

Dmitrović · Grabusin · Bujanović · Miletić · Kager

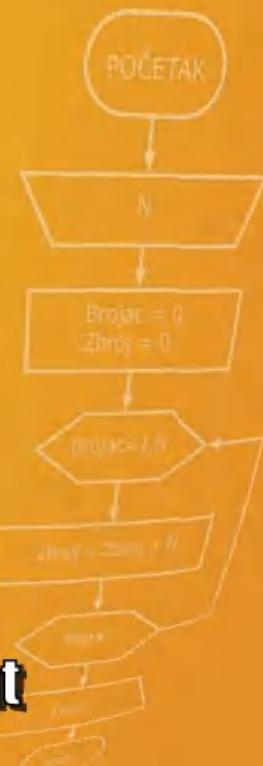
Informatika 3

Udžbenik iz informatike za 3. razred
prirodoslovno-matematičkih gimnazija



```
>>> lista = ['abcd', 3.14, 'Python']
>>> lista.append(100)
>>> lista
['abcd', 3.14, 'Python', 100]
>>> lista.insert(3, 'NOVI')
['abcd', 3.14, 'Python', 'NOVI']
>>> lista.count(123)
0
>>> lista.index('Python')
1
>>> lista.pop(2)
3.14
>>> lista.remove(123)
>>> lista
['abcd', 'NOVI', 'MIOC', 'Python']
>>> lista.reverse()
['abcd', 'NOVI', 'MIOC', 'Python']
>>> lista.sort()
[3.14, 'Python', 'NOVI', 'abcd']
```

Udžbenik • SysPrint



N. Dmitrović, S. Grabusin, Z. Bujanović, Lj. Miletić, D. Kager

Informatika 3

Udžbenik iz informatike za 3. razred
prirodoslovno-matematičkih gimnazija

1. izdanje

Zagreb, 2020.

Informatika 3

udžbenik za 3. razred prirodoslovno-matematičkih gimnazija

Autori:

Nikola Dmitrović, Sanja Grabusin, Zvonimir Bujanović, Ljiljana Miletić, Dalia Kager

Urednik:

Vinkoslav Galešev

Recenzija:

Ante Đerek, Emina Grmić

Lektura:

Matea Srebačić

Priprema:

Tomislav Stanojević

Naslovnica:

Shutterstock

Tisk:

Gradska tiskara Osijek

Nakladnik:

Udžbenik.hr d.o.o.

Ivanečka 31, 10000 Zagreb, Hrvatska

tel: (01) 655 8740, fax: (01) 655 8741

e-mail: info@udzbenik.hr, web: e.udzbenik.hr

Za nakladnika:

Robert Šipek

CIP zapis je dostupan u računalnome katalogu Nacionalne i sveučilišne knjižnice u Zagrebu pod brojem

© **Udžbenik.hr d.o.o.**, Zagreb, 2020.

Nijedan dio ove knjige i pratećih sadržaja ne smije se umnožavati, fotokopirati niti bilo na koji način reproducirati bez nakladnikova pisanih dopuštenja.

SADRŽAJ

1. Stringovi.....	6
1.1. Ponavljanje.....	6
1.2. Metode za rad sa stringovima	8
Od problema do rješenja.....	20
2. Lista listā.....	30
2.1. Uvod	30
2.2. Matrice u matematici	36
Od problema do rješenja	38
3. Rekurzije	58
3.1. Uvod	58
3.2. Rekurzije u zadacima	67
3.3. Pisanje rekurzivnih funkcija	75
4. Datoteke u programskom jeziku Python.....	94
4.1. Osnovne metode za rad s datotekama	94
4.2. Primjeri u interaktivnom okruženju za open(), read() i readline()	95
4.3. Primjeri u interaktivnom okruženju za open(), write() i writelines()	96
5. Korisnička grafička sučelja.....	106
5.1. Modul <i>tkinter</i>	106
5.2. Grafički elementi (<i>Widget</i>)	107
5.3. Upravitelji rasporeda (<i>Layout managers</i>)	113
5.4. Poruke	115
Od problema do rješenja	116
6. Koordinatna grafika.....	122
6.1. Kornjačina grafika	122
6.2. Boje u Pythonu	138
6.3. Grafički prikaz matematičkih funkcija.....	144
Od problema do rješenja	151
7. Složenost algoritama	156
7.1. Što je složenost algoritama?	156
7.2. Algoritmi za sortiranje	163
Od problema do rješenja	169

8. Zapis	178
8.1. Zapis – osnovna ideja.....	178
8.2. Zapis u Pythonu.....	178
Od problema do rješenja.....	182
9. Multimedijijski projekt.....	186
9.1 Od „Camera obscura“ do Instagrama	186
9.2 Projektni zadatak	186
9.3 Obrada slike	187
9.4 Obrada videa	191
9.5 Obrada zvuka	195
10. Baze podataka.....	200
10.1 Osnove baze podataka	201
10.2 Relacijske baze podataka.....	202
10.3 Baze podataka i Python.....	203
10.4. Baze koje život znače	211
11. Izrada mrežnih stranica	214
11.1. HTML jezik	215
11.2. Osnovne mogućnosti HTML jezika	217
11.3. Objavljivanje mrežnih stranica i online grafički uređivači	225
11.4. Priprema multimedijijskih sadržaja za prikazivanje na mreži	226
12. Kriptografija i Tkinter	228
12.1. Teorija prije problema	228
12.2. Supstitucijske šifre	229
12.3. Vigenèrova šifra.....	230
12.4. Playfairova šifra.....	231
Od problema do rješenja.....	233
13. A sad, nešto sasvim drugačije	234
13.1. Varijable u Pythonu	234
13.2. Matrice na Pythonov način.....	240
13.3. Rekurzije – detalji i zanimljivosti	246
Ishodi i međupredmetne teme.....	250
Kazalo	252

Predgovor

Udžbenik je namijenjen nastavi informatike u 3. razredu prirodoslovno-matematičkih gimnazija, ali i drugim vrstama škola, odnosno obrazovnim programima koji u svojim nastavnim planovima djelomice obrađuju predmetni sadržaj.

Osim mnoštva primjera i zadataka, gradivo je oprimjereno algoritamskim primjerima i zadacima koji vas vode od početnoga problema i njegove analize do završnoga programske rješenja. Taj dio gradiva prepoznat ćete po tome što je tiskan na inverznoj podlozi.

Na kraju udžbenika naići ćete na sadržaju i stilom ponešto drugačiji tekst. Napisao ga je kolega **dr. sc. Vedran Čačić** i predstavlja svojevrsno poniranje u tajne mističnog svijeta *Python* koje na drugačiji način osvjetljava spoznaje iz redovnog gradiva. Čitajte ga s užitkom jer ga je s tom namjerom autor i napisao.

Sastavni dio udžbenika je **mrežna stranica** uz udžbenik na adresi e.udzbenik.hr/g3. Na njoj se nalaze digitalni materijali potrebni za izvođenje nastave, kao i neki dodatni sadržaji.

Prateći dio udžbenika je **on-line digitalna zbirka zadataka** koja sadrži zadatke razvrstane po poglavljima, a opremljena je **evaluatorom** za automatsko vrednovanje vaših programskih rješenja. Digitalna zbirka zadataka nalazi se na web-adresi: e.udzbenik.hr/eva.

Za dodatna pitanja pišite nam na e-mail adresu edu@udzbenik.hr ili posjetite naše mrežne stranice e.udzbenik.hr..

Što je što

U tekstu ćete naići na karakteristične grafičke oznake. U tablici su navedena njihova značenja.

oblik	značenje	označavanje programskog koda
	posebno istaknut dio gradiva	
	dodatne informacije za radoznaće	
Teorijski zadaci	ponavljanje i utvrđivanje gradiva	Da biste razlikovali programski kod od obična teksta, označili smo ga plavom trakom slijeva i ispisali ga drukčijim pismom:
Od problema do rješenja	obrada i primjena algoritama (inverzna podloga)	<pre>a=input('Prvi pribrojnik: ') b=input('Drugi pribrojnik: ') zbroj=int(a)+int(b) print('Zbroj dvaju upisanih pribrojnika jest ',zbroj)</pre>
Algoritamski zadaci	primjena algoritama i traženje novih rješenja problema	Peti redak pomaknut ulijevo nije tiskarska greška! Predstavlja nastavak kôda iz četvrtoga retka koji zbog duljine nije stao u jedan redak.

1. Stringovi

1.1. Niz znakova

Podsjetimo se, string je tip podataka koji omogućuje rad s nizom znakova koji predstavljaju riječ ili rečenicu, a definira se nizom znakova unutar jednostrukih ili dvostrukih navodnika.

Svakom znaku u stringu možemo pristupiti navođenjem njegova indeksa (pri čemu treba imati na umu da indeksi kreću od nule), dok podstringu možemo pristupiti navođenjem indeksa prvog znaka podstringa i indeksa do kojeg uzimamo znakove razdvajajući ta dva indeksa dvotočjem, a sve unutar uglatih zagrada. Indeksi smiju biti i negativni.

Primjer 1.

```
a = "Programski jezik Python"  
a[3] = "g" a[-3] = "h"
```

Podsjetnik:
Python ne omogućuje
izravno mijenjanje stringa
na razini znakova.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
P	r	o	g	r	a	m	s	k	i	j	e	z	i	k	P	y	t	h	o	n		
-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Operatori koje možemo upotrebljavati na ovom tipu jesu operator konkatenacije + (zbrajanje) i operator repliciranja * (množenje) te in i not in, kojima ispitujemo je li neki znak u stringu ili nije.

Primjer 2.

```
>>> a,b,c = "Programski", 'jezik', "Python"  
>>> a,b,c  
('Programski', 'jezik', 'Python')
```

+ operator
konkatenacije

* operator
repliciranja

Primjer 3.

```
>>> a,b,c = "Programski", 'jezik', "Python"  
>>> a + b  
'Programskijezik'  
>>> a + " " + b  
'Programski jezik'  
>>> (c + " ") * 5  
'Python Python Python Python Python '
```

1. Stringovi

Primjer 4.

```
>>> a,b,c = "Programski",'jezik',"Python"  
>>> a[4]  
'r'  
>>> b[-3]  
'z'  
>>> c[1:4]  
'yth'  
>>> a[-5:-3]  
'am'
```

Primjer 5.

```
>>> a,b,c = "Programski",'jezik',"Python"  
>>> 'ram' in a  
True  
>>> 'ram' not in a  
False
```

Funkcije definirane na tipu string jesu:

funkcija	opis
len()	vraća duljinu stringa
min()	vraća znak s najmanjom kodnom vrijednosti
max()	vraća znak s najvećom kodnom vrijednosti
ord()	vraća dekadski kod pojedinog znaka
chr()	vraća znak odgovarajućeg dekadskog koda
str()	vraća znakovni prikaz željenog broja
repr()	pretvara objekt bilo kojeg tipa u string

Primjer 6.

```
>>> a,b,c = "Programski",'jezik',"Python"  
>>> len(b)  
5  
>>> min(b)  
'e'  
>>> max(b)  
'z'  
>>> ord('P')  
80  
>>> chr(80)  
'P'  
>>> str(123)  
'123'  
>>> repr(34.56)  
'34.56'
```

1. Stringovi

1.2. Metode za rad sa stringovima

Python je i objektno orijentirani programski jezik pa varijablu tipa string možemo promatrati i kao objekt nad kojim upotrebljavamo unaprijed definirane metode koje su ugrađene u Pythonove standardne biblioteke. Metodu primjenjujemo tako da nakon imena varijable navedemo točku i ime metode u skladu s njezinom sintaksom.

