**1.uzd**

import numpy

import math

def is\_natural(n):

# Pārbauda vai simbolu virkne ir naturāls skaitlis vai nav

# Ja ir naturāls skaitlis, tad True. Ja nav tad False.

# n - simbolu virkne, kuru pārbauda.

if str(n).isdigit() and float(n) == int(n) and int(n) > 0:

return True

else:

return False

def izveidot\_masivu\_ar\_garumu(n):

# Izveido masīvu ar noradīto garumu n

# n - naturāls skaitlis

a = numpy.arange(n)

for i in range(n):

b = input("Ievadiet " + str(i) + ".elementu ===> ")

b = is\_natural\_i(b, i)

a[i] = b

return a

def is\_natural\_i(x, i):

while True:

try:

x = int(x)

1/x

math.sqrt(x)

except:

x = input("Kļūda! Ievadiet " + str(i) + ".elementu ===> ")

else:

return int(x)

def izvade(x):

# Izvada masīva elementus pēc kārtas līdz pēdējam

# x - viendimensijas masīvs

n = len(x)

s = str(x[0])

for i in range(1, n):

s = s + ", " + str(x[i])

print(s)

def mode(masivs):

# Atrod masīva modu un atgriež masīvu ar modam (vai modu)

# Ja masīva visi elementi ir vienādi, tad atgriež False (moda netika atrasta)

# masivs - viendimensijas masīvs (kurš vel netika kārtots augoša secība)

'''

Paskaidrojums, kā šī funkcija darbojas:

Funkcija vispirms paņem, ka moda ir pirmais sakārtotā masīva elements.

Pēc tam, funkcija iziet cauri katram masīva elementam un atjaunina sarakstu ar modam un to biežumu katru reizi, kad tiek atrasta jauna mode ar augstāku biežumu (elements, kuru ir vairāk).

Ja tas saskaras ar modu ar tādu pašu biežumu (biežums - skaits, cik reizes paradās šis elements) kā pašreizēja moda, tad tas pievieno jauno modu, modu sarakstam.

Funkcija iterē katru elementu sakārtotajā masīvā un pārbauda, vai tas ir vienāds ar iepriekšējo elementu.

Ja tā ir, mainīgais "sk" tiek palielināts. Ja tā nav, funkcija pārbauda, vai pašreizējais skaits ir lielāks par līdz šim saglabāto maksimālo skaitu.

Ja ir lielāks, tad maksimālais skaits tiek atjaunināts un pašreizējais skaitlis (elements) tiek pievienots modas sarakstam.

Ja pašreizējais skaits ir vienāds ar maksimālo līdz šim saglabāto skaitu, modas sarakstam tiek pievienots arī šis skaitlis (elements) (vairākas modes).

Pēc tam kad tika "iziterēts" viss masīvs, tad pārbaudām, vai pēdējā elementa biežums ir lielāks par līdz šim redzēto maksimālo biežumu.

Ja tā ir, tad maksimālais skaits tiek atjaunināts un modas saraksts ir tas pēdējais elements.

Ja pēdējā elementa skaits ir vienāds ar maksimālo līdz šim redzēto (saglabāto) skaitu, modas sarakstam tiek pievienots arī pēdējais elements (vairākas modes).

Ja maksimālais biežums ir vienāds ar 1 (t.i., visi masīva elementi ir unikāli), funkcija atgriež vērtību False.

Pretējā gadījumā tas atgriež atrasto modas sarakstu.

'''

sort\_atrais\_augosa(masivs, 0, len(masivs) - 1) # Ievadītais masīvs tiek izkārtots augošā secībā, izmantojot sort\_atrais\_augosa

sk = 1 # seko pašreizējā apstrādājamā skaitļa (elementa) biežumam

max\_sk = 1 # saglabā jebkura masīva skaitļa (elementa) maksimālo biežumu

modas\_list = [masivs[0]] # līdz šim atrasto modu saraksts

if len(masivs) == 1: # Ja masīva garums ir 1, tad moda arī ir tas skaitlis (vienīgais elements)

modas\_list = [masivs[0]]

return modas\_list

for i in range(1, len(masivs)):

if masivs[i] == masivs[i - 1]: # Ja elements i ir vienāds ir i - 1, tad palielinām skaitītāju

sk = sk + 1

else:

