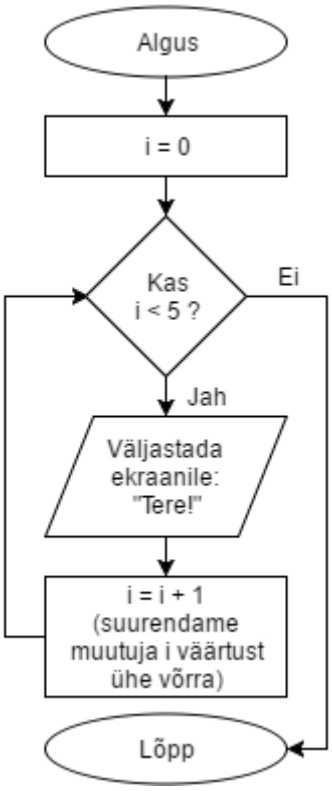
Lähme tagasi algse ülesande juurde. Me tahaks, et tsükli abil viis korda "Tere!" ekraanile tuleks. Seega peab olema midagi, mis tsükli sisus muutub nii, et just pärast viiendat korda

“piirivalvur” enam tsükli sisu juurde ei lubaks. Kuidas me inimlikult sellises olukorras loendaksime? Üks võimalus oleks näiteks sõrmedel lugeda ja nii meeles hoida, kui palju kordi juba tsükli sisu on täidetud. Põhimõtteliselt teeme sarnaselt ka programmeerides. Võtame kasutusele ühe muutuja, mille nimeks saagu `i`. Olgu `i` väärtus esialgu 0: `i = 0`. Igal tsükli sammul liidame väärtusele `1`. Varem olid näited, kus muutujale saime erinevaid väärtusi anda mingite teiste muutujate või näiteks arvude ja tehete abil. Nüüd aga on vaja selle sama muutuja väärtust muuta. Saame seda teha sellise avaldisega

```
i = i + 1
```

Võimalik, et selline võimalus vajab natuke harjumist. Kui vaatame koolimatemaatikat, siis võib see paista üsna kummaline, aga võrdusmärgi tähendus on siin teistsugune kui matemaatikas. Vasak pool näitab, et muutuja `i` saab uue väärtuse. Paremal pool on avaldis, millega see uus väärtus arvutatakse. Selles arvutamises kasutatakse ka muutuja `i` senist väärtust. Enne programmi kokkupanekut mõtleme veel jätkamistingimusele. Selleks sobib `i < 5`, sest kui `i` on esialgu `0` ja igal sammul liidetakse `1`, siis just 5 sammuga jõuame niikaugele, et tingimus `i < 5` ei ole enam täidetud. Panemegi nüüd programmi kokku. Olulisel kohal on taas koolon ja taandamine.

```
i = 0
while i < 5:
    print("Tere!")
    i = i + 1
```



On kokkulepe, et just `i` ongi tavaliselt sellise loendaja (*tsüklimuutuja*) nimeks. Püüame nüüd programmi tööd analüüsida sammude kaupa.

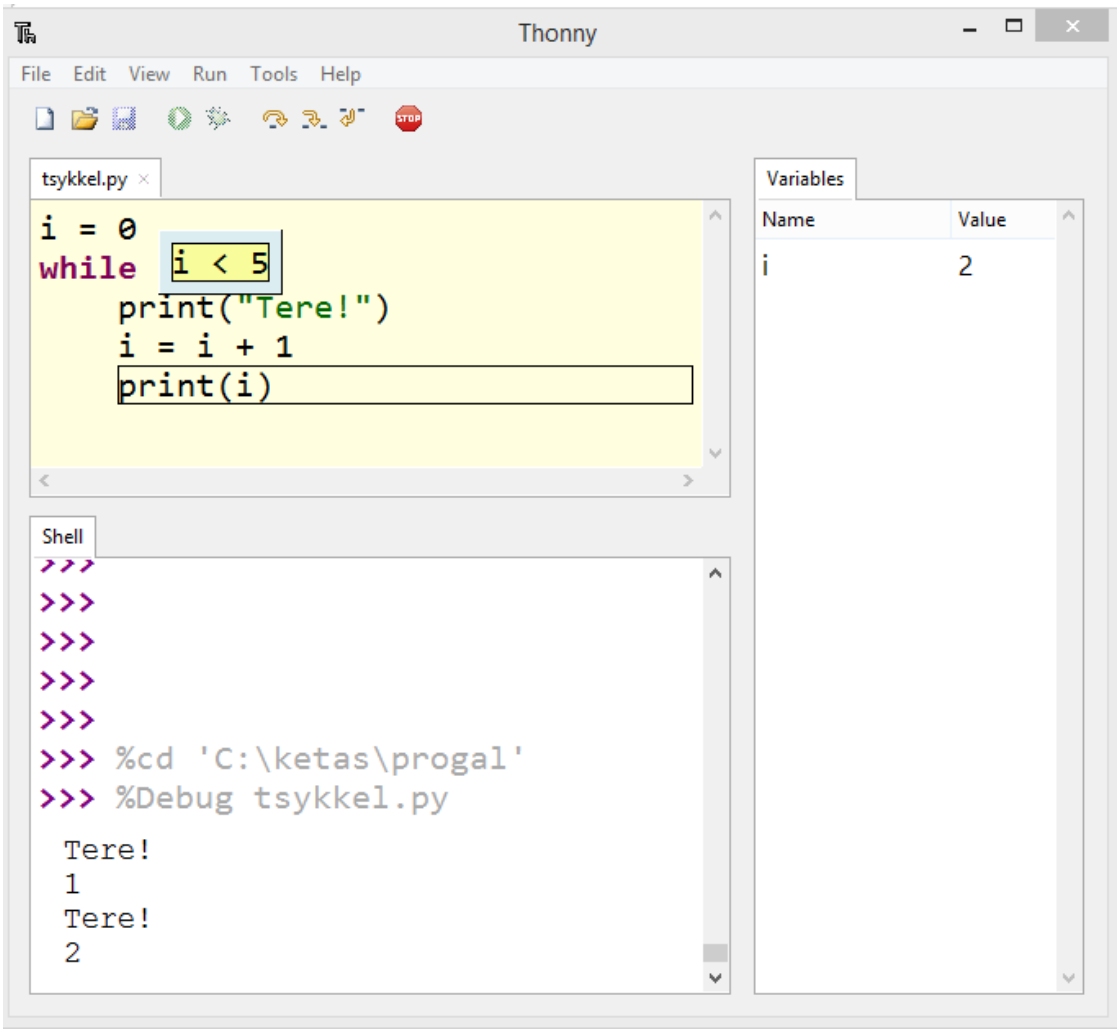
- Jätkamistingimus (`i < 5`) on täidetud, kui esimest korda tsükli juurde jõuame, sest `0 < 5`.
- Pärast esmakordset sisu täitmist on `i` väärtus `1` ja jätkamistingimus ikkagi täidetud, sest `1 < 5`.
- Pärast teist korda on `i` väärtus `2` ja ikka saame jätkata, kuna `2 < 5`.
- Ja siis `i` on `3` ja ikka `3 < 5`.
- Ja siis `i` on `4` ja ikka `4 < 5`.
- Ja siis `i` on `5` ja kontrollime, kas `i < 5`? Kas `5 < 5`? Ei ole, sest `5` ja `5` on võrdsed, seega võrratus `5 < 5` ei kehti ja jätkamistingimus on väär.

Lisame tsükli kehasse ühe rea, mis `i` väärtuse ekraanile tooks, siis saame seda paremini jälgida.

```
i = 0
while i < 5:
    print("Tere!")
    i = i + 1
    print(i)
```

Pange programm tööle.