metoda	opis
<code>s.capitalize()</code>	prvi znak u stringu pretvara u veliko slovo
<code>s.count(s1, poc, kraj)</code>	vraća broj pojavljivanja podstringa <code>s1</code> počevši od indeksa <code>poc</code> do indeksa <code>kraj</code> u stringu <code>s</code>
<code>s.find(s1, poc, kraj)</code>	vraća indeks prvoga pojavljivanja podstringa <code>s1</code> u stringu <code>s</code> od indeksa <code>poc</code> do indeksa <code>kraj</code> . Ako se on ne pojavljuje, tada vraća vrijednost -1
<code>s.index(s1, poc, kraj)</code>	vraća indeks prvoga pojavljivanja podstringa <code>s1</code> u stringu <code>s</code> od indeksa <code>poc</code> do indeksa <code>kraj</code> . Ako se on ne pojavljuje, tada vraća poruku o grešci (ValueError: substring not found)
<code>s.replace(s1, s2, koliko)</code>	zamjenjuje svako pojavljivanje podstringa <code>s1</code> u stringu <code>s</code> sa stringom <code>s2</code> . Ako se <code>koliko</code> navede, tada se zamjenjuje samo prvih <code>koliko</code> pojavljivanja stringa <code>s1</code> .
<code>s.upper()</code>	vraća kopiju stringa <code>s</code> u kojoj su sva slova velika
<code>s.lower()</code>	vraća kopiju stringa <code>s</code> u kojoj su sva slova mala
<code>s.title()</code>	vraća kopiju stringa <code>s</code> u kojoj svaka riječ stringa počinje velikim slovom
<code>s.swapcase()</code>	vraća kopiju stringa <code>s</code> u kojoj su sva velika slova zamijenjena malim slovima, a sva mala velikim
<code>s.center(duljina, znak)</code>	vraća kopiju stringa duljine <code>duljina</code> u kojoj je string <code>s</code> centriran, a slijeva i zdesna dodan je potreban broj znakova <code>znak</code>
<code>s.ljust(duljina, znak)</code>	vraća kopiju stringa duljine <code>duljina</code> u kojoj je string <code>s</code> lijevo poravnat, a desno je dodan potreban broj znakova <code>znak</code>
<code>s.rjust(duljina, znak)</code>	vraća kopiju stringa duljine <code>duljina</code> u kojoj je string <code>s</code> desno poravnat, a slijeva je dodan potreban broj znakova <code>znak</code>
<code>s.endswith(s1, poc, kraj)</code>	vraća vrijednost <code>True</code> ako string (ili njegov podstring) ako su navedeni parametri <code>poc</code> i <code>kraj</code> završava podstringom <code>s1</code>
<code>s.startswith(s1, poc, kraj)</code>	vraća vrijednost <code>True</code> ako string (ili njegov podstring) ako su navedeni parametri <code>poc</code> i <code>kraj</code> počinje navedenim podstringom <code>s1</code>
<code>s.rfind(s1)</code>	vraća indeks zadnje pojave podstringa <code>s1</code> u stringu <code>s</code>
<code>s.strip()</code>	vraća kopiju stringa <code>s</code> iz koje su obrisane sve vodeće i završne praznine
<code>s.lstrip()</code>	vraća kopiju stringa <code>s</code> iz koje su obrisane sve vodeće praznine
<code>s.rstrip()</code>	vraća kopiju stringa <code>s</code> iz koje su obrisane sve završne praznine
<code>s.isdigit()</code>	vraća vrijednost <code>1</code> ako su svi znakovi u stringu <code>s</code> znamenke
<code>s.isalnum()</code>	vraća vrijednost <code>True</code> ako string <code>s</code> sadržava najmanje jedan znak i ako su svi znakovi stringa alfanumerički (slova engleske abecede ili znamenke)
<code>s.isalpha()</code>	vraća vrijednost <code>True</code> ako su svi znakovi stringa <code>s</code> slova engleske abecede
<code>s.isupper()</code>	vraća vrijednost <code>True</code> ako su sva slova u stringu <code>s</code> velika
<code>s.islower()</code>	vraća vrijednost <code>True</code> ako su sva slova u stringu <code>s</code> mala
<code>s.join(s1)</code>	vraća string u kojem su pojedini elementi stringa <code>s1</code> povezani separatorom <code>s</code>
<code>s.split(s1)</code>	rastavlja string <code>s</code> obzirom na podstring <code>s1</code>
<code>s.expandtabs()</code>	vraća kopiju stringa <code>s</code> u kojoj su tab oznake (<code>\t</code>) zamijenjene odgovarajućim brojem praznina
<code>s.partition(s1)</code>	rastavlja string <code>s</code> na dio ispred podstringa <code>s1</code> , podstring <code>s1</code> i dio nakon podstringa <code>s1</code>
<code>s.zfill(n)</code>	vraća kopiju stringa <code>s</code> u kojoj je s lijeve strane dodan potreban broj znakova '0' da bi ukupna duljina bila <code>n</code>

1. Stringovi

Popis svih metoda definiranih na nekoj varijabli tipa string možemo dobiti naredbom `dir()` kako je prikazano u sljedećem primjeru:

`dir()`

Primjer 7.

```
>>> s = 'Programski jezik Python'  
>>> dir(s)  
['__add__', '__class__', '__contains__', '__delattr__',  
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
'__getattribute__', '__getitem__', '__getnewargs__',  
'__gt__', '__hash__', '__init__', '__iter__', '__le__',  
'__len__', '__lt__', '__mod__', '__mul__', '__ne__',  
'__new__', '__reduce__', '__reduce_ex__', '__repr__',  
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',  
'__str__', '__subclasshook__', 'capitalize', 'casefold',  
'center', 'count', 'encode', 'endswith', 'expandtabs',  
'find', 'format', 'format_map', 'index', 'isalnum',  
'isalpha', 'isdecimal', 'isdigit', 'isidentifier',  
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',  
'isupper', 'join', 'ljust', 'lower', 'lstrip',  
'maketrans', 'partition', 'replace', 'rfind', 'rindex',  
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',  
'startswith', 'strip', 'swapcase', 'title',  
'translate', 'upper', 'zfill']
```

Isti popis dobili bismo i da smo umjesto konkretnje varijable tipa string napisali `dir(str)`.

`help()`

Detaljnju pomoć za svaku metodu možemo dobiti naredbom `help()`.

Primjer 8.

```
>>> help(s.count)  
Help on built-in function count:  
  
count(...)  
    s.count(sub[, start[, end]]) -> int  
  
    Return the number of non-overlapping occurrences of sub-  
    string sub in  
    string S[start:end]. Optional arguments start and end are  
    interpreted as in slice notation.
```

1. Stringovi

Primjer 9.

```
>>> d = 'Programski\tjezik\tPython'  
>>> d.expandtabs()  
'Programski      jezik      Python'
```

Primjer 10.

```
>>> a = 'Programski'  
>>> a.find('s')  
7  
>>> a.index('s')  
7  
>>> a.find('z')  
-1  
>>> a.index('z')  
Traceback (most recent call last):  
  File "<pyshell#21>", line 1, in <module>  
    a.index('z')  
ValueError: substring not found
```

Primjer 11.

partition

```
>>> a,b,c = "Programski",'jezik',"Python"  
>>> a.partition('g')  
('Pro', 'g', 'ramski')  
>>> b.partition('z')  
('je', 'z', 'ik')  
>>> c.partition('t')  
('Py', 't', 'hon')  
>>> b.zfill(15)  
'000000000jezik'
```

U nastavku slijedi nekoliko zadataka da se podsjetimo svojstava stringa i pokažemo kako se problemi mogu riješiti njihovom uporabom.

Zadatak 1.

Napišite program koji će učitati string i na temelju njega formirati i ispisati drugi čiji su svi znakovi na parnim mjestima (drugi, četvrti,...) zamijenjeni znakom ‘-’.

Primjer testnih podataka:

ulaz	ulaz
Proba	Programski jezik Python
izlaz	izlaz
P-o-a	P-o-r-m-k- -e-i- -y-h-n

1. Stringovi

Opis algoritma

Prva je ideja da se prolazi od prvog do zadnjeg znaka učitanog stringa i ako mu je indeks neparan (indeks je prvog znaka 0), u novi string doda se znak ‘-’, u protivnom dodamo i-ti znak prvog stringa.

```
a = input()
b = ''
for i in range(len(a)):
    if i % 2 == 1:
        b = b + '-'
    else:
        b = b + a[i]
print(b)
```

Ako u rješenju kombiniramo i liste, dobijemo nešto kraći kod:

```
a = list(input())
s = "-".join(a[::2])
print(s)
```

join()

Zadatak 2.

Nakon napornog dana Edi se sjetio da za sutra mora još napisati i esej iz hrvatskog jezika. Prionuo je na posao i, sav zanesen inspiracijom, nije ni uočio da mu je na tipkovnici bila uključena tipka Caps Lock, tj. da tipka shift djeluje suprotno od očekivanog, tako da sve što je trebalo biti napisano velikim slovima piše malima i obratno, sve što je trebalo biti napisano malim slovima napisano je velikima. Pomognite Ediju i napišite program koji će zadani string ispisati pravilno.

Primjer testnih podataka:

ulaz	ulaz	ulaz
pROBA	eDI PIŠE ESEJ	sLAVONSKI bROD
izlaz	izlaz	izlaz
Proba	Edi piše esej	Slavonski Brod

Opis algoritma

Uporabom metode s.swapcase() do rješenja se dolazi u jednom koraku.

```
s = input("Zadajte string: ")
print(s.swapcase())
```

swapcase()

Zadatak 3.

Edi se u slobodno vrijeme bavi proučavanjem šifriranja i pronašao je još jedan zanimljiv način kodiranja poruka. Radi se o sljedećem: svaku riječ duljine N znakova kodira tako da prvih N/2 znakova napiše u obrnutom poretku, a zatim na to doda zadnjih N/2 početne riječi također napisane u obrnutom poretku. Tako će riječ *programski* biti zapisana *rgorpiksma*.

1. Stringovi

Primjer testnih podataka:

ulaz	ulaz	ulaz
programske	Proba	programiranje
izlaz	izlaz	izlaz
rgorpiksma	rPoab	argorpmejnari

Opis algoritma

Treba razlikovati dva slučaja, kad zadana riječ ima paran, odnosno neparan broj znakova.

```
a = input("Upišite riječ: ")
n = len(a)
if (n%2 == 0):
    b = a[0:n//2]
    c = a[n//2:n]
    print(b[::-1] + c[::-1])
else :
    b = a[0:n//2]
    c = a[n//2 + 1:n]
    print(b[::-1] + a[n//2] + c[::-1])
```

Zadatak 4.

Tajne poruke opet su zainteresirale Edija pa je smislio novi način nijihovog zapisivanja. Svaku poruku najprije zapiše u obrnutom poretku, a zatim u tako dobivenoj poruci svaka dva susjedna znaka zamijene mjesta. Napravite program koji će Ediju pomoći pišati tajne poruke.

Primjer testnih podataka:

ulaz	ulaz	ulaz
programske	jezik	Python
izlaz	izlaz	izlaz
kimsraogpr	ikezj	onthPy

Opis algoritma

Da bismo cijeli string a zapisali naopako, možemo upotrijebiti `a[::-1]`, nakon toga potrebno je samo "okrenuti" po dva znaka u stringu. Naravno, ako polazni string ima neparan broj znakova, zadnji znak ne mijenja poziciju.

```
a = input()
a = a[::-1]
b = ''
for i in range(0,len(a)-1,2):
    b = b + (a[i:i + 2])[::-1]
if len(a) % 2 == 1:
    b = b + a[len(a)-1]
print(b)
```

1. Stringovi

Zadatak 5.

Svi smo dobro upoznati s osnovnim pravilima za kreiranje lozinki za svoje profile na omiljenoj društvenoj mreži. Odjel za sigurnost i tajnost podataka te mreže u svojoj je zadnjoj verziji nažalost napravio jedan propust. Naime, sve lozinke koje u sebi sadržavaju jednak broj slova "V", "I", "R", "U", "S" (bez obzira radi li se o velikim ili malim slovima) nisu pouzdane i novi virus koji se pojavio može ih zaobići i tako pristupiti vašem profilu. Edi, koji stječe radno iskustvo odradujući ljetnu praksu u Odjelu za sigurnost i tajnost podataka, ima zadatak da za sve kreirane lozinke provjeri jesu li pouzdane ili ne.

Ulazni podaci:

Riječ duljine ne više od 100 znakova koja predstavlja lozinku za pristup omiljenoj društvenoj mreži.

Izlazni podaci:

"Pouzdana", odnosno "Nije pouzdana", ovisno o tome sadržava li lozinka različit ili isti broj slova "V", "I", "R", "U", "S".

Primjer testnih podataka:

ulaz	ulaz	ulaz
virusi	virus	ProgramskiJezikPython
izlaz	izlaz	izlaz
Pouzdana	Nije pouzdana	Pouzdana

Opis algoritma

Da bismo olakšali brojanje bitnih slova u stringu – lozinci, s obzirom na to da ne treba razlikovati velika i mala slova, poželjno je uporabom metoda `s.upper()` ili `s.lower()` osigurati da se string sastoji samo od velikih ili samo od malih slova i zatim metodom `s.count()` pristupiti prebrajanju.

```
aa = input("Upišite svoju lozinku: ")
aa = aa.upper()
d = len(aa)
bv = aa.count('V')
bi = aa.count('I')
br = aa.count('R')
bu = aa.count('U')
bs = aa.count('S')
if (bv == bi) and (bv == br) and (bv == bu) and (bv == bs) :
    print("Nije pouzdana")
else :
    print("Pouzdana")
```

count()

Drugo rješenje istog problema

```
lozinka = input().upper()
V = I = R = U = S = 0
```

1. Stringovi

```
for c in lozinka:
    if c == 'V': V += 1
    if c == 'I': I += 1
    if c == 'R': R += 1
    if c == 'U': U += 1
    if c == 'S': S += 1
if V == I == R == U == S:
    print('Nije pouzdana')
else: print('Pouzdana')
```

Zadatak 6.