if sk > max\_sk: # ja skaititajs kļūst lielāks nekā maksimālais skaitītājs

max\_sk = sk # tad maksimālais skaititajs atjaunojas

modas\_list = [masivs[i - 1]] # modas saraksts atjaunojas un paliek tas masīva elements, kurš vislielāko reizi paradījās pēc skaitītajā

elif sk == max\_sk: # Ja skaititajs ir vienāds ar pašreizējo maksimālo skaitītāju (max biežumu)

modas\_list.append(masivs[i - 1]) # tad pievienojam sarakstam jauno modu (papildus modu)

sk = 1 # pēc iterācijām sk atkal kļūst par 1

if sk > max\_sk: # Tas ir veidots pēdējam elementam. Ja lielāks skaititajs nekā pašreizējais maksimālais.

max\_sk = sk # tad atjaunojam maksimālo skaitītāju

modas\_list = [masivs[-1]] # atjaunojam modas sarakstu

elif sk == max\_sk: # Ja modas biežumi ir vienādi, tad pievienojam modas sarakstam

modas\_list.append(masivs[-1])

if max\_sk == 1: # Ja maksimālais biežums ir vienāds ar 1 (visi masīva elementi ir unikāli), atgriež vērtību False

return False

else:

return modas\_list # Atgriež atrasto modas sarakstu

def sort\_atrais\_augosa(a, sv, bv):

# Sakārto masīvu augoša secība un atgriež salīdzināšanas skaitu, lai sakārtotu masīvu

# Kārtošanas tiek izmantota Hoara (ātrais) metode (quicksort)

# a - viendimensijas masīvs

# sv - sākuma vērtība

# bv - beigu vērtība

if sv < bv:

i = sv

j = bv

solis = -1

lv = True

while i != j:

if lv == (a[i] > a[j]):

x = a[i]

a[i] = a[j]

a[j] = x

x = i

i = j

j = x

lv = not lv

solis = -solis

j = j + solis

sort\_atrais\_augosa(a, sv, i - 1)

sort\_atrais\_augosa(a, i + 1, bv)

# ---------------------------------------------------------

# Galvenā programmas daļa

# ---------------------------------------------------------

m = input("Ievadiet masīva izmēru N ===> ")

while is\_natural(m) == False:

m = input("Masīva izmērs ir naturāls skaitlis!\nIevadiet masīva izmēru N ===> ")

m = int(m)

print("Ievadiet masīva skaitļus!")

t = izveidot\_masivu\_ar\_garumu(m)

print("\nIevadīta skaitļu kopa:")

izvade(t)

sort\_atrais\_augosa(t, 0, len(t) - 1)

print("\nSakārtota skaitļu kopa:")

izvade(t)

modas = mode(t)

if modas == False:

print("\nModa netika atrasta. NAV.")

else:

print("\nSkaitļu kopas moda(s) ir:")

izvade(modas)

**2.uzd**

import numpy

def is\_cipars(n):

# Pārbauda vai simbolu virkne ir vesels skaitlis vai nav

# Ja ir vesels skaitlis, tad True. Ja nav tad False.

# n - simbolu virkne, kuru pārbauda.

try:

n = int(n)

except:

return False

else:

if n >= 10:

return False

if n <= 0:

return False

return True

def is\_cipars\_100(n):

# Pārbauda vai simbolu virkne ir vesels skaitlis vai nav

# Ja ir vesels skaitlis, tad True. Ja nav tad False.

# n - simbolu virkne, kuru pārbauda.

try:

n = int(n)

except:

return False

else:

if n >= 101:

return False

if n <= 0:

return False

return True

def izvade\_matrica\_int(a):

# Atgriež divdimensiju masīvu (matricu), tabulas veidā, str formāta, kur katra rinda ir atdalīta ar jauno rindkopu

# a - divdimensijas masīvs (matrica)

n = a.shape[0] # "x axis" masīva izmēri

m = a.shape[1] # "y axis" masīva izmēri

s = ""

for i in range(n):

for j in range(m):

s = s + "{:8.0f}".format(int(a[i, j]))

s = s + "\n"

return s

def ievade\_matrica(n, m):

# Lietotājs var ievādīt nXm matricas elementus un funkcija atgriež divdimensijas masīvu ar n rindam un m kolonnam ar ievadītām vērtībam