Kui me programmi alles teeme, siis on mingites kohtades muutujate väärtuste väljastamine täiesti omal kohal, et olla kursis, mis seis vastaval hetkel on. Thonnys on muutuja väärtuste jälgimine sisse ehitatud ja seda saab nähtavale tuua *View*-menüüst valikuga *Variables*. Kuna programm töötab kiiresti, siis tavaliselt käivitades jäävad näha ainult programmi töö lõpus kehtivad väärtused. Selleks, et sammsammult programmi tööst ülevaadet saada, tuleb käivitada hoopis silumisrežiimis - putukaga nupuga või *Run*-menüüst *Debug current script*. Seejärel saab erineva ulatusega samme teha vastavate nuppude või *Run*-menüü valikute abil.



Kokkuvõttev video: http://www.uttv.ee/naita?id=23914

Kuna muutuja väärtuse muutmist eelmise väärtuse alusel tuleb päris sagedasti ette, siis on selleks ka lühemad variandid olemas. Näiteks `a = a + 3` asemel võime kirjutada `a += 3`. Samasugused variandid on ka lahutamise (`-=`), korrutamise (`*=`), jagamise (`/=`), täisarvulise jagamise (`//=`), jäägi leidmise (`%=`) ja astendamise (`**=`) jaoks.

Ülesanne

Millised muudatused programmis muudaksid ekraanile väljastatavat?

```
i = 0
while i < 5:
    print("Tere!")
    i = i + 1
    print(i)
```

Vali Tingimus on *i <= 5*

Vali Tingimus on *i != 5*

Vali Lause *i = i + 1* asemel on *i += 1*

Vali Muutuja nimeks on *i* asemel igal pool hoopis *j*

Ülesanne

Mis ilmub ekraanile?

```
i = 0
while i < 5:
i = i + 1
print(i)
```

Vali 0

Vali 5

Vali 6

Vali Arvud 1 kuni 5 üksteise all

Vali Veateade

Vali Mitte ükski eeltoodutest

Ülesanne

Mis ilmub ekraanile?

```
i = 0
while i < 5:
    i = i + 1
print(i)
```

Vali 0

Vali 5

Vali 6

Vali Arvud 1 kuni 5 üksteise all

Vali Veateade

Vali Mitte ükski eeltoodutest

Ülesanne

Mis ilmub ekraanile?

```
i = 0
while i <= 5:
    i = i + 1
print(i)
```

Vali 0

Vali 5

Vali 6

Vali Arvud 1 kuni 5 üksteise all

Vali Veateade

Vali Mitte ükski eeltoodutest

Ülesanne

Mis ilmub ekraanile?

```
i = 0
while i == 5:
    i = i + 1
print(i)
```

Vali 0

Vali 5

Vali 6

Vali Arvud 1 kuni 5 üksteise all

Vali Veateade

Vali Mitte ükski eeltoodutest

3.2 Tsükkel. Igal sammul juhtub midagi

KILPKONN TSÜKLIS

Eelmisel nädalal tutvusime kilpkonnagraafikaga ja tegime näiteprogrammina ruudu.

Näiteprogramm. Ruut

```
from turtle import *          # * lisamisel imporditakse kõik
kilpkonna käsud

forward(100)                  # Kilpkonn liigub edasi 100 pikslit
left(90)                      # Kilpkonn pöörab 90° vasakule
forward(100)                  # Kordame eelnevaid käske, sest ruudul
on neli külge
left(90)
forward(100)
left(90)
forward(100)

exitonclick()                 # Saame akna sulgeda hiireklõpsuga
```

On näha, et kilpkonn peab täitma korduvalt samu käske: minema 100 pikslit edasi ja pöörama seejärel 90° vasakule. Tegemist on tsüklilise tegevusega, seega saame kasutada tsüklit.

Kirjutame programmi ümber nii, et ruut joonistatakse `while` -tsüklit rakendades.

Näiteprogramm. Ruut II

```
from turtle import *

i = 0                         # Muutuja i väärtus on esialgu 0
while i < 4:                  # Kilpkonn joonistab tsükli abil ruudu.
Tsükli keha läbitakse neli korda.
    forward(100)
    left(90)
    i = i + 1                 # Muutuja i väärtust suurendatakse ühe
võrra

exitonclick()
```

Pange see programm tööle ja püüdke seda modifitseerida nii, et joonistataks hoopis võrdkülgne kolmnurk. Mis sellisel juhul on ruudust erinev? Külgi on kolm ja pöörama peab ... proovige ise!

IGAL SAMMUL JUHTUB MIDAGI

Nagu juba eelmiste näidete puhul oli näha, juhtub igal sammul midagi. Võib juhtuda samu asju, võib natuke erinevaid asju juhtuda. Näiteks "Tere!" väljastamisel juhtus see nii igal sammul. Samuti liikus kilpkonn igal tsüklisammul 100 pikslit edasi ja pööras 90 kraadi vasakule. Kui tsüklis oli aga käsk `print(i)`, siis igal sammul toimus väljastamine, aga tsüklimuutuja `i` väärtus oli eri sammudel erinev. Tsüklimuutuja väärtuse muutumine tagati reaga `i = i + 1`.

Modifitseerime ruudu joonistamise programmi nii, et igal sammul tehtav sõltuks ka tsüklimuutuja `i` tolle hetke väärtusest. Asendame rea `forward(100)` reaga `forward(100 * i)`. Pange programm tööle ja proovige toimunut seletada:

```
from turtle import *

i = 0                          # Muutuja i väärtus on esialgu 0
while i < 4:                   # Tsükli keha läbitakse neli korda.
    forward(100 * i)
    left(90)
    i = i + 1                  # Muutuja i väärtust suurendatakse ühe
võrra

exitonclick()
```

Muutke programmis tsükli sammude arvu. Selleks tuleb tingimuses `i < 4` arv `4` asendada suurema või väiksema arvuga. Samuti võib `100` piksli asemel kasutada mõnda väiksemat arvu, et tulemus paremini nähtav oleks.

Tsükli kehas ei pruugi alati midagi silmnähtavat juhtuda. Näiteks võidakse hoopis mingeid asju "meelde jätta" ja pärast tsüklit kasutada. Nii on järgmises programmis võetud kasutusele muutuja `summa`, millesse hakatakse "koguma" summat.

```
i = 0
summa = 0
while i < 5:
    summa = summa + i
    i = i + 1
print(summa)
```

Ülesanne

Mis ilmub ekraanile?

```
i = 0
summa = 0
while i < 5:
  summa = summa + i
  i = i + 1
print(i)
```

Vali 0

Vali 4

Vali 5

Vali 6

Vali Veateade

Ülesanne

Mis ilmub ekraanile?

```
i = 0
summa = 0
while summa < 5:
  summa = summa + i
  i = i + 1
print(i)
```

Vali 0

Vali 4

Vali 5

Vali 6

Vali Veateade

PIKEM SAMM! VÕI HOOPIS LÜHEM?

Eelmistes näidetes on tsüklimuutuja muutunud iga sammuga ühe võrra. See on küll kõige sagedasem variant, kuid kaugeltki mitte ainus. Põhimõtteliselt saame tsüklimuutujat muuta ükskõik millise lubatud tehtega. Proovige näiteks nii.

```
i = 0
while i < 5:
    i = i + 2
    print(i)
```

Samm võib ka hoopis lühem olla: `i = i + 0.5`. Või näiteks võib igal sammul `i` väärtus kolm korda kasvada: `i = i * 3`. Korrutamise puhul peab aga hoiduma esialgsest väärtusest `0`, sest korrutamine seda ei muuda.