Edi uči svoju mlađu sestru Evu zbrajati. Da bi joj lakše objasnio, za početak zbraja samo brojeve 1, 2 i 3. Nakon što joj je pokazao kako se zbraja, Edi je sestri zadao puno zadataka koje ona treba rješiti dok se on opušta uz omiljenu igricu. Eva je svladala zbrajanje ako su brojevi napisani u nepadajućem poretku, odnosno zna zbrojiti $1+1+3$, ali ne može izračunati koliko je $1+3+1$. Kako bi Eva rado odradila zadaću prije prikazivanja omiljenog crtića, moli vas za pomoć, odnosno da zadatke koje je dobila preformulirate u oblik iz kojeg će znati izračunati zbroj.

Ulazni podaci:

String koji sadržava samo znamenke 1, 2, 3 i znak +.

Izlazni podaci:

Preoblikovan string u kojem su znamenke u nepadajućem poretku.

Primjer testnih podataka:

ulaz	ulaz	ulaz
1+2+1+2	3+2+1	3+3+2+3+2+1
izlaz	izlaz	izlaz
1+1+2+2	1+2+3	1+2+2+3+3+3

Opis algoritma

Jedna je od ideja da u učitanom stringu prebrojimo koliko se puta pojavljuje koja znamenka i zatim ispis prilagodimo broju pojavljivanja pojedine znamenke.

```
n = input("Upišite izraz: ")
s = ''
b1 = n.count('1')
b2 = n.count('2')
b3 = n.count('3')
for i in range (b1):
    s = s + '1'
    if i < b1 - 1:
        s = s + '+'
for i in range (b2):
```

1. Stringovi

```
if b1 or i!=0:  
    s = s + '+'  
s = s + '2'  
for i in range (b3):  
    if b1 or b2 or i != 0:  
        s = s + '+'  
    s = s + '3'  
print(s)
```

Postoji elegantnije i općenitije (gdje nije nužno da se radi samo sa znamenkama 1, 2, i 3) rješenje kombiniranjem stringa i liste.

```
l = list(input("Upišite izraz: "))  
g = l[::2]  
g.sort()  
l[::2] = g  
s = str('')  
for i in l:  
    s = s.__add__(i)  
print(s)
```

U ovom rješenju koristili smo se specijalnom metodom kojom se ostvaruje operator +, a to je metoda `__add__()`.

Zadatak 7.

Napišite funkciju koja će znakove dobivenog stringa zarotirati za n mesta udesno ako je n pozitivan broj, a u suprotnom za n mesta ulijevo.

Primjer testnih podataka:

ulaz	ulaz	ulaz
Programski	jezik	Python
5	- 2	3
izlaz	izlaz	izlaz
amskiProgr	zikje	honPyt

Opis algoritma

```
def rot(s,n):  
    l = len(s)  
    if (n < 0):  
        n = l + n  
        n = n % l  
    if (n):  
        s = s[-n:] + s[:l-n]  
    return s  
  
s = input("Zadaj string: ")  
n = int(input("N = : ( pozitivan - rotacija u desno, negativan - rotacija u lijevo)"))  
print(rot(s,n))
```

1. Stringovi

Zadatak 8.

Istražujući internet, Edi je pronašao zanimljivu igru sa slovima – Slovojed. U toj igri za zadani string potrebno je ispisati koje je slovo Slovojed, tj. koje slovo može “pojesti” najviše slova ako su pravila sljedeća: Svako veliko slovo pojede svako malo slovo. Također, svako veliko slovo može pojesti sva velika slova koja su u abecednom rasporedu ispred njega. Isto tako, i svako malo slovo može pojesti sva mala slova koja su u abecednom rasporedu ispred njega. Edi treba odrediti slovo koje je “pojelo” najviše slova (uvijek je samo jedno takvo slovo).

Primjer testnih podataka:

ulaz	ulaz	ulaz
informatika	ProgramskiJezik	HiperBiliRubiNemija
izlaz	izlaz	izlaz
k	P	R

Opis algoritma

Prikazano rješenje koristi se pomoćnim stringom u kojem definira potreban poredak velikih i malih slova prema uvjetima iz zadatka.

```
vs = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
s = input()
b = 0
i = 0
m = 0
ss = ""
sl = s[i]
while i < len(s)-1:
    if i < len(s)-1 and vs.find(sl) > vs.find(s[i + 1]):
        b = b + 1
        if b > m:
            m = b
            ss = sl
    else:
        if vs.find(sl) < vs.find(s[i + 1]):
            sl = s[i + 1]
            b = 0
    i = i + 1
if m == 0:
    ss = sl
print(ss)
```

Zadatak 9.

Edi osim kriptografije i programiranja jako voli i proslave rođendana pa ga zanima u kojem će mjesecu imati najviše prilika biti na rođendanskim zabavama.

1. Stringovi

Uzni podaci:

U prvom retku učita se prirodni broj N($1 \leq N \leq 1000$) koji predstavlja broj prijatelja. U narednih N linija upisuje se datum rođenja za svakog od prijatelja, i to u standardnom formatu: dan, mjesec, godina rođenja odvojeni znakom ". " (npr. 25.3.2000.).

Izni podaci:

Za sve mjesece ispisati pripadajući broj rođendana.

Primjer testnih podataka:

ulaz	ulaz
Koliko učenika: 5	Koliko učenika: 10
13.2.2000.	11.11.2000.
14.5.2001.	15.5.2000.
24.2.2001.	18.6.2001.
8.3.1999.	8.6.1999.
22.12.2004.	6.8.2000.
	8.8.1999.
	22.11.1999.
	31.12.2001.
	24.6.1999.
	13.12.2000.
izlaz	izlaz
1 0	1 0
2 2	2 0
3 1	3 0
4 0	4 0
5 1	5 1
6 0	6 3
7 0	7 0
8 0	8 2
9 0	9 0
10 0	10 0
11 0	11 2
12 1	12 2

Opis algoritma

Uočiti da iz datuma treba izdvojiti mjesec, a on se nalazi između dva znaka ". ".

```
m = [0,0,0,0,0,0,0,0,0,0]
n = int(input("Koliko učenika: "))
for i in range(n):
    x = input()
    p = x.find('.')
```

1. Stringovi

```
k = x.find('.',p + 1,len(x))
j = int(x[p + 1:k])-1
m[j] = m[j] + 1
for i in range(12):
    print(i + 1,m[i])
```

Zadatak 10.

Igre sa stringovima sve više zanimaju Edija. Otkrio je još jednu, čiji je cilj otkriti maksimalni broj koraka za uklanjanje podstringa B iz stringa A. Jednim korakom smatra se uklanjanje svih podstringova pri prolasku kroz string A od prvog do zadnjeg znaka. Sljedeći korak slijedi radi uklanjanja novonastalih podstringova B nakon provedbe prošlog koraka.

Primjer testnih podataka:

ulaz	Pojašnjenje
aabcbck abc	Nakon prvog koraka, uklanjanja podstringa abc iz zadanog stringa, dobije se abck, koji opet u sebi sadržava podniz B=abc.
izlaz	
2	Nakon drugog koraka string A ima samo jedan znak, slovo k.

Opis algoritma

Osnovna je ideja da dok god u stringu A postoji podstring B, brišemo podstring B iz stringa A



split()

```
A,B = map(str,input().split())
b = 0
s1 = A
while A.find(B) > -1:
    b = b + 1
    s1 = A.replace(B,'')
    A = s1
print(b)
```

Nešto sažetija realizacija iste ideje:

```
a,b = input().split()
s = 0
while b in a:
    s += 1
    a = a.replace(b,'')
print(s)
```

1. Stringovi

Teorijski zadaci

1. Što će se ispisati nakon izvršenja sljedećeg koda:

```
s="petak je danas"  
s=s[9:]+s[5:9]+s[0:5]  
print(s)
```

2. Što će se ispisati nakon izvršenja sljedećeg koda:

```
a='informatika'  
print(a[2:5])
```

3. Što će ispisati sljedeći programski kod?

```
a='Programski jezik Python'  
print(a[::-3])
```

4. Što će ispisati sljedeći programski kod?

```
s='mad'  
s[:1] + s[1:3] + s[-2] + s[0]
```

5. Što će ispisati sljedeći programski kod?

```
a="UNO"  
b='M'  
aa=list(a)  
for i in range (len (a)+1):  
    aa.insert(i,b)  
    c=''.join(aa)  
    print(c)  
    aa=list(a)
```

6. Što će ispisati sljedeći programski kod:

```
a=list(input())  
a.sort()  
s=''.join(a)  
print(s)
```

ako se nakon pokretanja upiše Informatika?

7. Što će ispisati sljedeći programski kod?

```
s='Programski jezik Pascal'  
s=s[:-6]+'Python'  
print(s)
```

8. Što će ispisati sljedeći programski kod?

```
s="Luka je pojeo luk prije nego što  
je odapeo luk"  
print(s.count('luk')-s.lower().  
count('luk'))
```

9. Koji od sljedećih izraza imaju vrijednost True?

- a) 'abc123'.isdigit()
- b) '12.34'.isalnum()
- c) 'apple'.upper() == 'APPLE'
- d) 'abc123'.isalnum()

10. Riječ B anagram je riječi A ako je možemo dobiti preslagivanjem slova riječi A. Napišite program koji će učitati dvije riječi i provjeriti radi li se o anagramu.

11. Napišite program koji će na temelju učitanog stringa formirati i ispisati drugi string kod kojeg su svi znakovi na neparnim mjestima (prvo, treće,...) zamijenjeni znakom '-'.

12. Napišite program koji će učitani string ispisati tako da iza svakog znaka, uključujući i zadnji, bude isписан znak razmaka. (PROBA → P R O B A)

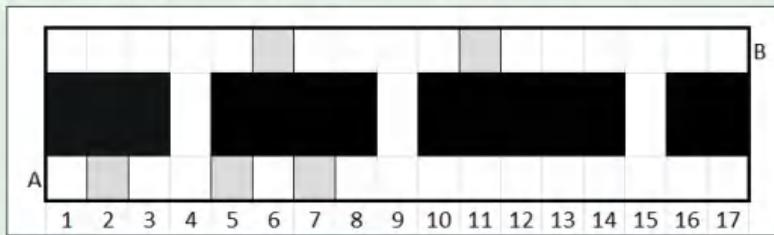
13. Napišite program koji će na temelju učitanog stringa formirati i ispisati drugi string kod kojeg su izostavljeni svi znakovi na parnim mjestima. (Proba → Poa)

1. Stringovi

Od problema do rješenja

Problem 1.

Zamisli da kasniš na nastavni sat i moraš što brže dotrčati do učionice koja se nalazi na drugom kraju škole. Škola ima dva hodnika: lijevi i desni. Ulaz je na početku desnog hodnika, a učionica na kraju lijevog. Na slici je plan škole koji odgovara prvom testnom podatku.



Ulaz u školu označen je slovom A, a učionica slovom B. U lijevom i u desnom hodniku u svakom se zasiviljenom kvadratiču nalazi po jedan učitelj koji se, kada prođeš pored njega, ljuti na tebe zbog kašnjenja. U prolazima koji spajaju lijevi i desni hodnik nema učitelja. Želiš doći od ulaza do svoje učionice tako da prođeš pokraj što je moguće manje "ljutih" učitelja. Napiši program koji za zadane pozicije "ljutih" učitelja u lijevom i u desnom hodniku te pozicije prolaza određuje najmanji mogući broj učitelja kraj kojih moraš proći na putu do učionice.

Uzeti podaci

U prvom retku nalaze se prirodni broj **N** ($1 \leq N \leq 40$), širina škole, tj. broj kvadratiča u svakom hodniku. U sljedeća tri retka nalazi se po **N** znakova. Prvi niz znakova predstavlja lijevi hodnik, drugi niz srednji dio škole (zidove i prolaze između gornjeg i donjeg hodnika), a treći niz predstavlja desni hodnik. U prvom i trećem nizu znak '.' (točka) predstavlja prazni kvadratič, a znak 'U' kvadratič u kojem je "ljuti" učitelj. U drugom nizu znak 'X' predstavlja zid, a znak '.' prolaz između lijevog i desnog hodnika.

