5.1 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.

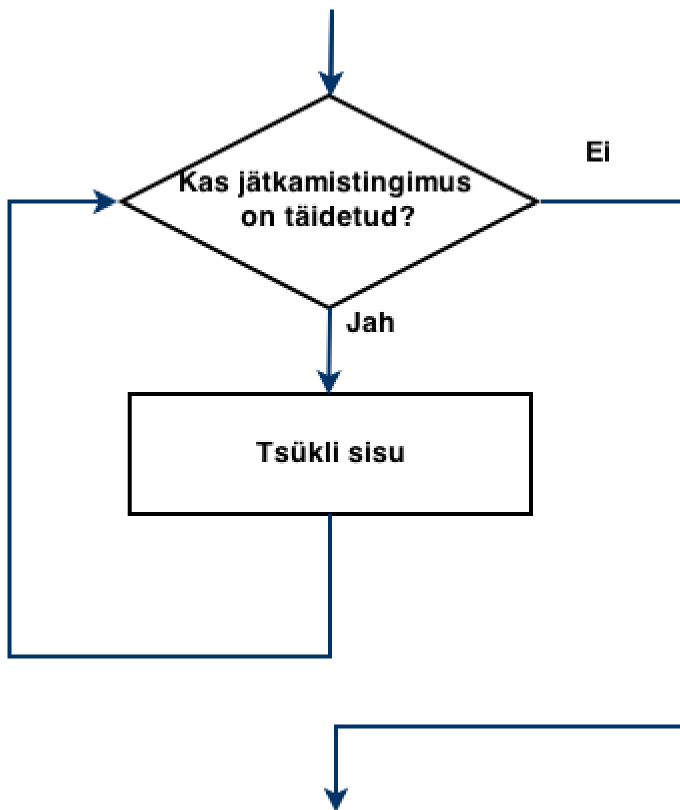
Kui tahta mingeid asju korduvalt teha, siis võivad ju programmid väga pikaks minna? Näiteks kui tahame, et programm väljastaks ekraanile viis korda üksteise alla `Tere!`, siis kõlbaks selline programm.

```
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. Selles näites on 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. Pole ju mõtet parooli uuesti küsida, kui juba õige on sisestatud.

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*-lauses 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. Variandi `while True` puhul on tegemist lõpmatu tsükliga, sest tingimusavaldis on alati väärtusega `True`. 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 programmitöö katkestada Stop-märgiga nupu või klahvikombinatsiooni *Ctrl + F2* abil. (Hoiame all *Ctrl*-klahvi ja vajutame samaaegselt alla *F2*-klahvi, mis on klaviatuuri ülemisel real.) (Keskkonnas IDLE katkestatakse programmitöö hoopis *Ctrl + C* abil.) Tegelikult saab lõpmatut tsüklit kasutada ka päris sihipäraselt sellises olukorras, kus tuleb näiteks midagi 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

Meie aga tahame ikkagi, 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 luba. Kuidas me inimlikult sellises olukorras loendaksime? Üks võimalus oleks näiteks sõrmedel lugeda ja nii meeles hoida, kui palju kordi juba tehtud. Põhimõtteliselt teeme sarnaselt ka programmeerides. Võtame ühe muutuja, mille nimeks saagu *i*. Muide just *i* ongi sageli sellise loendaja nimeks. 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 natuke teine. 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
```

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 2 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.

Lisame programmile 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.

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

KILPKONN TSÜKLIS

Eelmises osas 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 ... Aga proovige ise!

5.2 Veel tsüklist

Eelmistes näidetes oli meil tsükli korduste arv juba algul teada. 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) juhul, kui sisestatud PIN-kood ei ole õige. Jätkamistingimuseks sobiks `sisestatud_pin != "1234"`, sest me peaks uuesti küsima just siis, kui sisestatud kood ei ole õige. 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 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, kas sisestatud PIN-kood on vale, aga lisaks kontrollitakse veel 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 järelejäänud katseid peab olema rohkem kui 0.

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" käsu `sleep(1)` abil, kuid selle kasutamiseks tuleb programmi algusesse lisada rida `from time import sleep`, mis ütleb Pythonile, et järgnevas programmis kasutatakse funktsiooni `sleep()`, mis pärineb moodulist `time`.

```
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)
```

Soovijad võiksid mõtiskleda ja katsetada, 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 *if*-lause *while*-tsükli sees.