Teeme programmi, mis väljastab ekraanile 1, 3, 9, 27 ... . Näeme, et iga eelmine tulemus korrutatakse järgmise saamiseks kolmega. Kasutajalt küsitakse näiteks, millisest arvust peab tulemus väiksemaks jääma:

```
piir = int(input("Millisest arvust väiksemaks peab tulemus jääma?
"))
tulemus = 1
while tulemus < piir:
    print(tulemus)
    tulemus = tulemus * 3
```

Nagu näeme, on siin oluliseks muutujaks `tulemus` ise. Selleks, et lugeda, mitmendal sammul me oleme (mitmendat tulemust oleme väljastamas) ja näiteks hoopis tulemuste arvu piirata, peab kasutusel olema teine muutuja.

```
piir = int(input("Millisest arvust väiksemaks peab tulemus jääma?
"))
tulemus = 1
jrk = 0
while tulemus < piir:
    print(str(jrk) + ". rida on " + str(tulemus))
    tulemus = tulemus * 3
    jrk = jrk + 1
```

Kui nüüd tahta, et arvutataks ainult näiteks 6 järjestikust tulemust, peaks vastava tingimuse asendama: `while tulemus < piir` asemel `while jrk < 6`.

3.3 Tsükkel. Mitu korda?

SUUR ARV?

Alustame elulisema näitega, kus tsüklimuutuja suureneb igal sammul ühe võrra, aga algväärtus pole `0`, vaid hoopis `2016`. Oletame, et aastal 2016 on kedagi 1 311 800 (https://www.stat.ee/12808). Iibe kordaja näitab selle arvu muutumist aasta jooksul 1000 inimese kohta (positiivse iibe kordaja korral on arv kasvanud, negatiivse korral langenud).

```
aasta = 2016                # Ei pea sugugi nullist alustama
arv = 1311800
iibe_kordaja = -1.14        # Promillides (muut 1000 kohta)
while aasta < 2080:
    arv = arv + arv * iibe_kordaja / 1000
    aasta = aasta + 1
    print("Aastal " + str(aasta) + " on meid " + str(round(arv)) +
".")
```

Proovige ka teiste väärtustega!

Kui aga näiteks tahaks teada, mis aastal oleks eestlasi 2 000 000, kui iibe kordaja oleks 6 promilli (tegelikult pole nii kõrget loomulikku iivet vähemalt viimase saja aasta jooksul Eestis olnud):

```
aasta = 2016
arv = 1311800
iibe_kordaja = 6            # Promillides
while arv < 2000000:
    arv = arv + arv * iibe_kordaja / 1000
    aasta = aasta + 1
    print("Aastal " + str(aasta) + " on arv " + str(round(arv)) +
".")
```

Täpsema prognoosi saamiseks on muidugi vaja arvestada väga erinevaid asjaolusid (huvi korral vt ka https://www.stat.ee/76578).

MITU KORDA?

Eelmistes näidetes oli meil tsükli korduste arv tegelikult juba algul teada või siis kindlatel alustel arvutatav nagu viimases näites. Nüüd aga püüame teha programmi, kus tsükli läbimiste arv sõltub kasutaja sisestatud vastustest. Võtame aluseks valikulause materjali juures olnud programmi, milles küsitakse PIN-koodi. Seal küsiti PIN-koodi üks kord ja lõplik otsus tehti juba ühe pakkumise järel:

```
print("Sisesta PIN-kood:")
sisestatud_pin = input()
if sisestatud_pin == "1234":
    print("Sisenesid pangaautomaati!")
else:
    print("Vale parool! Enesehävitusrežiim aktiveeritud: 3 … 2 … 1
....")
```

Tavaliselt siiski antakse ka eksimisvõimalusi ja vale koodi puhul küsitakse uuesti. Püüamegi nüüd selle programmi vastavalt ümber kirjutada. Ilmselt on siin kasu tsüklist. Antud juhul peame tsükliliselt käituma (uuesti küsima) vaid siis, kui sisestatud PIN-kood ei ole õige. Jätkamistingimuseks sobiks seega `sisestatud_pin != "1234"`. Programm ise oleks näiteks järgmine:

```
print("Sisesta PIN-kood:")
sisestatud_pin = input()
while sisestatud_pin != "1234":
    print("Sisesta PIN-kood:")
    sisestatud_pin = input()
print("Sisenesid pangaautomaati!")
```

Püüdke samm-sammult läbi mõelda, kuidas selline programm töötab. Vajadusel joonistage selle plokkskeem. Proovige see programm tööle panna ja testige erinevate PIN-koodidega. Proovige ka oma pangakaardi tegeliku koodiga! :-) Või ärge ikka proovige!

Programmi vaadates näeme, et järgmine lõik on kahes kohas:

```
print("Sisesta PIN-kood:")
sisestatud_pin = input()
```

Võime proovida tsükli eest selle lõigu ära jätta:

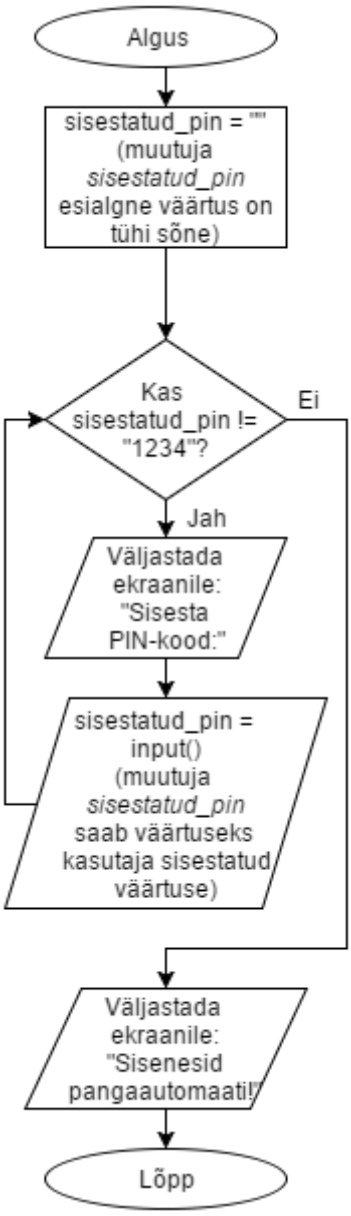
```
while sisestatud_pin != "1234":
    print("Sisesta PIN-kood:")
    sisestatud_pin = input()
print("Sisenesid pangaautomaati!")
```

Sellisel juhul aga tuleb veateade, sest muutujal `sisestatud_pin` ei ole väärtust, kui seda `while`-tingimuses esimest korda kontrollida tahetakse:

```
Traceback (most recent call last):
  File "C:/Python33/pin.py", line 1, in <module>
    while sisestatud_pin != "1234":
NameError: name 'sisestatud_pin' is not defined
```

Anname muutujale `sisestatud_pin` esialgseks väärtuseks tühja sõne. Sellega garanteerime, et muutujal `sisestatud_pin` on alati väärtus ja `while` jätkamistingimus on esialgu kindlasti tõene, sest tühi sõne ei ole võrdne sõnega `"1234"`.

```
sisestatud_pin = ""
while sisestatud_pin != "1234":
    print("Sisesta PIN-kood:")
    sisestatud_pin = input()
print("Sisenesid pangaautomaati!")
```



Nüüd ei pääse ilma parooli sisestamata edasi. Paraku on süsteem ebaturvaline, sest katsetada saab suvaline arv kordi. Püüame ka katsete arvu piirata:

```
sisestatud_pin = ""
katseid = 3
while sisestatud_pin != "1234" and katseid > 0:
    print("Sisesta PIN-kood:")
    print("Jäänud on " + str(katseid) + " katset.")
    katseid -= 1
    sisestatud_pin = input()
print("Sisenesid pangaautomaati!")
```

Nüüd koosneb jätkamistingimus kahest osast - endiselt kontrollitakse, ega sisestatud PIN-kood õige ei ole, aga lisaks kontrollitakse ka seda, mitu korda veel vastata saaks. Enne tsüklit on kordade arvuks määratud 3 ja pärast igat tsükli keha täitmist väheneb see arv 1 võrra. Esimesel korral saab muutuja `katseid` väärtuseks `2`, teisel korral `1` ja kolmandal korral `0`.