Napomena: U drugom se redu neće pojavit dva uzastopna znaka '.'. Postojat će barem jedan prolaz između hodnika; ni u prvom ni u N-tom kvadratiču neće biti prolaza. Učitelji se nikada neće nalaziti odmah ispred prolaza.

Izlazni podaci

Minimalni broj ljutih učitelja kraj kojih moraš proći.

Primjeri testnih podataka

RB	Uzeti	Izlaz	Objašnjenje
1.	17U.....U..... XXX.XXXX.XXXXX.XX .U..U.U.....	2	Srest ćeš najmanje ljudih učitelja ako ideš desnim hodnikom od ulaza do kvadratiča 4, pa lijevim hodnikom od kvadratiča 4 do kvadratiča 9, onda desnim hodnikom od kvadratiča 9 do kvadratiča 15 te lijevim hodnikom od kvadratiča 15 do učionice.
2.	10 UU.UUU..UU XX.XXX.XXX U...UU....	5	

1. Stringovi

Opis algoritma:

Ovo je jedan od onih zadataka u kojem do rješenja dolazimo pitajući se: Kako bismo postupili da je riječ o stvarnoj situaciji u kojoj smo se našli? Sigurno bismo morali ići desnim hodnikom do prvog prolaza. Onda bismo krišom pogledali iza zida i ispred sebe te vidjeli koliko se ljudi učitelja nalazi od trenutnog do sljedećeg prolaza u desnom, a koliko u lijevom hodniku. Očito je da bismo kretanje nastavili onim hodnikom u kojem ima manje učitelja. Na kraju, od zadnjeg prolaza do učionice morali bismo se kretati lijevim hodnikom. Ideja je tu. Pitanje je sada kako tu ideju implementirati i kodirati.

simulacija

Programski kod:

Ulagni podaci zadani su u obliku stringa pa ćemo ih kao takve i učitati u tri varijable: *lijevi*, *srednji* i *desni*.

```
N = int(input())
lijevi = input()
srednji = input()
desni = input()
```

Prebrojimo koliko ima učitelja u desnom hodniku od njegova početka do prvog prolaza. Ustvari trebamo prebrojiti koliko u stringu *desni* ima znakova 'U' počevši od početka pa sve do prvog pojavljivanja znaka '.'. U varijablu *ukupno* zapisivat ćemo broj učitelja pored kojih smo prošli na svojem putu.

```
ukupno = desni.count('U', 0, srednji.index('.'))
```

U nastavku trebamo za svaki dio između dvaju prolaza provjeriti koliko ima učitelja u lijevom, a koliko u desnom hodniku. Krećemo se onim dijelom u kojem ima manje učitelja. Kada analiziramo jedno područje, zatvorit ćemo prolaz na početku dijela i nastaviti dalje. Analizu ćemo ponavljati sve dok ne dođemo do posljednjeg dijela, koji ćemo onda posebno analizirati.

```
while srednji.find('.') != srednji.rfind('.'):
    pocetak = srednji.index('.')
    srednji = srednji.replace('.', 'X', 1)
    kraj = srednji.index('.')

    lijevo = lijevi[pocetak:kraj].count('U')
    desno = desni[pocetak:kraj].count('U')
    if lijevo < desno:
        ukupno += lijevo
    else:
        ukupno += desno
```

Pribrojimo u varijablu *ukupno* broj učitelja u lijevom hodniku od zadnjeg prolaza do kraja.

```
ukupno += lijevi.count('U', kraj, N)
print(ukupno)
```

1. Stringovi

Problem 2.

Ovce su smještene između ograda i mirno pasu travu. Iznenada su se pojavili vukovi, upali u prostor između ograda i počeli napadati ovce.

Srećom, ovce se mogu braniti i istjerati vukove ako ih između dviju ograda ima više od vukova. U protivnom, ako ovaca ima manje ili jednako od vukova, vukovi istjeraju sve ovce između dviju ograda.

Napiši program koji će učitati string koji se sastoji od znakova # (ograda), O (ovca) i V (vuk). Taj string opisuje situaciju točno u trenutku kada su vukovi uskočili. Program treba ispisati string koji opisuje situaciju nakon što vukovi istjeraju sve ovce koje mogu i ovce istjeraju sve vukove koje mogu.

Preciznije:

- ako ovce istjeraju vukove između dviju ograda, onda između tih dviju ograda trebaju ostati samo znakovi O
- ako vukovi istjeraju ovce između dviju ograda, onda između tih dviju ograda trebaju ostati samo znakovi V.

Ulazni podaci

String koji se sastoji od znakova #, O i V te počinje i završava znakom #. Između dviju susjednih ograda uvijek će se nalaziti barem jedna životinja.

Izlazni podaci

String koji predstavlja situaciju nakon što vukovi istjeraju sve ovce koje mogu i ovce istjeraju sve vukove koje mogu. Znakovi #, O i V od kojih se treba sastojati taj string imaju isto značenje kao i kod ulaznih podataka.

Primjeri testnih podataka

	ulaz	ulaz
Opis algoritma:	#oooo#vovo#oovvv#	#vvo#oo#vov#
	izlaz	izlaz
	#000#vv#vvv#	#vv#oo#vv#

Temelj rješenja zadatka

Temelj rješenja zadatka usporedba je broja pojavljivanja znakova 'O' i 'V' u dijelovima stringa između dvaju pojavljivanja znaka '#'. Uočimo da se kao separator tih ključnih dijelova pojavljuje isti znak '#'. Zbog toga ćemo kreirati listu L koristeći se naredbom *split* i na taj način vrlo lako dobiti sve ključne dijelove. Elemente te liste trebamo korigirati ovisno o ispunjenosti uvjeta zadatka o brojčanoj premoći ovaca nad vukovima, odnosno ovisno o tome kojih znakova, 'O' ili 'V', ima više. Na kraju ćemo tako izmijenjenu listu ponovno pretvoriti u string koristeći se naredbom *join*.

Programski kod:

```
tor = input()
L = tor.split('#')
for i in range(len(L)):
```

1. Stringovi

```
if L[i].count('O') > L[i].count('V'):
    L[i] = 'O' * L[i].count('O')
else:
    L[i] = 'V' * L[i].count('V')

print('#'.join(L))
```

Problem 3.

U povijesti šifriranja poznata je Cezarova šifra, nazvana po velikom rimskom vojskovođi Cezaru. Možda negdje postoji još neotkrivena i Neronova šifra, nazvana po slavnom caru Rimskog Carstva, Neronu. Neron je tom šifrom latinske riječi **pretvarao** u prirodne brojeve. Opišimo kako za zadani broj možemo otkriti koju latinsku riječ on predstavlja. Osnovu postupka otkrivanja čini **ključna riječ**. Svakom slovu ključne riječi **priđružujemo** njegov redni broj unutar ključne riječi. Zadani prirodni broj **N** pretvaramo u riječ tako da svaku znamenku tog broja **zamjenimo** slovom koje odgovara toj znamenki u ključnoj riječi. Ako neka znamenka nema svoje pridruženo slovo, tada se ta znamenka zamjenjuje slovom "X". Traženu latinsku riječ dobijemo tako da iz dobivene riječi izbacimo sva slova "X". Npr.:

Ključna riječ: OREGON						Prirodni broj: 6329586						
1	2	3	4	5	6	6	3	2	9	5	8	6
O	R	E	G	O	N	N	E	R	X	O	X	N

Latinska riječ koja odgovara zadanim prirodnim broju:

NERON

Napiši program koji za zadalu ključnu riječ i prirodni broj **N** ispisuje traženu latinsku riječ.

Uzeti podaci

U prvom retku nalazi se ključna riječ **S**, duljine najviše 9 znakova, sastavljena od velikih slova engleske abecede (bez slova "X"). U drugom retku nalazi se prirodni broj **N**.

Izlazni podaci

U jednom retku treba ispisati traženu latinsku riječ iz teksta zadatka.

Primjeri testnih podataka

ulaz	ulaz	ulaz
OREGON	PIJETAO	MARUN
6329586	18740596	24341
izlaz	izlaz	izlaz
NERON	POETA	AURUM

Opis algoritma:

Prirodno bi bilo prirodni broj **N** učitati kao broj naredbom `int(input())`. Međutim, u

ovom slučaju i ključnu riječ i zadani broj treba učitati kao stringove. To će nam olakšati posao. Svaku znamenku u broju **N** (koje su sada znakovi) treba zamjeniti onim slovom iz ključne riječi **ključna** čiji je indeks jednak brojčanoj vrijednosti znaka/znamenke koju zamjenjujemo. Kako nije moguće izravno mijenjati znak u stringu, onda ćemo kreirati novi string *rijec*. Primijetimo da dio zadatka o izbacivanju znaka **X** iz nove riječi slobodno možemo zanemariti jer ćemo prilikom kreiranja te riječi zanemariti sve znamenke koje



Neron - peti
rimski car,
zadnji iz
Julijsko-Klaudi-
jevske dinastije. Bio
je rimski car od 54.
do 68. godine.

simulacija

1. Stringovi

su veće od duljine ključne riječi ili jednake nuli. U zadatku se krije još jedna zamka, a to je da znamenka nula nema svoj pripadajući znak u ključnoj riječi.

Programski kod:

```
kljucna = input()
N = input()
rijec = ""
for i in N:
    if int(i) <= len(kljucna) and int(i) != 0:
        rijec += kljucna[int(i) - 1]
print(rijec)
```

Problem 4.

Ima jedan robot na čijem ekranu piše rečenica sastavljena od pet riječi. Kada robot čuje neki peteroznamenkasti broj ABCDE, pri čemu su A, B, C, D, E znamenke od 1 do 5 (svaka se javlja jednom), on mijenja poredak riječi u rečenici na ekranu, i to na sljedeći način:

- prva riječ prelazi na A-to mjesto u rečenici,
- druga riječ prelazi na B-to mjesto,
- treća riječ prelazi na C-to mjesto,
- četvrta riječ prelazi na D-to mjesto,
- peta riječ prelazi na E-to mjesto u rečenici.

Napiši program koji za dani broj koji je prvi izgovoren pronalazi broj koji treba izgovoriti nakon toga da se riječi vrate u prvotni poredak!

Ulazni podaci

U jedinome retku nalazi se peteroznamenkasti broj koji je prvi izgovoren. On se sastoji od znamenaka 1, 2, 3, 4, 5, pri čemu se svaka od njih javlja točno jednom.

Izlazni podaci

U jedini redak ispiši traženi peteroznamenkasti broj.

Primjeri testnih podataka

ulaz	ulaz
31542	12354

Opis algoritma:

Opet možemo učitati **N** kao prirodni broj, ali tada moramo

izlaz	izlaz
25143	12354

"izvlačiti" njegove znamenke uzastopnim dijeljenjem s 10. Ovdje je lakše **N** promatrati kao string od pet znakova koje redom prolazimo for-petljom.

Unutar petlje najprije želimo od trenutnoga, **i**-tog znaka stringa **N** dobiti odgovarajući jednoznamenkasti broj (nazovimo ga **k**), npr. '4' → 4. To činimo tako da od ASCII vrijednosti dotičnoga znaka oduzmemo ASCII vrijednost znaka '1'. Potom se koristimo glavnim zaključkom: ako na **i**-tom mjestu od **N** stoji znamenka **k**, onda na **k**-tom mjestu tra-

1. Stringovi

ženog rješenja mora stajati znamenka **i**. Dakle, u nizu koji predstavlja rješenje valja na **k**-to mjesto upisati **i**.

Naravno, na koncu ispisujemo niz znamenaka koji predstavlja rješenje, a popunili smo ga tijekom gore opisane petlje.

Programski kod:

```
N = input()
M = [0 for i in range(5)]
for i in range(5):
    M[ord(N[i]) - ord('1')] = i + 1
print(''.join(map(str, M)))
```

Problem 5.

Zadan je niz sa znakovima: '?', '(', ')', '[', ']', '{', '}'. U nizu svaki znak '?' treba zamijeniti s nekim od znakova ')', ']', '}' tako da se dobije pravilan niz. Pravilan niz definiramo na poznati način:

1. (), [], {} su pravilni nizovi;
2. Ako je **X** pravilan niz, onda su i nizovi **(X)**, **[X]**, **{X}** pravilni;
3. Ako su **X** i **Y** pravilni nizovi, onda je i niz **XY** pravilan.