# n - naturāls skaitlis, kurš nosaka matricas rindas skaitu

# m - naturāls skaitlis, kurš nosaka matricas kolonnas skaitu

a = numpy.empty((n, m), dtype=numpy.int64)

for i in range(n):

for j in range(m):

temp = input("Ievadiet matricas elememtu a(" + str(i) + "," + str(j) + ") ===> ")

while is\_cipars(temp) == False:

temp = input("Kļūda! Ievadītais elements nav viencipara naturāls skaitlis!\nIevadiet matricas elememtu a(" + str(i) + "," + str(j) + ") ===> ")

a[i, j] = int(temp)

return a

def matrix\_row\_sum(a):

# rindas summa

b = a.shape[0]

c = numpy.zeros(b)

for i in range(b):

summa = 0

for j in range(b):

summa = summa + a[i, j]

c[i] = summa

return c

def matrix\_col\_sum(a):

# kolonnu summa

b = a.shape[0]

c = numpy.zeros(n)

for i in range(b):

summa = 0

for j in range(b):

summa = summa + a[j, i]

c[i] = summa

return c

def matrix\_diag\_sum(a):

# diagonalu summa

b = a.shape[0]

c = numpy.zeros(n)

summa = 0

for i in range(b):

summa = summa + a[i, i]

c[0] = summa

summa = 0

for i in range(b):

summa = summa + a[i, n - 1 - i]

c[1] = summa

return c

def new\_matrix(a, c, e, d):

n = a.shape[0]

new\_a = numpy.zeros((n + 2, n + 2))

new\_a[1:n + 1, 1:n + 1] = a # centrs

new\_a[1:n + 1, 0] = d

new\_a[0, 1:n + 1] = e

new\_a[1:n + 1, -1] = d

new\_a[-1, 1:n + 1] = e

new\_a[n + 1, n + 1] = c[0]

new\_a[n + 1, 0] = c[1]

new\_a[0, n + 1] = c[1]

new\_a[0, 0] = c[0]

return new\_a

# galvenā programma

n = input("Ievadiet kvadrātitskas matricas izmēru ==> ")

while is\_cipars\_100(n) == False:

n = input("Kļūda! Ievadiet kvadrātitskas matricas izmēru ==> ")

n = int(n)

a = ievade\_matrica(n, n)

b = izvade\_matrica\_int(a)

print("Ievadītā matrica:")

print(b)

# print(b)

# izsauc summu funkcijas

c = matrix\_diag\_sum(a)

d = matrix\_row\_sum(a)

e = matrix\_col\_sum(a)

new\_a = new\_matrix(a, c, e, d)

new\_b = izvade\_matrica\_int(new\_a)

print("Jauna matrica:")

print(new\_b)

**3.uzd**

import numpy

def is\_whole(n):

# Pārbauda vai simbolu virkne ir vesels skaitlis vai nav

# Ja ir vesels skaitlis, tad True. Ja nav tad False.

# n - simbolu virkne, kuru pārbauda.

try:

n = int(n)

except:

return False

else:

return True

def ievade\_matrica\_sim(n, m):

# Lietotājs var ievādīt nXm matricas elementus un funkcija atgriež divdimensijas masīvu ar n rindam un m kolonnam ar ievadītām vērtībam

# n - naturāls skaitlis, kurš nosaka matricas rindas skaitu

# m - naturāls skaitlis, kurš nosaka matricas kolonnas skaitu

a = numpy.empty((n, m))

for i in range(n):

for j in range(m):

if i == j or i < j:

temp = input("Ievadiet matricas elememtu a(" + str(i) + "," + str(j) + ") ===> ")

while is\_whole(temp) == False:

temp = input("Kļūda! Ievadītais elements nav vesels skaitlis!\nIevadiet matricas elememtu a(" + str(i) + "," + str(j) + ") ===> ")

a[i, j] = int(temp)

return a

def izvade\_matrica\_int(a):

# Atgriež divdimensiju masīvu (matricu), tabulas veidā, str formāta, kur katra rinda ir atdalīta ar jauno rindkopu

# a - divdimensijas masīvs (matrica)

n = a.shape[0] # "x axis" masīva izmēri

m = a.shape[1] # "y axis" masīva izmēri

s = ""

for i in range(n):

for j in range(m):

s = s + "{:8.0f}".format(int(a[i, j]))

s = s + "\n"

return s

def make\_symmetrical(a):

n = a.shape[0]

m = a.shape[1]

b = numpy.empty((m, n))

for i in range(m):

for j in range(n):

if i > j:

b[i, j] = a[j, i]

elif i < j or i == j:

b[i, j] = a[i, j]

return b

n = int(input("Ievadiet matricas izmēru ==> "))