Jätkamistingimuses kasutatakse võtmesõna `and`, mis tähendab, et tingimuse kehtimiseks peab nii selles sõnast paremal pool kui vasakul pool olev tingimus tõene olema, teisisõnu peavad mõlemad *osaavaldised* olema tõesed. Tõesti, selleks et PIN-koodi peaks uuesti küsima, ei tohi olla veel õiget sisestatud ja järele jäänud katseid peab olema rohkem kui 0. Nagu eelmise nädala materjalides juttu oli, võib ka siin tingimus olla ükskõik kui keeruline. Tehetega `and`, `or` ja `not` saab väga erinevaid avaldisi tekitada.

Kui eelnevas näites midagi segaseks jäi, siis saab vaadata kokkuvõtvat videot programmi koostamise kohta: https://youtu.be/OJhIV5IcE5o

Ülesanne

Mida eelmine programm teeb, kui kolm korda järjest vale kood sisestada?

- Vali — Väljastab ekraanile "Sisenesid pangaautomaati!"
- Vali — Küsib ka neljandat korda.
- Vali — Lõpetab küsimise ja ei väljasta midagi.
- Vali — Veateade

Mure on nüüd selles, et küsimiste arv on küll piiratud, aga isegi kui valet koodi kolm korda sisestatakse, saadakse ikka pangaautomaati sisse. Muudame programmi nii, et arvestataks, kas lõpuks sisestati õige kood või mitte. Kui ei sisestatud, siis on kuri karjas ja lahkumiseks antakse 10 sekundit! :-) Programmi saab üheks sekundiks "uinutada" käsuga `sleep(1)`, kuid selle kasutamiseks tuleb see importida moodulist `time` nii: `from time import sleep`.

```
from time import sleep
sisestatud_pin = ""
katseid = 3
while sisestatud_pin != "1234" and katseid > 0:
    print("Sisesta PIN-kood:")
    print("Jäänud on " + str(katseid) + " katset.")
    katseid -= 1
    sisestatud_pin = input()
if sisestatud_pin == "1234":
    print("Sisenesid pangaautomaati!")
else:
    print("Enesehävitusrežiim aktiveeritud:")
    i = 10
    while i > 0:
        print(i)
        i -= 1
        sleep(1)
```

Huvi korral mõtisklege ja katsetage, mida teeks programm teisiti, kui

```
if sisestatud_pin == "1234":
```

asemel oleks

```
if katseid > 0:
```

Eelmises programmis oli viimane tsükkel valikulause `else`-osa sees. Näeme, et taane on vastavalt läinud veel kaugemale. Selline mitmetasemeline struktuur on programmides täiesti tavaline. Vabalt võib olla ka näiteks `while`-tsükkel, mille sees on tingimuslause, mille sees on veel üks tsükkel, aga sellise programmi käitumise ette ennustamine võib muutuda keeruliseks, seega tuleks võimaluse korral keerulisemast struktuurist hoiduda.

Klassikalises arvamismängus on aga kasu tingimuslausest tsükli sees:

```
from random import randint

arv = randint(1,19)    # Juhuslik täisarv
print("Mõtlen ühele 20-st väiksemale naturaalarvule. Arva ära!")
arvamus = int(input())

while arvamus != arv:
    if arv > arvamus:
        print("Minu arv on suurem!")
    else:
        print("Minu arv on väiksem!")

    print("Arva veel!")
    arvamus = int(input())
print("Õige! Tubli!")
```

Proovige mängida, aga ärge sellest sõltuvusse sattuge! Proovige mängu modifitseerida. Näiteks las programm loeb vastamiste arvu ka ja reageerib vastavalt sellele. (Kui arvatakse arv esimese korraga ära, siis näiteks võiks mängijal soovitada oma selgeltnägija võimeid ehk laiemaltki kasutada.) Huvitav oleks ka teistpidi mäng, kus arvuti oleks arvaja rollis ja arvaks võimalikult mõistliku strateegiaga.

Ülesanne

Mõelda oma elule ja ümbritsevale ning kirjeldada ühte tsüklilist protsessi. Seejuures märkida, kas tsükli läbimiste arv on enne teada või selgub täitmise ajal. Protsess ei pea olema (lihtsasti) programmeeritav. Lahendus esitada Moodle'is vastava testiküsimuse vastusena.

3.4 Juhuslikust arvust veel

VISKAME TÄRINGUID

Eelmisel nädalal tutvusime juhuslike arvudega, mida saab kasutada näiteks erinevate mänguliste programmide loomisel. Proovime juhuslikkust nüüd tsükliga siduda. Esialgu teeme programmi, mis viskab viis korda täringut. Olgu see tavaline kuuetahuline täring.

```
from random import randint
i = 1
while i < 6:
    print("Täringu " + str(i) + ". viskel saadi " +
str(randint(1, 6)) + ".")
    i += 1
```

Kui me midagi muud visketulemusega teha ei taha, siis võimegi selle väljundisse sisse kirjutada. Sageli aga tahame viske tulemusega midagi teha. Näiteks tulemused kokku liita ja kuue saamisel eriliselt rõõmustada.

```
from random import randint
i = 1
tulemus = 0
while i < 6:
    vise = randint(1, 6)
    print("Täringu " + str(i) + ". viskel saadi " + str(vise) + ".")
    if vise == 6:
        print("Hurraaaaaaaa!")
    tulemus += vise
    i += 1
print("Kogutulemus on " + str(tulemus) + ".")
```

Soovijad võivad täringu nii ümber teha, et täringul oleks mingi muu arv tahke. Näiteks mängus Dungeons & Dragons on ka 4-, 8-, 10-, 12- ja 20-tahulised täringud. Võib ka soovitud tahkude arvu kasutajalt küsida.

KUI TÕENÄOSUSED POLE VÕRDSED

Eelmises näites oli meil täring, mille iga tahk võis tulla võrdse tõenäosusega. Kuuetahulise täringu puhul oli iga tahu tulemise tõenäosus 1/6. Selle tagas funktsioon `randint(1, 6)`. Mündiviske puhul saame ausa mündi funktsiooni `randint(1, 2)` abil.

Nüüd aga kujutame ette, et keegi on mündiga manipuleerinud ning teinud nii, et kulli ja kirja tõenäosused pole enam võrdsed. Olgu näiteks kirja tulemise tõenäosus 73% (ehk 0,73). Sellise mündi viskamist saab programmiga simuleerida mitmel viisil. Näiteks võiksime

kasutada funktsiooni `randint(1, 100)`, mis annab võrdse tõenäosusega ühe täisarvu lõigus 1 kuni 100. Saadud arv on 73 või väiksem tõenäosusega 73%. Nii saame järgmise programmi:

```
from random import randint
tõenäosus = 73
i = 0
while i < 10:
    if randint(1, 100) <= tõenäosus:
        print("kiri")
    else:
        print("kull")
    i += 1
```