Primjeri su pravilnih nizova: ((())), ([]), {}(), []{}{}, ([{}]{}){}, a primjeri nepravilnih nizova: {}, ()[], (()){}, [[()]], {{()}}(). Rješenje će uvijek postojati i bit će jedinstveno.

Ulazni podaci

U prvom retku nalazi se niz znakova iz teksta zadatka.

Izlazni podaci

U jednom retku treba ispisati pravilan niz znakova dobiven postupkom opisanim u zadatku.

Primjeri testnih podataka

ulaz	ulaz	ulaz
({?})	[?]	{[?] ?(?[]
izlaz	izlaz	izlaz
({})	[]	{[]}{()}[]

Opis algoritma:

Za svaki upitnik u učitanom stringu **zagrada** trebamo odrediti koju **otvorenu za-**

gradu zatvara. Ideja je da istovremeno analiziramo string **zagrada** počevši slijeva nadesno i izgrađujemo novi string **pravilan**. U novi string redom ćemo prepisivati sve otvorene zgrade, dok ćemo kada nađemo na zatvorenu zgradu ili upitnik napraviti jednu dodatnu provjeru. Neka se **prva** zatvorena zgrada (ili upitnik) nalazi na mjestu **i**. Primijetimo da ta zgrada zatvara zadnju otvorenu zgradu koja se sigurno nalazi na mjestu **i-1**. U suprotnom, niz ne bi bio pravilan. Kada nađemo takav par zagrada, možemo ih izbaciti iz stringa **zagrada** i ponoviti postupak. **Zgrade** s izbačenim parom zagrada

1. Stringovi

brisanje uzastopnih znakova

i dalje će biti pravilan string i neće doći do promjene rješenja. Postupak ponavljamo sve dok postoji string zagrade.

Tijekom implementacije te ideje trebat ćemo iz stringa zagrade obrisati dva uzastopna znaka.



Znamo da metodom `replace` možemo zamijeniti svako pojavljivanje podstringa `s1` sa stringom `s2`. Međutim, nije moguće mijenjati jedan ili više znakova samo ako znamo na kojem se indeksu nalaze. Zbog toga ćemo iskoristiti svojstvo stringa da možemo pristupiti njegovim segmentima i na taj način zaobići znak (dva znaka) koji želimo izbaciti. Npr.:

```
>>> s = 'Python'
>>> s = s[:1] + s[3:] # 'P' + 'hon'
>>> s
'Phon'
```

Programski kod:

```
zagrade = input()
pravilan = ''

gdje = 0
while zagrade != '':
    ok = True
    if zagrade[gdje] in ['(', '[', '{']:
        pravilan += zagrade[gdje]
    elif zagrade[gdje] in [')', ']', '}']:
        pravilan += zagrade[gdje]
        zagrade = zagrade[:gdje-1] + zagrade[gdje + 1:]
        ok = False
    elif zagrade[gdje] == '?':
        if zagrade[gdje - 1] == '(':
            pravilan += ')'
        if zagrade[gdje - 1] == '[':
            pravilan += ']'
        if zagrade[gdje - 1] == '{':
            pravilan += '}'
        zagrade = zagrade[:gdje-1] + zagrade[gdje + 1:]
        ok = False

    if ok:
        gdje += 1
    else:
        gdje -= 1

print(pravilan)
```

1. Stringovi

Problem 6.

Zmija je poznata računalna igrica. Zamislimo da se naša zmija može kretati samo unutar jednog retka i to samo ili u lijevu ili u desnu stranu. Redak u kojem se nalazi zmija podijeljen je na N kvadratića. Na početku igre zmija je duljine 1 i zauzima jedan kvadratić (znak 'o'). Prazna mjesta označena su točkom (znak '.'). Vrijede sljedeća pravila:

- **Kretanje.** Zmija se kroz redak kreće pomičući se svakim svojim dijelom za jedan kvadratić u smjeru kretanja. Jedan pomak zmije traje **jednu** sekundu. Npr., ako redak u nekom trenutku izgleda ovako: "...oo...", nakon pomaka ulijevo izgledat će ovako: ".ooo....".
- **Izlazak.** Može se dogoditi da dio zmije ili cijela zmija izađe izvan retka. U tom slučaju dio zmije koji se još uvijek nalazi unutar retka (ako postoji takav dio) nastavlja se kretati kao i prije. Npr., ako redak izgleda ovako "...oo", nakon pomaka udesno izgledat će ovako "...o...".
- **Hrana.** Kada se zmija pomakne na polje gdje se nalazi hrana (**znak "+"**), zmija se produžuje za jedan kvadratić. Npr., ako redak izgleda ovako: ".+oo...", nakon pomaka ulijevo izgledat će ovako: ".ooo...".
- **Otrovna hrana.** Kada se zmija pomakne na polje gdje se nalazi otrovna hrana (**znak "x"**), nakon pomaka skraćuje se za jedan kvadratić. Npr., ako početni redak izgleda ovako: "..oox..", nakon pomaka udesno izgledat će ovako: "...o..".
- **Supermoć.** Od trenutka kada zmija naiđe na supermoć (**znak "!"**), svaka hrana koju zmija od tada pojede produžuje zmiju za jedan kvadratić – čak i otrovna. Supermoć nije hrana.
- **Prepreka.** Kada se zmija pomakne na polje s preprekom (**znak "#"**), zmija nestaje s mape. Isto se dogodi kada zmija postane duljine nula.

Na temelju početnog stanja retka u kojem se nalazi zmija odredi i ispiši kako izgleda redak nakon T sekundi u slučaju da se krećemo desno i u slučaju da se krećemo lijevo.

Ulagani podaci

U prvom retku nalazi se prirodni broj **N** ($1 \leq N \leq 100$). U drugom retku nalazi se **N** znakova koji opisuju izgled retka u kojem se nalazi zmija. Svaki znak opisuje jedan kvadratić retka kako je opisano u tekstu zadatka. U trećem retku nalazi se prirodni broj **T** ($1 \leq T \leq 100$), broj sekundi.

Izlazni podaci

U prvi redak ispiši stanje nakon kretanja udesno, a u drugi za kretanje ulijevo.

Primjeri testnih podataka	ulaz	ulaz	ulaz
	8	9	9
	++ . o + ...	x . + o ! x x x +	# . x ! o + x x +
	4	4	4
	izlaz	izlaz	izlaz
	++ oo	x . + . o o o o +	# . x ! +
	oo . . + ! x x x +	# + x x +

1. Stringovi

Opis algoritma:

složena
simulacija

Ovo je slojevit zadatak za čije će nam rješavanje trebati dosta strpljenja. Krenimo redom. Opišimo algoritam za slučaj kada se zmija kreće udesno. U svakom trenutku moramo pamtiti koji je krajnji desni vidljivi kvadratič koji pripada zmiji (*glava_zmije=redak[index('o')]*), kolika je vidljiva duljina zmije (*duljina=1*), je li zmija živa i ima li zmija supermoć (*power=False*). Je li zmija živa, možemo pamtiti u istoj varijabli kao i duljinu zmije, tj. ako zmija više nije živa, samo postavimo duljinu zmije na 0. Krenimo simulirati kretanje zmije tako što ćemo mijenjati vrijednosti navedenih varijabli ovisno o trenutnom stanju u retku. Simulaciju prekidamo kada duljina zmije padne na nulu ili kada simuliramo svih T sekundi. U svakom koraku vrijedi da se *glava_zmije* poveća za 1, osim u slučaju kada je zmija na rubu retka – tada se *glava_zmije* ne mijenja. Primjetimo da određena pravila igre možemo definirati i ovako:

- Ako je *redak[glava_zmije]* jednak '.', onda se *duljina* ne mijenja;
- Ako je *redak[glava_zmije]* jednak '#', postavljamo *duljina* na 0;
- Ako je *redak[glava_zmije]* jednak '+', povećavamo *duljina* za 1;
- Ako je *redak[glava_zmije]* jednak '!', postavljamo *power* na *true*;
- Ako je *redak[glava_zmije]* jednak 'x', a *power* je *false*, onda povećavamo *duljina*;
- Ako je *redak[glava_zmije]* jednak 'x', a *power* je *true*, onda smanjujemo *duljina*.

Sve kvadratiče kroz koje je zmija prošla i ostala živa tijekom simulacije moramo postaviti na '.' jer su vrijednosti zapisane na njima nestale. Kad je simulacija gotova, moramo još označiti polja u kojima se nalazi zmija.



Znamo da metodom *replace* možemo zamijeniti svako pojavljivanje podstringa *s1* sa stringom *s2*. Međutim, nije moguće mijenjati jedan ili više znakova samo ako znamo na kojem se indeksu u stringu nalaze. U takvim situacijama možemo iskoristiti svojstva lista. Evo jednog načina kako to možemo napraviti:

```
def zamijeni(s, gdje, znak):  
    lista = list(s)  
    lista[gdje] = znak  
    s = ''.join(lista)  
    return s
```

Uočite da među metodama za rad sa stringovima ne postoji metoda logičnog naziva *reverse* koja bi zamijenila poredak znakova u stringu. Nije ni potrebna jer se zamjena obavlja na trivijalan način naredbom *a = a[::-1]*.

Da ne bismo morali dvaput implementirati taj algoritam, kretanje uljevo riješit ćemo trikom: možemo obrnuti cijeli redak, pozvati funkciju koja riješi zadatak ako se zmija kreće udesno te ispisati unatrag dobiveni izgled retka.

Programski kod:

```
def solve(idemo_lijevo, n, a, t):  
    if idemo_lijevo:  
        a = a[::-1]  
  
    b = a[:]
```

1. Stringovi

```
glava_zmije = a.index('o')
b = zamjeni(b, glava_zmije, '.')
duljina = 1
power = False

for i in range(t):
    glava_zmije += 1

    if glava_zmije >= n:
        glava_zmije = n - 1
        duljina -= 1
        if duljina == 0 :
            break
        continue

    if a[glava_zmije] == '#':
        duljina = 0
    elif a[glava_zmije] == '+':
        b = zamjeni(b, glava_zmije, '.')
        duljina += 1
    elif a[glava_zmije] == 'x' :
        b = zamjeni(b, glava_zmije, '.')
        if power :
            duljina += 1
        else :
            duljina -= 1
    elif a[glava_zmije] == '!':
        b = zamjeni(b, glava_zmije, '.')
        power = True

    if duljina == 0 :
        break

for i in range(duljina) :
    b = zamjeni(b, glava_zmije - i, 'o')

if idemo_lijevo:
    b = b[::-1]
print(b)

N = int(input())
redak = input()
T = int(input())

solve(False, N, redak, T)
solve(True, N, redak, T)
```

2. Lista listā

2.1. Uvod

Lista

Prije nego što uvedemo pojam liste lista, prisjetimo se što su to uopće liste. **Lista** je najkorišteniji složeni tip podataka u Pythonu. Varijabla tipa lista sadržava elemente navedene unutar uglatih zagrada i odvojene zarezom. Element liste može biti bilo koji tip podataka, a nama je u ovom poglavlju najzanimljivije što to može biti i lista. Svaki element u listi jednoznačno je određen svojom pozicijom u listi, a međusobno ih razlikujemo prema njihovu indeksu. Prvi element u listi ima indeks nula, drugi jedan itd. Znamo:

- da je lista promjenjiva struktura koja dopušta izravan pristup pojedinačnim elementima (navođenjem imena liste i odgovarajućeg indeksa unutar uglatih zagrada) i promjene njihovih vrijednosti
- da se iz liste mogu izdvojiti i mijenjati i neki njezini pojedinačni dijelovi
- da se liste mogu konkatenerati (+, zbrajanje) i replicirati (*, množenje)
- da u ugniježđenoj listi možemo pristupati elementima u elementima.

Listama smo se koristili kada smo više podataka istog ili različitog tipa htjeli zapisati u memoriju i naknadno obrađivati. U većini slučajeva takvi su podaci bili određeni jednim indeksom. Npr. ako smo imali N prirodnih brojeva koji opisuju visinu svakog pojedinog učenika u razredu, tada smo te podatke spremili u listu tako da je npr. visina prvog učenika iz e-Dnevnika bila zapisana na prvu poziciju u listi i imala je indeks nula.

Lista lista

Neki podaci s kojima radimo često su jednoznačno određeni dvama indeksima. Takve su na primjer točke u koordinatnoj ravnini koje su zadane dvjema koordinatama, x i y . Takve podatke u Pythonu možemo spremati u memoriju listom lista. Lista lista nije posebna struktura podataka, već uobičajena lista čiji su elementi liste, tj. lista u kojoj su ugniježđene liste. Pokažimo na jednom primjeru kada se i kako možemo koristiti takvom strukturom.