Klassikalises arvamismängus see nii ongi.

```
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!")
```

Näites oli kasutatud ka tühje ridu, et inimesel oleks programmi lihtsam lugeda. Python tühje ridu ei arvesta. Samuti ei arvesta Python # järel olevaid kommentaare.

Pange mäng tööle ja mängige, aga ärge sellest sõltuvusse sattuge! Proovige mängu modifitseerida. Näiteks las programm loeb vastamiste arvu ka ja annab reaktsiooni vastavalt sellele. (Kui arvatakse arv esimese korraga, siis näiteks võiks mängijal soovitada oma selgeltnägija võimeid ehk laiemaltki kasutada.) Huvitav oleks ka teistpidi ülesanne, kus arvuti on arvaja osas ja arvab 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 programmeeritav. Lahendus esitada Moodle'is vastava testiküsimuse vastusena.

5.3 Silmaring. 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.

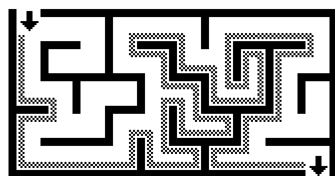
Suuremat huvi pakuvad aga keerdkäigustikust väljapääsemise ülesanded, sest nende lahendused on algoritmilised. Järgmisena tutvustatakse mõnda lahenduseeskirja, mis aitavad eksinud ränduril leida keerulisest labürindist pääsetee.

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 (https://www.brainpop.com/games/blocklymaze/) 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.

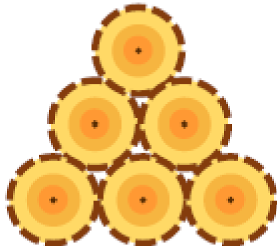
ALLIKAD

1. http://www.labyrinthos.net
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

5.4 Kontrollülesanne V

Palkide arv

Metsast toodud ümarpalgid soovitakse virna laduda nii, et ülemises reas on üks palk ja igas järgmises reas on eelmisest ühe võrra rohkem, nt nii:



Kirjutada programm, kus kasutajalt küsitakse ridade arv ning arvutatakse ja väljastatakse ekraanile virnas olevate palkide arv. Võib eeldada, et kasutaja sisestab positiivse täisarvu.

Näiteks kui sisestatud ridade arv on 5, siis ekraanile kuvatakse 15, sest 1+2+3+4+5 = 15. Kui sisestatud ridade arv on 189, siis ekraanile kuvatakse 17955.

Arvude summa peab arvutama `while`-tsükli abil. Tsükli igal sammul tuleb liita eelmise sammuga võrreldes ühe võrra erinev arv.

Edasijõudnutele: Täiendada programmi nii, et kui kasutaja sisestab mittepositiivse täisarvu, siis väljastada ekraanile 0.

1. vihje.

Ülesande lahendamiseks tuleb läbi mõelda, milles ülesanne seisneb. Kui ülesandest aru ei saa, siis on suhteliselt vähe lootust, et lahendamine välja tuleb. Püüame mõelda, kuidas me lahendaksime selle ülesande ilma arvutita.

Kui me oleme liiga palju matemaatikat õppinud, siis võib tekkida mõte, et tegemist on aritmeetilise jadaga ja sellel on summa valem ja ... See on hea mõte, aga ei kasuta tsüklit.

Püüame nüüd tsüklilise lahendamise peale mõelda. Meil on vaja liita kokku arvud alates 1 kuni teatud arvuni. Olgu selleks arvuks näiteks 5. Muidugi peame programmi lõpuks selliseks saama, et 5 asemel on kasutaja poolt sisestatud arv. Seda summeerimist on võimalik teha nii alates arvust 1 ja lõpetades arvuga 5 või ka alates arvust 5 ja lõpetades arvuga 1.

Kui oleme selle otsuse teinud, siis saame täpsemalt mõelda, mis tsükli igal sammul toimub. Püüame summeerida alates arvust 1 suuremate arvude poole. Kõigepealt on meil siiski vaja muutujat, mille väärtusesse summat "koguda". Olgu selleks muutuja `summa`. Algväärtuseks võtame 0.

```
summa = 0
```

See muutuja tuleb mängu võtta enne tsüklit. Tsükli igal sammul peame sinna siis midagi juurde liitma. Esimesel sammul 1, teisel sammul 2 ja kolmandal 3 jne. Tsükli jätkamistingimuses peame tagama, et samme oleks täpselt parasjagu.