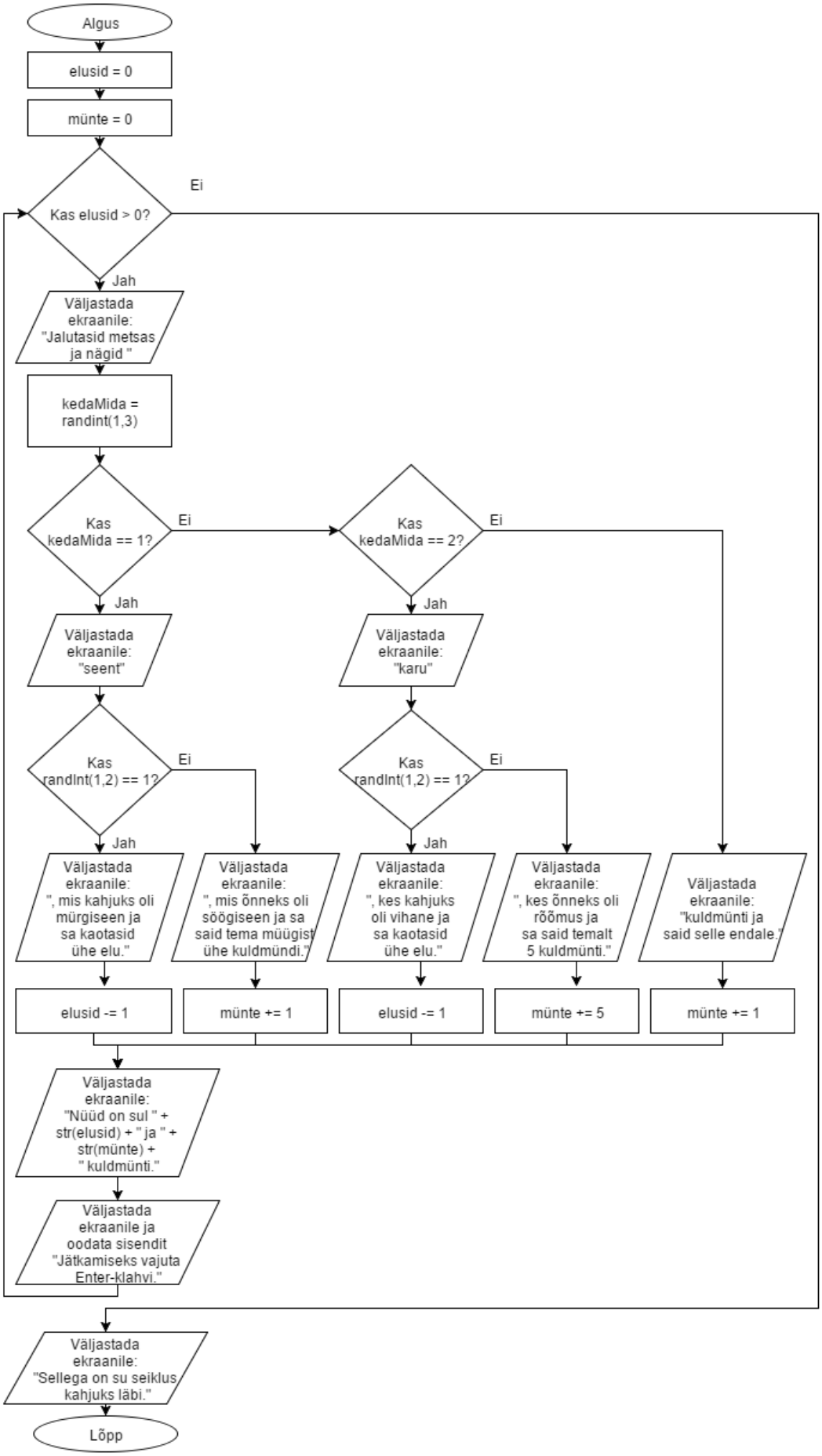
Kui seda programmi korduvalt tööle panna, siis näeme, et kümnest viskest kipub tõesti kirjasid kullidest rohkem olema. Võite ka kirjade loenduri lisada, mis seda aitab jälgida. Siiski juhtub vahel, et kirjasid on samapalju või vähem kui kulle. Ka see on oodatav.

Samalaadselt saame simuleerida ka teiste tõenäosuslike protsesside tulemusi.

SEIKLUS METSAS

Järgmine programm on eelnevatest pikem ja simuleerib võimalikku jalutuskäiku metsas. Hoiatus! Tegemist on metsaga, kus leidub nii ohte kui ahvatlusi. Soovitame aga siiski katsetada. Ennekõike aga püüdke programmist aru saada.

```
from random import randint
elusid = 3
münte = 0
while elusid > 0:
    print("Jalutasid metsas ja nägid ", end = "") # ei vaheta rida
    kedaMida = randint(1, 3)
    if kedaMida == 1:
        print("seent", end = "") # ei vaheta rida
        if randint(1, 2) == 1:
            print(", mis kahjuks oli mürgiseen ja sa kaotasid ühe
elu.")
            elusid -= 1
        else:
            print(", mis õnneks oli söögiseen ja sa said tema
müügist ühe kuldmündi.")
            münte += 1
    elif kedaMida == 2:
        print("karu", end = "") # ei vaheta rida
        if randint(1, 2) == 1:
            print(", kes kahjuks oli vihane ja sa kaotasid ühe
elu.")
            elusid -= 1
        else:
            print(", kes õnneks oli rõõmus ja sa said temalt 5
kuldmünti.")
            münte += 5
    else:
        print("kuldmünti ja said selle endale.")
        münte +=1
    print("Nüüd on sul " + str(elusid) + "elu ja " + str(münte) + "
kuldmünti.")
    input("Jätkamiseks vajuta Enter-klahvi.")
print("Sellega on su seiklus kahjuks läbi.")
```


Algus
elusid = 0
münte = 0
Kas elusid > 0?
Ei
Jah
Väljastada ekraanile: "Jalutasid metsas ja nägid "
kedaMida = randint(1,3)
Kas kedaMida == 1?
Ei
Kas kedaMida == 2?
Ei
Jah
Väljastada ekraanile: "seent"
Jah
Väljastada ekraanile: "karu"
Kas randInt(1,2) == 1?
Ei
Kas randInt(1,2) == 1?
Ei
Jah
Väljastada ekraanile: ", mis kahjuks oli mürgiseen ja sa kaotasid ühe elu."
Väljastada ekraanile: ", mis õnneks oli söögiseen ja sa said tema müügist ühe kuldmündi."
Jah
Väljastada ekraanile: ", kes kahjuks oli vihane ja sa kaotasid ühe elu."
Väljastada ekraanile: ", kes õnneks oli rõõmus ja sa said temalt 5 kuldmünti."
Väljastada ekraanile: "kuldmünti ja said selle endale."
elusid -= 1
münte += 1
elusid -= 1
münte += 5
münte += 1
Väljastada ekraanile: "Nüüd on sul " + str(elusid) + " ja " + str(münte) + " kuldmünti."
Väljastada ekraanile ja oodata sisendit "Jätkamiseks vajuta Enter-klahvi."
Väljastada ekraanile: "Sellega on su seiklus kahjuks läbi."
Lõpp

Soovitatav on püüda seda programmi täiendada. Näiteks teha nii, et

- teatud müntide arvu juures kuulutatakse mängija võitjaks. Hetkel on selgelt letaalse seiklusega tegemist;
- ohte või ahvatlusi oleks rohkem;
- elu kaotamise tõenäosus oleks väiksem;
- ka kasutaja saaks saatuse määramisel kuidagi kaasa lüüa.

Võite ka ise veel täiendusi juurde mõelda.

Ülesanne

Keda või mida metsas nähakse, kui muutuja kedaMida väärtus on 3?

Vali Seent

Vali Karu

Vali Kuldmünti

Vali Väärtus ei saa kunagi 3 olla

3.5 Labürint

MIS ON LABÜRINT?

Labürindiks nimetatakse keerdkäikudega ehitist ehk keerdkäigustikku (vt ÕS 2013) ning sõna labürint tuleneb kreekakeelsest sõnast *labyrinthos*.

Sageli jagatakse labürindid kahte erinevasse liiki, mis erinevad teineteisest läbitavuse poolest.

- Leidub labürinte (ingl *labyrinth*), mille puhul pole eesmärgiks inimese eksitamine, vaid rändaja juhtimine ühe võimaliku tee kaudu. Selliseid keerdkäigustikke tuntakse juba tuhandeid aastaid. Eestiski leidub selline labürint Aegna saarel, mille ehitamise aega täpselt ei teata (vt kivilabürint).
- Samas on rajatud labürinte (ingl *maze*), mille eesmärgiks on sinna sattunud külalist segadusse ajada erinevate võimalike teedega. Neid võib nimetada ka labürintmõistatusteks ning üle maailma on neid rajatud hekklabürintidena näiteks aedadesse. Ka Kreeka mütoloogias esinev Minotauruse labürint on mõeldud selleks, et sinna sattunud inimene eksiks ja ei leiaks väljapääsu.

Meie keskendumegi niisugustele labürintidele, mis püüavad inimesi eksitada.