#m = int(input("Ievadiet matricas izmēru ==> "))

a = ievade\_matrica\_sim(n, n)

print("Jānīša matrica pirms atjaunošanas")

print(izvade\_matrica\_int(a))

print("Jānīša matrica pēc atjaunošanas")

sim\_a = make\_symmetrical(a)

print(izvade\_matrica\_int(sim\_a))

**4.uzd**

import random

def start\_izloze():

# pirmo reiz piecu lodīšu izlozi

b = set()

for i in range(5):

a = random.randint(1, 49)

while a in b:

a = random.randint(1, 49)

b.add(a)

return b

def again\_izloze(b):

# atkārtots vienas lodītes lozēšana un pievieno to izlozēto lodīšu kopai

a = random.randint(1, 49)

while a in b:

a = random.randint(1, 49)

b.add(a)

return b

lod\_janis = start\_izloze() # Janisa lodites

lod\_peter = start\_izloze() # pēterīša lodites

laimests = 1024

meginajumi = 1

kopigas\_lodes = lod\_janis.intersection(lod\_peter) # kopigas

# ja uzreiz visi 7 ir vienadi tad 1024

if len(kopigas\_lodes) == 7:

laimests = laimests

meginajumi = meginajumi

else:

# Ja kopigas kopas garums nav 7, tad lozesim atkal

while len(kopigas\_lodes) < 7:

again\_izloze(lod\_janis)

again\_izloze(lod\_peter)

kopigas\_lodes = lod\_janis.intersection(lod\_peter)

else:

i = len(lod\_janis) - 7

meginajumi = i + meginajumi

laimests = laimests / 2\*\*i # dalam ar kapinataju

if laimests < 1: #meginajumi > 10 or laimests :

laimests = 0

print("Spēle tiek pārtraukta! Tika izdarīti vairāk nekā 10 mēģinājumi!\n")

print(meginajumi, "mēģinājumi tika izdarīti.")

print("Jānīša visas izvilktas lodītes:")

print(lod\_janis)

print("Pēterīša visas izvilktas lodītes:")

print(lod\_peter)

print("Kopīgās lodītes:")

print(kopigas\_lodes)

print("Laimests:")

print(laimests, "EUR")

**5.uzd**

import copy

import random

class Kopa:

# Klase kopa.

def \_\_init\_\_(self):

# Inicializēsim tukšo kopu.

# self - kopa (objekts Kopa()).

self.\_\_kopa = []

def saturs(self):

# Atgriež kopas saturu.

# self - kopa (objekts Kopa()).

return self.\_\_kopa

def izdrukat(self):

# Izdrukā glīta veidā kopas saturu.

# Tukša kopa tiek attēlota ar Ø simbolu.

# Piemēram, izdrūka { 1 2 3 }.

# self - kopa (objekts Kopa()).

sk = len(self.\_\_kopa)

if sk == 0:

sv = "Ø" # Tukša kopa ∅

else:

sv = "{ "

for i in self.\_\_kopa:

sv = sv + str(i) + " "

sv = sv + "}"

print(sv)

def pievienot(self, elements):

# Ļauj pievienot vai nu vienu elementu kopai, vai nu elementus kā list (sarakstu), vai tuple (kortežu).

# Piemēram, a.pievienot([1, 3, 4]), vai b.pievienot((1, 2, 3)), vai c.pievienot(2).

# self - kopa (objekts Kopa()).

# elements - elements vai elementi, kurus gribam pievienot kopai.

paz = True

#print(i[0])

for i in self.\_\_kopa:

#print(i[0])

#print(elements)

if i[0] == elements[0]:

paz = False

#if i == elements:

# paz = False

if paz:

self.\_\_kopa.append(elements)

def izmest(self, elements):

# Nodzēs norādītu vienu elementu (elements) no kopas.

# Ja tāda elementa nav, tad neko nenodzēs.

# Līdzīgs set discard.

# self - kopa (objekts Kopa()).

# elements - elements vai elementi, kurus gribam pievienot kopai.

for i in self.\_\_kopa:

if i == elements:

self.\_\_kopa.remove(elements)

def nonemt(self, elements):