(0,0)	Gutiven	Mirca	Stupelar		Postira				
4	8		7	4	15	3	0	0	
	Dobovišća			Škrp Dol		Pučišća		Povlja	
8	15		1	5	(1,4)	Pražnica	2	Novo Selo	3
0					Brač	Gornji Humac		Selca	Sumertin
					Vidova Gora		0		(2,7)
					Bol	3			5

Primjer 1.

Ivica je na tavanu stare obiteljske kuće na Braču pronašao mapu s blagom. Na toj mapi Brač je bio podijeljen na 3×8 kvadratiča i u svakom od ta 24 kvadratiča pisao je jedan cijeli broj. Taj je broj označavao broj zlatnika koji se mogu pronaći na tom konkretnom području. Npr., s mape se može pročitati

2. Lista listā

da u kvadratiču koji se nalazi u prvom retku i trećem stupcu možemo pronaći sedam zlatnika. Ivicu zanima koliko se ukupno zlatnika može prikupiti prekopavajući otok.

Opis algoritma:

Rješenje je očito i svodi se na zbrajanje 24 navedena broja. Ukupan broj zlatnika u ovom je konkretnom slučaju 147. Riješimo zadatok na informatički način. Koristeći se listama u listi, konkretnu situaciju možemo zapisati u memoriju na sljedeći način:

```
>>> L = [[4, 8, 7, 4, 15, 3, 0, 0], [8, 15, 1, 5, 32, 6, 2, 3], [0, 8, 12, 5, 3, 1, 0, 5]]
```

Uočimo da su vrijednosti iz prvog, drugog i trećeg retka mape s blagom elementi uobičajene liste, a te su liste (redci) elementi nove liste. Iz toga slijedi da ćemo broju zlatnika u kvadratiču koji se nalazi u prvom retku i prvom stupcu mape (četiri zlatnika) pristupiti tako da najprije navedemo indeks elementa u velikoj listi koji ga sadrži, a zatim indeks u listi koja je element velike liste. Pazi, elementi liste indeksiraju se počevši od nule. Na slici su neki kvadratiči označeni svojim indeksima radi lakšeg uočavanja načina indeksiranja.

```
>>> L = [[4, 8, 7, 4, 15, 3, 0, 0], [8, 15, 1, 5, 32, 6, 2, 3], [0, 8, 12, 5, 3, 1, 0, 5]]  
>>> redak_1 = L[0]  
>>> redak_1  
[4, 8, 7, 4, 15, 3, 0, 0]  
>>> redak_1[0]  
4
```

Naravno, cijeli ovaj postupak možemo skratiti i napisati:

```
>>> L[0][0]  
4
```

Odredimo sada broj koji je tražio Ivica. Za rješenje je dovoljno proći po svim elementima malih lista u velikoj listi i raditi uzastopno zbrajanje.

Prvi način:

```
>>> zbroj = 0  
>>> for redak in L:  
        for j in redak:  
            zbroj += j  
>>> zbroj  
147
```

Drugi način:

```
>>> zbroj = 0  
>>> for i in range(3):  
        for j in range(8):  
            zbroj += L[i][j]  
>>> zbroj  
147
```

Rad s listom lista

2. Lista listâ

Treći način:

```
>>> zbroj = 0
>>> for redak in L:
...     zbroj += sum(redak)
>>> zbroj
147
```

Zamislimo da se Ivica odlučio fokusirati samo na one kvadratiće u kojima se može pronaći deset ili više zlatnika. Ideja je da ponovno prođemo po svim cijelim brojevima, detektiramo one koji zadovoljavaju uvjet i ispišemo o kojim se kvadratićima radi. Podsjetimo se da broj elemenata u listi određujemo funkcijom *len()*.

Programski kod:

```
>>> zbroj = 0
>>> for i in range(len(L)):
...     for j in range(len(L[i])):
...         if L[i][j] >= 10:
...             print("Kvadratić u", i + 1, "retku i", j + 1, "stupcu")
Kvadratić u 1 retku i 5 stupcu
Kvadratić u 2 retku i 2 stupcu
Kvadratić u 2 retku i 5 stupcu
Kvadratić u 3 retku i 3 stupcu
```

Kako su elementi u listama indeksirani indeksima počevši od nule, morali smo indeksu pribrojiti jedan kako bismo dobili stvarnu poziciju na mapi s blagom.

Međutim, ulazni podaci rijetko su kada unaprijed zadani - najčešće se učitavaju sa standardnog ulaza. U većini slučajeva koje ćemo ovdje spominjati ulazni podaci bit će organizirani u tabličnom obliku s prepoznatljivom strukturu od *r* redaka i *s* stupaca. Pokažimo tri načina na koje ćemo moći tako zadane podatke učitavati u listu lista.

Prvi način:

```
r = int(input())                      # broj redaka
s = int(input())                        # broj stupaca
tablica = []                            # u početku je tablica prazna lista
for i in range(r):                     # onoliko puta koliko ima redaka
    tablica.append([])                  # dodaj praznu listu (za redak) u listu tablica
    red = map(int, input().split())    # učitaj sve elemente i-tog retka s ulaza
    for element in red:              # svaki element učitanog retka dodaj u podlistu
        tablica[-1].append(element)
print(tablica)
```

Npr:

```
>>>
2
5
2 5 6 7 5
5 4 1 2 8
[[2, 5, 6, 7, 5], [5, 4, 1, 2, 8]]
```

Učitavanje liste
lista

2. Lista listâ

Drugi način:

```
r = int(input())
s = int(input())
tablica = []
for i in range(r):
    redak = list(map(int, input().split()))
    tablica.append(redak)
print(tablica)
```

Npr:

```
3
4
2 3 4 5
7 6 5 4
8 7 6 5
[[2, 3, 4, 5], [7, 6, 5, 4], [8, 7, 6, 5]]
```

U rješenjima zadatka koristit ćemo se drugim načinom učitavanja tablice podataka.

 Prilikom učitavanja jednog retka nužno je koristiti se funkcijom *list*. U protivnom će nam redak biti iterabilni objekt, a ne konkretna lista.

```
>>> print(list(map(int, input().split())))
3 4 5 6 7
[3, 4, 5, 6, 7]
>>> print(map(int, input().split()))
3 4 5 6 7
<map object at 0x02EA0870>
```

Za kraj ovog dijela objedinimo upis i ispis tablice podataka s *r* redaka i *s* stupaca.

```
r, s = map(int, input().split())
tablica = []
for i in range(r):
    redak = list(map(int, input().split()))
    tablica.append(redak)

for i in tablica:                      # elementima pristupamo izravno
    for j in i:
        print(j, end = ' ')
    print()

for i in range(r):                      # elementima pristupamo preko indeksa
    for j in range(s):
        print(tablica[i][j], end = ' ')
    print()
```

Učitavanje i
ispisanje liste lista

Riješimo sada nekoliko kratkih zadatka kroz koje ćemo se upoznati s raznim načinima rada s listom lista.

2. Lista listâ

Primjer 2.

Napiši program koji za zadatu tablicu (s r redaka i s stupaca) prirodnih brojeva određuje i ispisuje zbroj svih brojeva koji se nalaze u parnim stupcima.

Opis algoritma:

Najprije ćemo u listu lista učitati tablicu prirodnih brojeva pa zatim obići sve elemente i zbrojiti samo one koji se u tablici nalaze u parnom stupcu. Oprez! U zadatku se traže parni stupci tablice (drugi, četvrti...). U listi lista kojom ćemo se koristiti to će biti elementi s neparnim indeksom unutar lista koje predstavljaju retke.

Npr.

Tablica	Lista listâ
2 3 4 5 8	L = [[2, 3, 4, 5, 8],[1, 2, 6, 7, 3],[3, 7, 6, 8, 1]]
1 2 6 7 3	
3 7 6 8 1	

Programski kod:

```
r , s = map(int, input().split())
tablica = []
for i in range(r):
    redak = list(map(int, input().split()))
    tablica.append(redak)

zbroj = 0
for i in range(r):
    for j in range(s):
        if j % 2 == 1:
            zbroj += tablica[i][j]
print(zbroj)
>>>
4
2 3 4 5
6 5 4 3
1 2 9 0
2 1 5 7
# zbrajanje je obavljeno na sljedeći način: 3+5+5+3+2+0+1+7
```

Primjer 3.

Napiši program koji za zadatu tablicu (s r redaka i s stupaca) različitih prirodnih brojeva određuje i ispisuje maksimalnu vrijednost u tablici i zatim redak u kojem se ona nalazi.

Opis algoritma:

Rješenje zadatka svodi se na određivanje najveće od najvećih vrijednosti svakog retka tablice. U rješenju ćemo iskoristiti poznati algoritam za traženje maksimuma, a maksimum unutar retka određivat ćemo funkcijom *max*.

2. Lista listā

Programski kod:

```
# učitamo zadanu tablicu
najveci = 0
for redak in tablica:
    if max(redak) > najveci:
        najveci = max(redak)
print(najveci)
```

Maksimum

Na žalost, to elegantno rješenje ne omogućava određivanje indeksa retka u kojem se nalazi najveća vrijednost. Napišimo novi programski kod koji će riješiti i taj dio zadatka.

Programski kod:

```
redak = najveci = 0
for i in range(r):
    if max(tablica[i]) > najveci:
        najveci = max(tablica[i])
        redak = i
print(najveci, redak + 1)
```

Uoči da je prilikom ispisa redak povećan za jedan. To je nužno zbog poznate činjenice o indeksiranju elemenata počevši od nule.

Primjer 4.

Napiši program koji za zadanu tablicu (s r redaka i s stupaca) različitih prirodnih brojeva određuje i ispisuje stupac u kojem se nalazi maksimalna vrijednost.

Opis algoritma:

U ovom slučaju nećemo se moći koristiti funkcijom *max* jer vrijednosti unutar jednog stupca tablice u listi lista nisu grupirane unutar jedne liste kao što je to slučaj s vrijednostima iz retka tablice. Za rješenje trebamo zamisliti da se spuštamo po stupcima i analiziramo svaki element posebno.

Programski kod:

```
# učitamo zadanu tablicu

stupac = najveci = 0
for j in range(s): # tradicionalno se stupac označava s j
    for i in range(r):
        if tablica[i][j] > najveci:
            najveci = tablica[i][j]
            stupac = j
    print(najveci, stupac + 1)
```

Postoji li način da ipak iskoristimo rješenje iz Primjera 3.? Postoji. Dovoljno je zadanu tablicu "transponirati", tj. zamijeniti u njoj retke i stupce. Uvjeri se da sljedeći kod uistinu transponira tablicu.

2. Lista listā

Programski kod:

transponiranje

```
# učitamo zadanu tablicu  
  
transponirana = []  
for i in range(s):  
    redak = [0] * r  
    transponirana.append(redak)  
for i in range(r):  
    for j in range(s):  
        transponirana[j][i] = tablica[i][j]
```



Na prvi pogled čini se da transponiranu tablicu možemo inicijalizirati na sljedeći način:

```
>>> transponirana = [[0] * r] *s  
>>> transponirana  
[[0, 0, 0...], [0, 0, 0...], ....]
```

Međutim, taj način nije dobar. Zašto? Provjeri!

2.2. Matrice u matematici

Dvodimenzionalni nizovi

Dvodimenzionalni niz, odnosno matricu, možemo predočiti kao tablicu brojeva. Pritom razlikujemo redove i stupce matrice. Svaki element matrice nalazi se na presjeku određenog reda i stupca. Indeksi u programskom jeziku Python kreću od 0: element matrice koji se nalazi u prvom redu i prvom stupcu ima indeks [0][0].

Prvi indeks označava red, a drugi indeks označava stupac. Svi elementi istog reda imaju jednak prvi, dok svi elementi istog stupca imaju jednak drugi indeks.

Neka je A dvodimenzionalni niz koji ima 4 reda i 3 stupca:

	1. stupac	2. stupac	3. stupac
1. red			
2. red			
3. red			
4. red		656	

Broj 656 nalazi se u 4.

edu i 2. stupcu danog polja A. A[3,1] = 656, gdje A označava ime niza, 3 indeks reda, 1 indeks stupca, a 656 vrijednost.

Matrica

U matematici se dvodimenzionalni nizovi nazivaju **matrice**. Matrica je tipa (ili reda) $m \times n$ ako ima m redova i n stupaca.