MAAILMA SUURIMAD

Labürinte on tekitatud erinevat moodi. Näiteks on neid ehitatud jääst, kasvatatud hekina ja rajatud maisipõllule. Tutvustame mõnda rekordilist labürinti.

Maailma suurim jääst valmistatud labürint tehti Buffalos, USAs *Buffalo Powder Keg* festivali raames 26. veebruaril 2010. aastal. Labürindi pindala oli 1194,33 m², laius 25,85 m ja pikkus 46,21 m. Müüride kõrguseks oli 1,83 m ning selle ehitamiseks kulus 2171 jääplokki, kusjuures üks plokk kaalus 136 kg.

Allikas: http://www.guinnessworldrecords.com/world-records/4000/largest-maze-ice-maze

Maailma suurim hekklabürint on Ananassi Aia Labürint (ingl *Pineapple Garden Maze*), mis asub Havail Wahiawas. See kuulub firmale Dole, mis tegeleb puuviljade ja köögiviljade kasvatamise ja turustamisega. Labürindi pindala on 12,74 ha ja pikima tee pikkus on ligi 4 km.

Allikas: http://www.guinnessworldrecords.com/world-records/1/largest-maze-permanent-hedge-maze

Suurim maisipõllule rajatud labürint on 24,28 ha suurune. See loodi ettevõtte *Cool Patch Pumpkins* poolt ja asub California osariigis Dixonis.

Allikas: http://www.guinnessworldrecords.com/world-records/1000/largest-maze-temporary-corn-crop-maze

Labürindi teematikaga seoses on valminud 2014. aastal ka film "Labürindijooksja" (ingl *The Maze Runner*), mis põhineb James Dashneri poolt kirjutatud samanimelisel ulmetriloogial. Filmis leiab poiss nimega Thomas, kelle mälu on kustutatud, end samavanuliste poste seast, kes kõik on väljapääsu otsides ühes suures labürindis lõksus. Võite filmi kohta täpsemalt uurida siit ning vaadata filmi treilerit: https://youtu.be/AwwbhhjQ9Xk

LABÜRINDIST PÄÄSEMINE

Labürint kui ehitis pole otseselt seotud programmeerimisega, kuigi labürinti saab arvuti abil planeerida. Selliseid generaatoreid on veebiski.

Suuremat huvi pakuvad aga keerdkäigustikust väljapääsemise ülesanded, sest nende lahendused on algoritmilised. Järgmisena tutvustatakse mõnda lahenduseeskirja, mis aitavad eksinud ränduril leida keerulisest labürindist pääsetee.

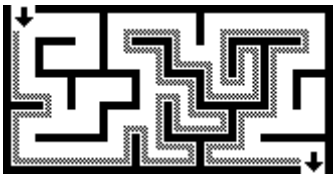
Hiire algoritm

Üks lihtsamaid algoritme, mida saab väljapääsu leidmiseks kasutada, on juhuslik hiire algoritm (ingl *Random mouse algorithm*). Nagu selle nimigi ütleb, siis tegemist on viisiga, kuidas hiir otsiks labürindist väljapääsu. Selle põhimõtteks on liikuda labürindis otse seinani ning seejärel pöörata suvalisse suunda ja liikuda taas otse seinani. Seesugust tegevust

korratakse kuni väljapääsu leidmiseni. Teisisõnu on idee selles, et uidatakse keerdkäigustikus ringi kuni avastatakse väljapääs. Tegelikult on siin tegemist tsüklilise käitumismustriga. Igal tsükli sammul minnakse otse seinani ja siis pööratakse suvalisse suunda. Selle algoritmi nõrkuseks on see, et sageli kulub palju aega enne väljapääsu leidmist, eriti kui on tegemist väga suure labürindiga.

Seinajärgija algoritm

Teine lihtne võimalus, kuidas labürinti läbida, on kasutada seinajärgija algoritmi. Esmalt tuleb valida parem või vasak käsi ja hoida see käsi pidevas kontaktis labürindi seinaga väljapääsu leidmiseni.



Allikas: https://en.wikipedia.org/wiki/Maze_solving_algorithm#/media/File:Maze01-02.png

Selline algoritm aga ei tööta, kui algus- ja lõpp-punkt pole omavahel seinapidi ühendatud.

Tupikute täitmise algoritm

Leidub labürindi lahendusalgoritme, mis ei aita tundmatus keerdkäigustikus teed kaotanud inimesel pääseda, sest tal pole ülevaadet kogu labürindist. Küll aga võimaldavad need leida tee, kui kogu labürindi kaart on ees. Näiteks võib selleks luua vastava programmi. Tupikute täitmise algoritmi puhul vaadatakse kogu labürinti korraga ning eesmärgiks on

1. leida kõik tupikud,
2. arvata kogu tee tupikust esimese ristmikuni kaardilt välja.

Selle meetodi mõistmiseks vaadake järgmist videot: https://youtu.be/yqZDYcpCGAI

Trémaux' algoritm

Algoritm on saanud oma nime selle looja Charles Pierre Trémaux' järgi, kes oli 19. sajandi prantsuse matemaatik. Tema algoritmi põhimõte on selles, et labürindi lahendamiseks peab eksinud rändur läbitud tee märkimiseks joonistama enda järele joone. Juhul, kui satutakse tupikusse, siis pööratakse ümber ja minnakse tuldud teed tagasi. Kui leitakse ristmik, kus varem käidud pole, siis valitakse suvaline suund (kust ei tuldud) ning jätkatakse teed. Kui kõnnitakse mööda teed, mida on juba külastatud (näiteks on üks kord joonega märgitud) ja satutakse ristmikule, siis valitakse uus tee, kui see on saadaval (pole joonega märgitud), ning minnakse mööda seda teed. Vastasel juhul minnakse mööda vana teed, mis oli ühel korral märgitud. Kõik teed on kas märkimata, märgitud üks kord (käidud on seda teed vaid üks kord) või märgitud kaks korda, mis tähendab seda, et seda mööda on käidud ja siis tagasi tuldud. Lõpptulemusena saadakse ühe joonega märgitud tee, mis ühendab algust ja lõppu. Algoritmi paremaks mõistmiseks vaadake selgitavat videot: https://youtu.be/6OzpKm4te-E

Millist algoritmi kasutaksite, kui satuksite labürinti?

Ülesanne

Labürindis liikumist saab harjutada näiteks Blockly: Maze mängu abil.

Edasijõudnutele

Need, kelle jaoks on tsükkel ja moodulite importimine selged ja soovivad oma programmerimisoskusi proovile panna, võiksid uurida lisamaterjali "Pykkar", kus saab luua ise labürindi ning kirjutada programmi, mis selle lahendab.

ALLIKAD

1. http://www.labyrinthos.net/typology.html
2. http://en.wikipedia.org/wiki/Maze_solving_algorithm
3. http://www.guinnessworldrecords.com/world-records/4000/largest-maze-ice-maze
4. http://www.guinnessworldrecords.com/world-records/1/largest-maze-permanent-hedge-maze
5. http://www.guinnessworldrecords.com/world-records/1000/largest-maze-temporary-corn-crop-maze

3.6 Lugu: alkeemia III

LUGU: ALKEEMIA III

Selle loo on kirjutanud Ander Peedumäe.

"Kogunege siia!" kutsus habemik mees tudengeid raudhingedega ukse juurde. Sagimine kajas kiviseintelt vastu ning jäi samm-sammult vaiksemaks.

„Teie, hakkajad alkeemikud, peate õppima kasutama veel üht vahendit. Materjal, mille peale meie linn on püstitatud - savi."

Alkeemik haaras lingist ning lukk lõksatas iseenesest lahti. Krigiseva ukse vahelt immitses niisket õhku ning kostus tuule ulgumine kõrgete löövide all.