# Nodzēs norādītu vienu elementu (elements) no kopas.

# Ja tāda elementa nav, tad raise ValueError, ka tāds elements nav kopā.

# Līdzīgs set remove.

# self - kopa (objekts Kopa()).

# elements - elements vai elementi, kurus gribam pievienot kopai.

if elements not in self.\_\_kopa:

raise ValueError(f"{elements} nav kopā. Nevar noņemt elementu, kuru nav. Izmantojiet 'izmest'.") # is not in the kopa

for i in self.\_\_kopa:

if i == elements:

self.\_\_kopa.remove(elements)

def vai\_pieder(self, elements):

# Pārbauda vai kopai piedēr norādītais elements.

# Atgriež True, ja norādītais elements piedēr kopai.

# Atgriež False, ja norādītais elements nepiedēr kopai.

# self - kopa (objekts Kopa()).

# elements - elements vai elementi, kurus gribam pievienot kopai.

for i in self.\_\_kopa:

if i == elements:

return True

return False

def izdzest(self):

# Pārverš kopu par tukšo kopu un atgriež to.

# self - kopa (objekts Kopa()).

self.\_\_kopa = []

return self.\_\_kopa

def atjaunot(self, elements):

# Vai pārmainīt visas vērtības kopai.

# Piemēram, a.update([4, 5, 6, 7]).

# Atjauno kopu līdzīgi, ka set update.

# self - kopa (objekts Kopa()).

# elements - elements vai elementi, kurus gribam pievienot kopai.

self.\_\_kopa = []

if type(elements) == list:

for el in elements:

if el not in self.\_\_kopa:

self.\_\_kopa.append(el)

elif type(elements) == tuple:

for el in elements:

if el not in self.\_\_kopa:

self.\_\_kopa.append(el)

else:

paz = True

for i in self.\_\_kopa:

if i == elements:

paz = False

if paz:

self.\_\_kopa.append(elements)

def pop(self):

# Atgriež vienu nejaušu elementu no kopas.

# Paņem nejaušu kopas elementu līdzīgi, ka set pop.

# Ja mēģināsim izņemt nejaušu elementu no tukšas kopas, tad print("Kļūda! Nevar izņemt elementu no tukšas kopas.").

# self - kopa (objekts Kopa()).

try:

a = random.choice(self.\_\_kopa)

except:

print("Kļūda! Nevar izņemt elementu no tukšas kopas.") # Raise Exception

else:

self.\_\_kopa.remove(a)

return a

@staticmethod

def apvienot(a, b):

# Apvieno divas kopas a un b.

# A U B

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

c = copy.deepcopy(a)

x = a.saturs()

y = b.saturs()

for i in y:

if i not in x:

c.pievienot(i)

return c

# Var izmantot arī šo metodi, "apvienot1" nevis "apvienot".

# Metode "apvienot1" neizmanto deepcopy.

'''

def apvienot1(a, b): # tas pats, bet bez deepcopy

c = Kopa()

x = a.saturs()

y = b.saturs()

for z in x:

c.pievienot(z)

for z in y:

if z not in x:

c.pievienot(z)

return c

'''

@staticmethod

def vai\_parklajas(a, b):

# Pārbauda vai divas kopas a un b pārklājas.

# Atgriež True, ja kopas pārklājas (ir vismaz viens kopīgs elements).

# Atgriež False, ja kopas nepārklājas (nav nevienā kopīga elementa).

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

c = Kopa()

x = a.saturs()

y = b.saturs()

tuksa\_kopa = Kopa()

d = c.skelums(a, b)

if Kopa.vai\_vienadas(d, tuksa\_kopa):

return False

return True

@staticmethod

def vai\_neparklajas(a, b):

# Pārbauda vai divas kopas a un b nepārklājas (tas pats kā vai\_parklajas, bet tikai negācija).

# Atgriež False, ja kopas pārklājas (ir vismaz viens kopīgs elements).

# Atgriež True, ja kopas nepārklājas (nav nevienā kopīga elementa).

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

c = Kopa()

x = a.saturs()

y = b.saturs()

tuksa\_kopa = Kopa()

d = c.skelums(a, b)

if Kopa.vai\_vienadas(d, tuksa\_kopa):

return True

return False

@staticmethod

def skelums(a, b):