Kuidas saada nii, et see liidetav arv nii muutuks. Võtame selleks ühe muutuja veel, näiteks `i`. Algväärtuseks võtame 1.

```
i = 1
```

Ja siis edasi ...

Näide programmi tööst:

```
>>> %Run yl5.py

  Sisesta ridade arv: 6
  21

>>> 
```

Kontrollülesannete lahendused esitatakse *Moodle*'is.

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.

5.5 Maalähedane lugu V

Nüüd see siis oligi õue peal - punane ja kiiskav. Lembit oli seda vana Jawat korras hoidnud juba aastakümneid. Viimasel ajal oli ta rohkem autoga sõitnud ja mootorratta välja ajanud vaid erilisteks hetkedeks. Nüüd oligi üks selliseid, sest Rasmus oli maal ja tahtis Liisule ka seda ilmaimet näidata. Rasmus oli pereliikmetest kõige rohkem tehnikast huvitatud ja vanaonu selgitas talle väga meelsasti, kuidas üks või teine asi tsikli juures käib ning mis mida ringi ajab. Lembit oli kuldsete kätega mees ja oli igasuguseid masinaid ise ehitanud ja parandanud - teadis neist palju. Lembit oli Rasmuselegi selgitanud, miks seda sõidukit paljudes keeltes just "tsikliks" (motorcycle, мотоцикл, motorcykel, motocicleta) kutsutakse - ikka selle tsüklilisuse pärast, ratas käib ringi.

Igatahes olid Rasmus ja Lembit suured sõbrad. Kui Rasmus veel päris väike oli, siis ükskord oli ta Lembitu pärast päris mures olnud. Nimelt, kui Ada neil linnas külas käis, oli Rasmus pealtkuulnud kurtmist, et Lembit ei saa tsiklist välja. See oli küll imelik asi - kuidas ei saa tsiklist välja. Autos saab sees kinni olla, aga tsiklist nüüd ei saa välja! Vaene väikemees ei teadnud, et Lembitul peale tehnika on veel teinegi "kiindumus".

VAHELEPÕIGE Ada, Rasmus ja Lembit muidugi ei mõelnud sellele, et tsüklitega ja nendest välja saamisega on muret ka programmides. Ühelt poolt on arvuti väga võimas asju tsükliliselt tegema. Teiselt poolt peavad need tsüklid sobivalt algama ja lõppema.

See oli aga ammu ja juba kaksteist aastat polnud Lembit tilkagi võtnud. Nüüd siis seisid nad ümber mootorratta ja arutasid, kes esimese tiiru teeb. Rasmus oli kevadel juhiloa kätte saanud ja kibeles ikka ise kohe sadulasse. Lembit ei olnud kade ja nii ta siis vaatas, kuidas Jawa koos Rasmuse ja Liisuga väravast välja müristas.

Plaan oli Voorvaili mäele sõita. Ümber mäe oli tee, millel sõitmine kedagi ei häirinud, sest lähimad majad olid kaugemal. Ettevaatlik pidi ikkagi olema, sest vahel oli mäel ikkagi inimesi. Rasmus ja Liisu hakkasid ümber mäe tiire tegema. Ringi alguses oli üks natuke märjem koht. Oli kirjutamata reegel,

et tiirutamine tuleb lõpetada, kui tee poriseks läheb. Nii saidki noored seekord kolm tiiru teha ja tulid siis koju tagasi.

PS Kas teate, kuidas Voorvaili mägi oma nime sai?

See oli selline lugu, et mõisahärra olla Inglismaal käinud ja seal mõned ingliskeelsed sõnad ka ära õppinud. Kaks neist oli "for" ja "while". Mõisahärra oli sõbralik mees ja polnud harvad juhud, kui ta taluperemeestega kõrtsis asju arutas. Ükskord läksid nad arutamisega nii liiale, et mõisahärra eksis koju minnes teelt ja teda tuli taga otsima hakata. Otsijad jõudsid pimedas mäe juurde ja kuulsid, kuidas mõisahärra laulis täiest kõrist. Laulus oli ainult kaks sõna - "for" ja "while". Nii hakatigi mäge Voorvaili mäeks kutsuma.