Pärast sisenemist ei soovinud õpilased sissepääsust palju kaugemale liikuda, suhtudes ettevaatusega üüratu ruumi pimedusse. Ainus, kelle uudishimu kaalus üle kõik mõeldavad ohud, oli Tea. Varjudes liikudes suutis naine kandiliste kivisammaste toel edukalt vältida komistamist konarlikul kivipõrandal. Rühma lõpus loivas väsinud välimusega mehemürakas nimega Ra, kes öösiti linna vahtkonna tallides töötas. Ülikoolielu ei olnud väiksest maakonnast pärit inimesele kõige odavam ning seetõttu ei suutnud ta loengutes alati silmi lahti hoida, kuid öistel tundidel sai kergema töö kõrvale õppetükke piisavalt korrata. Tema selja taga sulgus raske uks ning kogu seltskond tardus paigale. Pimedus. Vaikus.

Kauguses süttis tuluke. Pärast esimest hakkas põlema ka teine ning üksteise järel lahvatasid leekidesse kõik tõrvikud kõrgete sammaste külgedel. Avanes vaatepilt hiiglaslikule saalile, mis sarnanes osalt katedraalile, mille alla ta ehitatud oli. Kui maapealne hoone oli heledates värvides ning kuldsete karniisidega, siis siin oli kõik ehitatud kividest ning savist.

Kaugelt, vahekäigu lõpust kostis alkeemiameistri kare hääl: „Astuge julgelt edasi ja pange tähele, mida ma räägin.“

Seltskond hakkas aeglaselt hargnema ning jaotus sammassaalis ühtlaselt. Tea imetles seina ääres kõrguvaid savist hobused ja üritas meelde tuletada käsitööteemalisi raamatuid, mida ta isa mõisas lugenud oli. Ta pani tähele, et kujud olid peensusteni välja voolitud silmadest kuni sabani. Lakad olid nöörist punutud ja jupphaaval savisse torgatud ning kapjade asemel olid suured sepistatud rauatükid.

„Kuninga sõjaväkke värvatud alkeemikud saavad Ülikoolilt kaasa väsimatu ratsu, kes ei kohku ka siis, kui nooled varjutavad taeva. Paljud rändalkeemikud lasevad endale säärase looma reisikaaslaseks valmistada. Siiski ei ole see nii lihtne, et lase meistritel hobune valmistada ning anna ohjad sõdurile. Savi ei mõtle. Savi ei oska teha valikuid. Kui alkeemik soovib mõistlikku looma, siis peab ta sellele ise mõistuse andma. See on teie ülesanne. Kaugetes maades võib see päästa teie elu.“

Seinalt, pimedatest aknalaadsetest avadest vaatasid alla suured vasknokkadega kotkad ja öökullid ning põletatud savist rongad. Sammaste vahel pidasid vahti hallist savist hundid, hõbedased silmad tõrvikuvalguses helkimas.

Tudengid kogunesid poolringi õppejõu ees. Mees köhatas hääle puhtaks ning võttis vesti taskust välja pisikese savist kilpkonna.

„Kilpkonn! See oli esimene teadaolev saviloom, kes pandi maagia abil liikuma. Alguses kümmekond sammu edasi ja sama palju tagasi. Sündis alkeemiaharu, mis uurib saviolendite liikumist, mõistust ja võimete piire. Kilpkonn õpetati end pöörama, kiiremini käima ja ka omamoodi mõtlema.“ Iga tegevuse kirjelduse peale tegi kilpkonn kivipõrandal just seda.

Habemik jälgis samal ajal õpilasi ning päris siis muigega: „Kui te mõistate tingimuslausete põhise aju meisterdamist, siis suudate ka kujud mõtlema panna, aga kuidas anda savile võime täita samu käske uuesti ja uuesti?“

Tea hakkas enda märkmeid uurima nagu kümmekond teist usinamat õpilast, kuid Ra tegi saapaninaga tolmusel põrandal ringe. Ta mõtles samal ajal kaasa, kuid raamatute avamine tundus sel hetkel nagu ületamatu vägitegu. Noormees haigutas ning vaatas maha. Ringid. Kui me kordame midagi lõpmatult, siis see on nõiaring. „Ring,“ pomises ta vaevu kuuldavalt.

Alkeemik vaatas Rale otsa ning kissitas silmi. Ta oli olnud kindel, et ükski õpilastest ei tuleks selle peale nii kiiresti. Habetunud näole ilmus naeratus: „Tore, et keegi raskemat mõttetööd ei karda. Tõepoolest, savist loodud loomade käskude käsikirjad tuleb paigutada ringi. See on maagiliste sõõride ja nõiaringide põhimõte. Luua midagi, mis kestab, kuni ring on murtud. Töö õpetab tegijat! Teil on aega kuni päikseloojanguni.“

Tudengitele anti vabad käed valida meelepärane savikuju, millele „elu“ anda. Alkeemik rõhutas, et maagiline käsikiri tuleb kirjutada pikale paberitükile ning seejärel otsest ringi põhimõttel kinnitada ja loomale lõugade vahele pista. Suurem osa õpilastest keskendus liikumise selgitamisele: kas käsk tuleks anda iga sammu jaoks või iga kümne sammu kohta? Tea ja Ra tulid üsnagi sarnasele ideele. Mõlemad kirjutasid teksti, mille põhjal loomad liiguvad enda omaniku kõrval või kohal ning vajadusel saab neile öelda kui kaugele nad isekeskis mingis suunas astuma või lendama peavad. Ülejäänud päev kulus lisakäskude kirjutamisele. Tea oli õppinud mitmete keeruliste maagiasüsteemide kohta ning üritas enda valitud savikotkale võimalikult palju trikke ette anda. Ra keskendus põhilisele, kuid ta ei suutnud vastu panna paarile kavalale ideele, mille peale teised tõenäoliselt ei mõtleks. Tõepoolest, miks ei võiks suur savihärg aeg-ajalt ninasõõrmetest tuleleeke paisata.

„On aeg!“ kostis alkeemik toolilt. Tudengid söötsid pabersõõrid loomadele ning vaatasid pealt, kuidas iga olend liikuma hakkas. Linnud nokkisid sulgi ja hobused koputasid kapjadega vastu maad ning tundusid hingavat nagu loomad ikka.

Enne, kui alkeemik suutis uut lauset alustada, muutus tõrvikuvalgus siniseks ning katedraali keskelt, siniste leekide lahvatusest ilmus meresinine hõljuv kristall.

„Tundub, et oleme isegi hiljaks jäänud. Mis seal ikka. Astuge ringi Rännukivi ümber!“

Selline ehmatus suutis äratada ka Ra enda väsimusest. Tea neelatas sügavalt ja hoidis enda käsi taskutes, et hoida neid värisemast.

Alkeemik ütles vaid viimased sõnad: „Alkeemia saadab teid maailma ning selle abil jõuate kindlasti ka tagasi. Alkeemia ei ole tegevus, vaid mõtteviis."

Ühel hetkel seisid nad ringis helendava kivi ümber ning teisel olid nad ümbritsetud sinisest tuleleegist. Kõrvulukustavate karjete saatel paisati kõik tudengid õhku ning maad puudutasid nad juba kusagil mujal.

Tea lamas kivisel pinnal ning kuulas, kuidas külm tuul vilistas. Ta avas silmad ning nägi enda kohal pimeda taeva all vaid helendavate silmadega sarvilist varju.

3.7 Kolmanda nädala kontrollülesanded 3.1, 3.2, 3.3

Kolmandal nädalal tuleb esitada nelja kohustusliku ülesande lahendused. Neljanda ülesande puhul on võimalik valida lahendamiseks vähemalt üks järgmistest ülesannetest, kas 3.4a, 3.4b või 3.4c (võib ka kaks või kolm lahendada). Lahendused tuleb esitada Moodle'is, kus need kontrollitakse automaatselt. Moodle'is on ka nädalalõputest 10 küsimusega, millest tuleb vähemalt 9 õigesti vastata.