# Atgriež kopas a un b šķēlumu (A ∩ B).

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

c = Kopa()

x = a.saturs()

y = b.saturs()

for i in x:

if i in y:

c.pievienot(i)

return c

@staticmethod

def simetriska\_starpiba(a, b):

# Atgriež kopas a un b simetrisku starpību (A △ B).

# Pēc formulas: A △ B = (A \ B) U (B \ A).

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

c = Kopa()

d = Kopa.starpiba(a, b)

f = Kopa.starpiba(b, a)

c = Kopa.apvienot(d, f)

return c

@staticmethod

def starpiba(a, b):

# Atgriež kopas a un b starpību (A \ B).

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

c = Kopa()

x = a.saturs()

y = b.saturs()

for i in x:

if i not in y:

c.pievienot(i)

return c

@staticmethod

def vai\_vienadas(a, b):

# Pārbauda vai divas kopas a un b ir vienādas (A == B).

# Atgriež True, ja divas kopas ir vienādas.

# Atgriež False, ja divas kopas nav vienādas.

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

x = a.saturs()

y = b.saturs()

for i in x:

if i not in y:

return False

for i in y:

if i not in x:

return False

return True

@staticmethod

def vai\_nav\_vienadas(a, b):

# Pārbauda vai divas kopas a un b nav vienādas (A != B).

# Atgriež True, ja divas kopas nav vienādas.

# Atgriež False, ja divas kopas ir vienādas.

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

x = a.saturs()

y = b.saturs()

for i in x:

if i not in y:

return True

for i in x:

if i not in y:

return True

return False

@staticmethod

def vai\_apakskopa(a, b):

# Pārbauda vai a ir apakškopa b. A ⊆ B.

# Atgriež True, ja a ir apakškopa b.

# Atgriež False, ja a nav apakškopa b.

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

x = a.saturs()

y = b.saturs()

for i in x:

if i not in y:

return False

return True

@staticmethod

def vai\_stingra\_apakskopa(a, b):

# Pārbauda vai a ir stingra apakškopa b. A ⊂ B. (ja A un B ir vienādas kopas, tad A nav stingra apakškopa B).

# Atgriež True, ja a ir stingra apakškopa b.

# Atgriež False, ja a nav stingra apakškopa b.

# a - 1.kopa (objekts Kopa()).

# b - 2.kopa (objekts Kopa()).

x = a.saturs()

y = b.saturs()

for i in x:

if i not in y:

return False

if Kopa.vai\_vienadas(a, b):

print(Kopa.vai\_vienadas(a, b))

return False

else:

return True

def check\_vards(vards):

alfabets = ["A", "Ā", "B", "C", "Č", "D", "E", "Ē", "F", "G", "Ģ", "H", "I", "Ī", "J", "K", "Ķ", "L", "Ļ", "M", "N", "Ņ", "O", "P", "R", "S", "Š", "T", "U", "Ū", "V", "Z", "Ž"]

for i in range(len(vards)):

if vards == "X":

return "X"

if vards[i] not in alfabets:

return False

else:

pass

return True

#vardi = ""

vardi = Kopa()

def input\_latvian\_word():

global vardi

x = input("Ievadiet latviešu vārdu ar lieliem burtiem ==> ")

paz = True

while paz:

if check\_vards(x) == "X":

vardi.izdrukat()

paz = False

if check\_vards(x) == False:

x = input("KĻŪDA! Ievadiet latviešu vārdu ar lieliem burtiem ==> ")

#print(check\_vards(x))

elif check\_vards(x) == True:

vardi.pievienot(x)

return x

'''

kopa1 = Kopa()

kopa1.pievienot("LABI")

kopa1.pievienot("OKAY")

kopa1.pievienot("LAKI")

kopa1.saturs()

kopa1.izdrukat()

'''

a = input\_latvian\_word()

while a != "X":

a = input\_latvian\_word()

vardi.izdrukat()

**6.uzd**

datne1 = open("C:\\Users\\37128\\Desktop\\OGAS.txt", mode="r", encoding="utf-8")

datne2 = open("C:\\Users\\37128\\Desktop\\POGAS.txt", mode="w", encoding="utf-8")

simbols = datne1.read(1)

while simbols != "":

space\_count = 0

while simbols == " ":

space\_count = space\_count + 1

simbols = datne1.read(1)

if space\_count > 0:

simbols = " " + simbols

datne2.write(simbols)

simbols = datne1.read(1)

datne1.close()

datne2.close()