Općeniti prikaz matrice tipa $m \times n$ u programskom jeziku Python izgleda ovako:

$$A = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n-1} \\ a_{10} & a_{11} & \dots & a_{1n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-10} & a_{n-11} & \dots & a_{n-1n-1} \end{bmatrix}$$

2. Lista listā

Prvi indeks označava broj reda, a drugi broj stupca. Kvadratna matrica ima isti broj redaka i stupaca i reda je n . Glavna dijagonalna kvadratne matrice sadržava elemente $(a_{00}, a_{11}, \dots, a_{n-1n-1})$ kojima je indeks reda jednak indeksu stupca. Sporedna dijagonalna sadržava elemente $(a_{0n-1}, a_{1n-2}, \dots, a_{n-10})$ kojima je zbroj indeksa reda i stupca jednak redu matrice umanjenom za 1.

Za učitavanje i ispis elemenata dvodimenzionalnog niza potrebne su dvije programske petlje, jedna za red (vanjska), a druga (unutrašnja) za stupac. Elemente matrice tipkovnicom uglavnom unosimo po redovima (najprije elementi prvog reda, pa elementi drugog reda itd.).

U programskom jeziku Python matrica je lista listā.

Primjer 1.

U interaktivnom okruženju:

```
>>> a = [[1,2,3],[2,3,1]]  
>>> a[1]  
>>> a[0][2]
```

```
[2,3,1]  
3
```

Primjer 2.

```
matrica=[ ]  
matrica.append([1,2,3])  
matrica.append([4,5,6])  
matrica.append([7,8,9])  
print("Sadržaj cijele liste")  
print(matrica)  
print("Ispisuje redak s indeksom 0")  
print(matrica[0])  
print("Ispis red po red")  
for red in matrica:  
    print(red)  
  
print("Cisti ispis svih elemenata - bez zagrada, zareza i sl.")  
  
for red in matrica:  
    for element in red:  
        print(element)  
    print()  
  
print("Zadnji element u zadnjem retku:", matrica [2][2])
```

Sadržaj cijele liste
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Ispisuje redak s indeksom 0
[1, 2, 3]
Ispis red po red
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
Cisti ispis svih elemenata - bez zagrada, zareza i sl.
1
2
3
4
5
6
7
8
9
Zadnji element u zadnjem retku: 9



Detaljnije o radu s matricama pročitajte u dokumentu [3g2matrice.pdf](#).

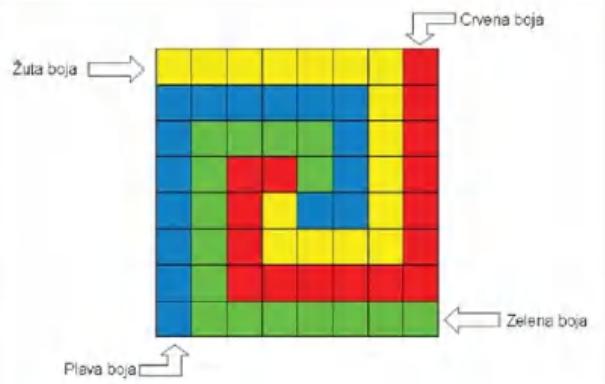
2. Lista listâ

Od problema do rješenja

Problem 1.

Klara je šahovsku ploču (64 polja, osam redaka, osam stupaca) obojila točno kao na slici.

Redci/stupci su označeni brojevima od jedan do osam, odozgo prema dolje/slijeva nadesno. Napišite program koji za dano polje ove ploče, ispisuje koje je boje to polje.



Opis algoritma

Zadatak se može riješiti sa 64 upita, što je naporno. Lakše ga je riješiti tako da definiramo matricu znakova (lista stringova) koja opisuje ploču, pri čemu različite boje različito označavamo, primjerice znakovima „.”, „#”, „o” i „|”, koji označavaju redom žutu, crvenu, zelenu i plavu boju. Nakon toga rješenje je jednostavno ispisati boju koja odgovara s-tom znaku r-tog stringa, točnije, znaku $m[r-1][s-1]$, jer u matrici retke i stupce brojimo od nule.

Programski kod:

Python omogućava „liste“ u kojima **ključ** po kojemu nalazimo element nije nužno neki indeks od 0 do $N - 1$, nego je taj ključ **bilo koji broj, string ili nešto treće**. Takva struktura zove se **rječnik (dictionary)** i može se definirati kao slijed parova **kljuc:vrijednost** unutar višičastih zagrada `{ }`. Element možemo dodavati, čitati ili mijenjati pišući `ime_rječnika[kljuc]`.

```
r = int(input())
s = int(input())
a = ['.....#',
     '| |||||. #',
     '|oooo|. #',
     '|o##o|. #',
     '|o#.||. #',
     '|o#....#',
     '|#####.#',
     '|ooooooo']

znak = a[r - 1][s - 1]
if znak == '.':
    print('ZUTA')
if znak == '#':
    print('CRVENA')
if znak == 'o':
    print('ZELENA')
if znak == '|':
    print('PLAVA')

boja_znaka = {
    '.': 'ZUTA', '#': 'CRVENA', 'o': 'ZELENA', '|': 'PLAVA'}
print(boja_znaka[znak])
```

Struktura rječnik nam u ovom slučaju može pojednostaviti ispis boje ovisno o znaku. Definirat ćemo „listu“ čiji su elementi riječi „ZUTA“, „CRVENA“, „ZELENA“ i „PLAVA“, a „indeksi“ znakovi „.“, „#“, „o“ i „|“, tako da možemo jednostavno ispisati `lista[znak]`.

2. Lista listā

Problem 2.

Za danu matricu ispunjenu vrijednostima 0 i 1 odredite najmanji broj elemenata koji treba promijeniti tako da matrica bude „šahovnica”, tj. da susjedna polja u retku ili stupcu uvijek budu različite vrijednosti.

Opis algoritma

Nakon što učitamo zadatu matricu, promatramo dva slučaja:

- A. Prvo polje šahovnice bit će 0.
- B. Prvo polje šahovnice bit će 1.

Ako znamo prvo polje šahovnice, možemo odrediti sva ostala polja, jer znamo da moraju naizmjence ići nule i jedinice. Tad uspoređujemo polje koje već piše u zadanoj matrici s poljem koje trebamo dobiti u šahovnici. Ako je trenutačni broj u polju jednak broju koji će biti dio šahovnice, ne moramo ga promijeniti, a inače moramo.

Možemo primijetiti da će u slučaju A vrijednosti 1 biti u poljima s neparnim zbrojem retka i stupca, a u slučaju B obrnuto.

Dakle, za svaki od slučajeva A i B brojimo koliko polja treba promijeniti odaberemo li taj slučaj. Na kraju ispisujemo povoljniji (manji) od dvaju dobivenih brojeva. Da ne bismo pisali gotovo isti kod za oba slučaja A i B, koristit ćemo se funkcijom koja prima vrijednost početnog polja šahovnice (0 ili 1) i vraća broj elemenata koje u tom slučaju valja promijeniti.

Programski kod

```
r, s = map(int, input().split())
m = []
for i in range(r):
    redak = list(map(int, input().split()))
    m.append(redak)

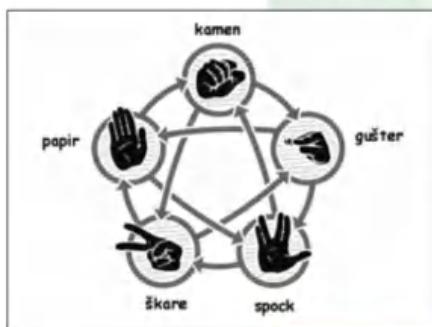
def broj_promjena(pocetno_polje):
    promjene = 0
    for i in range(r):
        for j in range(s):
            if pocetno_polje == 0:
                polje_sahovnice = (i + j) % 2
            else:
                polje_sahovnice = (1 + i + j) % 2
            if polje_sahovnice != m[i][j]:
                promjene += 1
    return promjene

a = broj_promjena(0)
b = broj_promjena(1)
print(min(a, b))
```

2. Lista lîstâ

Problem 3.

U humorističnoj seriji Teorija velikog praska (engl. *The Big Bang Theory*) opisana je jedna društvena igra koju igraju prijatelji Leonard, Sheldon, Penny, Howard i Rajesh. To je igra kamen-papir-škare-gušter-Spock (engl. *Rock-Paper-Scissors-Lizard-Spock*).



Pravila igre jednostavna su. Kada je igraju dva igrača, u jednoj rundi svaki odabere jednu od pet mogućnosti (kamen, papir, škare, gušter, Spock) i pokaže je rukom koristeći se unaprijed dogovorenim znakom. Pobijedio je onaj koji je odabrao i pokazao jaču mogućnost. Pri tome vrijede sljedeći odnosi (jača je mogućnost podebljana): **škare** režu papir, **papir** prekriva kamen, **kamen** gnječi guštera, **gušter** truje Spocka, **Spock** trga škare, **škare** režu glavu gušteru, **gušter** jede papir, **papir** pobija Spocka, **Spock** pretvara kamen u prah i **kamen** drobi škare. Ako su oba igrača odabrala istu mogućnost, onda je neriješeno.

Ako više igrača igra igru, onda se kao ishod jedne runde **za svakog igrača** provjeri **od koliko je** drugih odabira njegov odabir **bio jači** te se tom igraču pridruži toliko bodova u toj rundi.

Napiši program koji će na temelju zadanog odabira za svakog od petoro prijatelja odrediti i ispisati koliko je svaki igrač dobio bodova u zadanoj rundi.

Ulazni podaci

U prvom retku nalazi se pet prirodnih brojeva **O** ($1 \leq O \leq 5$), oznake odabira redom za Leonarda, Sheldona, Penny, Howarda i Rajesha. Pri tome vrijedi da je s '1' označen kamen, s '2' papir, s '3' škare, s '4' gušter i s '5' Spock.

Izlazni podaci

Bazinga!

U prvom retku opisan je ishod igre. Pri tome je prvi broj u retku broj osvojenih bodova Leonarda, drugi Sheldona, treći Penny, četvrti Howarda i peti Rajesha.

Primjeri testnih podataka

ulaz	ulaz	ulaz
1 3 2 4 5	2 3 2 4 5	3 4 4 5 1
izlaz	izlaz	izlaz
2 2 2 2 2	1 3 1 3 1	2 1 1 2 3

Opis algoritma:

Na početku kreiramo tablicu *stanje* s pet redaka i pet stupaca koja će imati unaprijed definirane vrijednosti. Ona će na poziciji $[i,j]$ imati upisanu vrijednost jedan ako je i -ti odabir jači od j -tog odabira ili vrijednost nula ako nije. Npr., na poziciji [1,3] upisat ćemo jedinicu jer je kamen (oznaka '1') jači od škara (oznaka '3'), a na poziciju [1,5] upisat ćemo nulu jer je kamen slabiji od Spocka. Uočimo da su vrijednosti na glavnoj dijagonali uvijek jednake nuli. Naravno, u listi lista indeksiranje počinje od nule i na to trebamo paziti prilikom pisanja koda.

Lista lîstâ kao pomoćna struktura

2. Lista listā

U posebnu listu *odabir* učitat ćemo odabrane vrijednosti s ulaza, dok ćemo u listi *igrac* pamti koliko je koji igrač dobio bodova. Na temelju tablice stanje za svakog igrača treba odrediti kako je prošao u dvoboju sa svakim od preostalih četvero igrača. To ćemo postići s dvjema ugniježđenim petljama, od kojih će prva pratiti igrača kojeg trenutno promatramo, dok će druga prolaziti po svim ostalim igračima.

Programski kod:

```
stanje = [[0, 0, 1, 1, 0],
          [1, 0, 0, 0, 1],
          [0, 1, 0, 1, 0],
          [0, 1, 0, 0, 1],
          [1, 0, 1, 0, 0]]

odabir = list(map(int, input().split()))

igrac = [0] * 5
for i in range(5):
    for j in range(5):
        igrac[i] += stanje[odabir[i]-1][odabir[j]-1]

print(igrac)
```

Problem 4.

Izbornik Ćiro dugo je razmišljaо kako na najbolji način pripremiti naše reprezentativce za slavni susret s reprezentacijom Francuske.

Jedna od važnijih stvari tijekom tih priprema bila je i analiza prethodno odigranih utakmica tih dviju reprezentacija. Izbornika zanima kojim su **rezultatom najčešće završavale te utakmice**. Rezultat utakmice zadan je brojem postignutih golova domaćina i brojem postignutih golova gosta na toj utakmici. Ako se više različitih rezultata pojavljuje isti najveći broj puta, tada se uzima onaj u kojem je ukupno postignuto više golova. Ako ni tada nije moguće jedinstveno odrediti najčešći, uzima se onaj u kojem je gost postigao više golova.