Eks seda tuleb ette ka edaspidi, et programm teeb põhimõtteliselt nõutud asja, aga väljastab midagi rohkem või vähem või kuidagi teisiti ei vasta täpselt ülesandele. (Näiteks on lahendus hoopis vingem kui ülesandes nõutud.) Sellisel juhul võib automaatkontroll teie lahenduse valeks lugeda. Kui teile tundub, et automaatkontroll töötab ebakorrektselt, siis palun kirjutage aadressil proge@ut.ee.

Kontrollülesanne 3.1. Äratus

Manivaldil on hommikuti raske tõusta ja tal on äratuskell, mis äratab teda teatud arv kordi koos tervitustekstiga.

Koostada programm, mis

- küsib kasutajalt, mitu korda äratus heliseb ning
- väljastab sama arv kordi ekraanile ***Tõuse ja sära!***.

Näited programmi tööst:

```
>>> %Run yl3.1.py

  Sisestage mitu korda äratada: 3
  Tõuse ja sära!
  Tõuse ja sära!
  Tõuse ja sära!

>>>
>>> %Run yl3.1.py

  Sisestage mitu korda äratada: 1
  Tõuse ja sära!

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 3.2. Laikimine

Kaisa tegi Facebooki postituse. Ta märkas, et iga minuti järel laikisid ta sõbrad tema postitust nii, et esimesel minutil kogus postitus 1 laigi, teisel minutil ei laikinud keegi, kolmandal minutil 3 laiki, neljandal minutil mitte ühtegi laiki, viiendal minutil 5 laiki jne.

Koostada programm, mis

- küsib kasutajalt minutite arvu (mittenegatiivne täisarv);
- arvutab while-tsükli abil postituse laikide koguarvu;
- väljastab saadud laikide arvu ekraanile.

Näiteks, kui kasutaja sisestas 7, siis paaritute arvude summa on 16, sest 1 + 3 + 5 + 7 = 16. Kui kasutaja sisestas 8, siis on summaks samuti 16, sest 1 + 3 + 5 + 7 = 16.

Näited programmi tööst:

```
>>> %Run yl3.2.py

  Sisestage minutite arv: 7
  Laikide koguarv on 16.

>>>
>>> %Run yl3.2.py

  Sisestage minutite arv: 8
  Laikide koguarv on 16.

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 3.3. Täringumäng

Erinevate täringumängude jaoks on vajalik erinev arv täringuid. Näiteks *Yahtzee* (*Yatzy*) jaoks on vaja 5 täringut, *Craps*i jaoks aga 2 täringut.

Koostada programm, mis

- küsib kasutajalt vajalike täringute arvu;
- viskab vastava arvu täringuid (genereerib vastava arvu suvalisi arve, mis jäävad 1 ja 6 vahele);
- väljastab iga arvu eraldi reale.

Vihje: kui kasutada tsüklit, mis teeb kasutaja sisestatud arvu samme, siis igal sammul tuleb genereerida üks juhuslik arv ja see väljastada.

Näited programmi tööst:

```
>>> %Run yl3.3.py

  Täringute arv: 5
  1
  4
  2
  1
  5

>>>
>>> %Run yl3.3.py

  Täringute arv: 2
  4
  1

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

3.8 Kolmanda nädala kontrollülesanded 3.4abc

Kolmandal nädalal tuleb esitada nelja kohustusliku ülesande lahendused. Neljanda ülesande puhul on võimalik valida lahendamiseks vähemalt üks järgmistest ülesannetest, kas 3.4a, 3.4b või 3.4c (võib ka kaks või kolm lahendada). Lahendused tuleb esitada Moodle'is, kus need kontrollitakse automaatselt. Moodle'is on ka nädalalõputest 10 küsimusega, millest tuleb vähemalt 9 õigesti vastata.

Eks seda tuleb ette ka edaspidi, et programm teeb põhimõtteliselt nõutud asja, aga väljastab midagi rohkem või vähem või kuidagi teisiti ei vasta täpselt ülesandele. (Näiteks on lahendus hoopis vingem kui ülesandes nõutud.) Sellisel juhul võib automaatkontroll teie lahenduse valeks lugeda. Kui teile tundub, et automaatkontroll töötab ebakorrektselt, siis palun kirjutage aadressil prog@ut.ee.

Järgmisest kolmest ülesandest (3.4a, 3.4b, 3.4c) tuleb lahendada vähemalt üks.

Kontrollülesanne 3.4a Laikimine ver. 2

Kaisa tegi postituse Facebooki. Ta märkas, et iga minuti järel laikisid ta sõbrad tema postitust nii, et esimesel minutil kogus postitus 1 laigi, teisel minutil 3 laiki, kolmandal minutil 5 laiki, neljandal minutil 7 laiki, viiendal minutil 9 laiki jne.

Koostada programm, mis

- küsib kasutajalt minutite arvu (mittenegatiivne täisarv);
- arvutab while-tsükli abil postituse laikide koguarvu;
- väljastab kingitavate laikide koguarvu.

Näiteks, kui kasutaja sisestas 4, siis paaritute arvude summa on 16, sest 1 + 3 + 5 + 7 = 16. Kui kasutaja sisestas 7, siis on summaks 49, sest 1 + 3 + 5 + 7 + 9 + 11 + 13 = 49.

Näited programmi tööst:

```
>>> %Run yl3.4a.py

  Sisestage minutite arv: 4
  Laikide koguarv on 16.

>>>
>>> %Run yl3.4a.py

  Sisestage minutite arv: 7
  Laikide koguarv on 49.

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 3.4b Vabavisked

Korvpalluri vabavisete senist visketabavust saab (teatud mööndustega) kasutada tuleviku visete tõenäosusena.

Koostada programm, mis

- küsib kasutajalt visketabavuse (tabavustõenäosuse) protsentides (täisarv 0 kuni 100);
- simuleerib while-tsükli abil 1000 viset ja igal viskel (arvestades tõenäosust) väljastab, kas see tabas;
  - iga viske kohta peab väljastama ühe rea ja see rida peab sisaldama sõna ***tabas*** või ***mööda***
- arvutab kokku tabanud visete arvu ja see väljastab selle kõige viimasena.

Näide programmi tööst:

```
>>> %Run yl3.4b.py
  Sisestage visketabavuse protsent: 45
  1. vise tabas
  2. vise tabas
  3. vise tabas
  4. vise mööda
  5. vise mööda
...
  995. vise mööda
  996. vise mööda
  997. vise mööda
  998. vise mööda
  999. vise tabas
  1000. vise tabas
  Tabas 458 viset.
>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.

Kontrollülesanne 3.4c Male

Legend räägib, et malemängu leiutajale olla tollane valitseja pakkunud tasu. (Sellest legendist räägib ka Tõnu Tõnso paarikümne aasta taguses leheloos.)

Leiutaja oli “tagasihoidlik” ja palus tasuks

- esimese ruudu eest 1 nisutera,
- teise ruudu eest 2 korda rohkem ehk 2,
- kolmanda ruudu eest veel 2 korda rohkem ehk 4,
- neljanda ruudu eest siis 8,
- viienda ruudu eest 16 jne

Malelaual on 64 ruutu.

Koostada programm, mis

- küsib kasutajalt ühe täisarvu;
- arvutab while-tsükli abil, mitu nisutera sellise järjekorranumbriga ruudu eest leiutaja küsis;
- tulemus väljastatakse ekraanile pärast tsüklit.

Näited programmi tööst:

```
>>> %Run yl3.4c.py

  Sisestage täisarv: 10
  Nisuteri 10. ruudu eest: 512

>>>
>>> %Run yl3.4c.py

  Sisestage täisarv: 24
  Nisuteri 24. ruudu eest: 8388608

>>>
```

Kui olete juba hulk aega proovinud ülesannet iseseisvalt lahendada ja see ikka ei õnnestu, siis võib-olla saate abi murelahendajalt. Püütud on tüüpilisemaid probleemseid kohti selgitada ja anda vihjeid.