Napiši program koji će za zadane rezultate posljednjih **N** odigranih utakmica Hrvatske i Francuske odrediti i ispisati najčešći rezultat kojim su te utakmice završile.

Uzni podaci

U prvom retku nalazi se prirodni broj **N** ($1 \leq N \leq 10$). U sljedećih **N** redaka nalaze se po dva prirodna broja odvojena razmakom, **GDi** ($0 \leq GDi \leq 6$, $i=1..N$) i **GGi** ($0 \leq GGi \leq 6$, $i=1..N$) koji predstavljaju rezultat i-te odigrane utakmice.

Izazni podaci

U jednom retku treba ispisati dva broja odvojena razmakom, traženi rezultat iz teksta zadataka.



2. Lista listā

Opis algoritma:

U zadatku se definira da će domaćin i gost moći postići najviše šest golova. Zbog toga postoji konačan broj rezultata kojima utakmica može završiti (počevši od 0:0 do 6:6). Ideja je prebrojiti koliko se puta koji rezultat pojavio i zatim odrediti koji se pojavio najviše puta. Za to ćemo definirati tablicu *rezultati* sa sedam redaka i sedam stupaca i u i-ti redak i j-ti stupac upisati broj pojavljivanja rezultata i:j.

Primjeri testnih podataka

ulaz	ulaz	ulaz
5	5	6
2 1	2 2	1 2
1 4	1 1	1 1
2 1	2 2	2 1
0 0	3 0	3 0
1 0	1 1	2 1
		1 2
izlaz	izlaz	izlaz
2 1	2 2	1 2

Programski kod:

Prvo trebamo kreirati tablicu *rezultati* i sve elemente te tablice postaviti na nulu. Za to ćemo se koristiti jednim od nekoliko načina koji postoje.

```
rezultati = []
for i in range(7):
    rezultati += [[0] * 7]
```

Istovremeno s učitavanjem ulaznih podataka (rezultata utakmica) radimo korekciju vrijednosti odgovarajućeg elementa u tablici.

```
N = int(input())
for i in range(N):
    a, b = map(int, input().split())
    rezultati[a][b] += 1
```

Traženje pripadajućeg najčešće pojavljivanog rezultata malo se komplicira zbog dodatnih uvjeta koji su postavljeni.

```
max = x = y = 0
for i in range(7):
    for j in range(7):
        if rezultati[i][j] > max:
            max = rezultati[i][j]
            x = i
            y = j
        else:
            if rezultati[i][j] == max:
                if i + j > x + y:
                    max = rezultati[i][j]
```

(nastavak na idućoj stranici)

2. Lista lîstâ

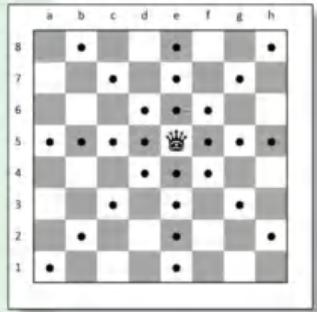
```
x = i  
y = j  
elif i + j == x + y:  
    if j > y:  
        max = rezultati[i][j]  
        x = i  
        y = j  
  
print(x,y)
```

Problem 5.

Promotrimo igru za dva igrača koja se igra na šahovskoj ploči. Jedan od igrača na šahovsku ploču postavi **tri figurice kraljice**, a onaj drugi mora prebrojiti **koliko je polja šahovske ploče napadnuto** uzimajući u obzir sve tri kraljice.

Šahovska ploča ima **osam redaka i osam stupaca**. Redci su označeni brojevima od 1 do 8, a stupci slovima od a do h kao na slici. Figurica kraljice napada sva polja koja se nalaze u **istom retku, stupcu ili na jednoj od dviju dijagonala** na kojima se ona nalazi (na slici su crnim kružićima označena polja šahovske ploče koja napada kraljica postavljena na polje u retku 5 i stupcu e). Pritom kraljica ne napada samo polje na kojem se ona nalazi (no ako se kraljica nalazi na polju napadnutom od druge kraljice, to polje smatramo napadnutim). Na svakom polju šahovske ploče može biti postavljena najviše jedna kraljica.

Za zadane pozicije triju kraljica odredi **koliko je polja šahovske ploče napadnuto**.



Izlazni podaci

U trima redcima nalaze se po dva znaka neodvojena razmakom: **S** ('a' ≤ S ≤ 'h') i **R** (1 ≤ R ≤ 8), oznake stupca i oznake retka polja šahovske ploče na kojem se nalazi prva, druga i treća kraljica.

Izlazni podaci

U jedini redak ispiši traženi broj napadnutih polja iz teksta zadatka.

Primjeri testnih podataka

ulaz	ulaz	ulaz
b3	a2	g7
f6	f3	c1
f1	e8	g1
izlaz	izlaz	izlaz
49	48	42

Opis algoritma:

Prirodno je da se u rješenju koristimo tablicom, tj. listom lista. Promotrimo tablicu dimenzija šahovske ploče. Ideja je da svako polje u tablici postavimo na vrijednost jedan ako ga napada neka od kraljica. Opišimo sada kako ćemo kreirati takvu tablicu.

Prije svega postavimo sve elemente tablice na nulu (čime označavamo da polja nisu napadnuta). Za svaku od triju kraljica potrebno je proći u svih osmih osnovnih smjerova krenuvši od polja susjednog polju na kojem se nalazi sama kraljica pa sve do ruba šahovske

2. Lista listā

Kretanje u svim smjerovima

ploče. To jednostavno možemo implementirati koristeći se pomoćnim listama za pomak retka i stupca dx i dy . Kretanje po tablici u nekom od smjerova određeno je pomakom po retku i stupcu. Npr. ako se od trenutne pozicije želimo pomaknuti za jedno mjesto prema gore (ako je to moguće), morat ćemo ostati u istom stupcu, ali će se koordinata retka smanjiti za jedan. Na isti način možemo odrediti i vrijednosti pomaka za ostale smjerove.

(x-1,y-1)	(x-1,y)	(x-1,y+1)
(x,y-1)	(x,y)	(x,y+1)
(x+1,y-1)	(x+1,y)	(x+1,y+1)

Za svaku kraljicu prođimo jednom petljom po osam smjerova, a drugom (ugniježđenom) petljom prolazimo po poljima u tom smjeru sve dok ne dođemo do ruba ploče. Dok prolazimo po poljima, postavljamo ih na vrijednost jedan (što označava da je polje napadnuto).

Nakon što smo na gore opisan način konstruirali traženu tablicu, samo prođemo po njoj i izbrojimo napadnuta polja, tj. polja koja su postavljena na jedan. Ako uočimo da su vrijednosti u tablici nula i jedan, tada je dovoljno samo zbrojiti sve vrijednosti.

Mali je problem kako iz zadanih oznaka polja šahovske ploče odrediti konkretan redak i stupac u kojem se nalazi kraljica. Ali i to možemo lako riješiti.

Programski kod:

```
dx = [-1, -1, 0, 1, 1, 1, 0, -1]
dy = [0, 1, 1, 1, 0, -1, -1, -1]

# kreiramo tablicu popunjenu nulama na jedan od mogućih načina
sahovnica = []
for i in range(8):
    redak = [0] * 8
    sahovnica.append(redak)

for i in range(3):          # za svaku od kraljica
    polje = input()
    redak = ord(polje[0]) - 96 - 1
    stupac = int(polje[1]) - 1

    for smjer in range(8):      # za svaki od osam smjerova
        udaljenost = 1
        while 1:
            novi_r = redak + udaljenost * dx[smjer]
            novi_s = stupac + udaljenost * dy[smjer]

            if novi_r < 0 or novi_r > 7 or novi_s < 0 or novi_s > 7: break
            else:
                sahovnica[novi_r][novi_s] = 1
                udaljenost += 1
```

(nastavak na idućoj stranici)

2. Lista lîstâ

```
sahovnica[novi_r][novi_s] = 1  
udaljenost += 1  
  
broj_napadnutih = 0  
for i in range(8):  
    broj_napadnutih += sum(sahovnica[i])  
  
print(broj_napadnutih)
```

Problem 6.

Bio jednom jedan šou koji je tražio neku zvjezdu. Nakon nekoliko lokalnih audicija suci šoua odabrali su **pet kandidata** za završni krug. Kandidate su označili slovima **A, B, C, D i E**.

U završnom krugu svaki je kandidat ocjenjivan prema **pet kriterija** pa je tako dobio **pet ocjena**. Ocjene su između **jedan i devet**. Nakon toga, ocjene se moraju **pretvoriti** u bodove prema sljedećem pravilu: natjecatelj dobije bod **svaki put** kada je njegova ocjena prema nekom kriteriju **jedina takva** među ocjenama drugih kandidata prema tom **istom kriteriju** ocjenjivanja.

Na kraju šoua suci su postavili tri pitanja. Odgovori na njih.

1. Koliko je bodova dobio natjecatelj s oznakom **A**?
2. Koji je kandidat dobio **najviše** bodova i **koliko** je to bilo bodova? Ako je više kandidata imalo isti broj bodova, tada treba ispisati onog čija je oznaka **prije** po abecedi.
3. Kako izgleda **poredak** kandidata od onog s najvećim brojem bodova do onoga s najmanjim? Ako više kandidata ima isti broj bodova, prije treba ispisati onoga čija je oznaka **prije** po abecedi.

Ulazni podaci

U pet redaka nalazi se po pet prirodnih brojeva **O** ($1 \leq O \leq 9$) odvojenih razmakom. U redcima se nalaze ocjene za kandidate s oznakama **A, B, C, D i E**, tim redom. U stupcima se redom nalaze ocjene prema prvom, drugom, trećem, četvrtom i petom kriteriju.

Izlazni podaci

U prvi redak treba ispisati odgovor na prvo pitanje, u drugi na drugo i u treći redak odgovor na treće pitanje.

Primjeri testnih podataka

Ulaz	Izlaz	Objašnjenje
5 4 3 6 8	3	A: 3 boda, po kriterijima $1+0+0+1+1$
3 4 5 2 4	E 4	B: 1 bod, po kriterijima $0+0+0+0+1$
1 2 3 2 6	EADBC	C: 1 bod, po kriterijima $1+0+0+0+0$
3 7 5 4 6		D: 2 boda, po kriterijima $0+1+0+1+0$
4 2 9 3 2		E: 4 boda, po kriterijima $1+0+1+1+1$

2. Lista listâ

Opis algoritma:

Zadatak se sastoji od rješavanja triju manjih odvojenih zadataka. Koristeći se tablicom prirodnih brojeva s pet redaka i pet stupaca najlakše ćemo sistematizirati dobivene podatke. U tablicu *ocjene* zapisat ćemo ocjene koje je pojedini kandidat (redak) dobio prema odgovarajućem kriteriju (stupac). Zatim ćemo u tablicu *bodovi* zapisati vrijednosti nula/jedan, ovisno o tome je li igrač (opisan tim retkom) dobio bod prema odgovarajućem kriteriju (opisanom tim stupcem).



Uoči da se lista listâ *T* može učitati i na sljedeći način:

```
T = [list(input())  
     for i in range(5)]
```

Na žalost, takav način ima nedostatak. Evo jednog primjera:

```
>>> T =  
[list(input()) for i  
in range(2)]  
12345  
98765  
>>> T  
[['1', '2', '3',  
'4', '5'], ['9', '8',  
'7', '6', '5']]
```

Vrijednost u tablici *bodovi* na poziciji [i,j] dobije se tako da se pronađe koliko je elemenata u j-tom stupcu tablice *ocjene* (osim onog koji odgovara i-tom retku) jednakom elementu na poziciji [i,j]. Ako je taj broj jednak nuli, tada je pripadajuća vrijednost 1, inače je 0.

Rješenje prvog problema zbroj je elemenata u prvom retku tablice *bodovi*. Odgovor na drugo pitanje oznaka je retka u kojem se nalazi najveći zbroj vrijednosti po redcima (naravno, odgovor je i ta vrijednost). Odgovor na treće pitanje zahtijeva malo više posla, a svodi se na sortiranje redaka ovisno o zbroju vrijednosti u njima. Ali, treba biti oprezan u situaciji kada su zbrojevi bodova dvoje kandidata prilikom sortiranja jednaki. Tada treba napraviti dodatnu provjeru jesu li oznake kandidata u dobrom poretku jer zadatak zahtijeva da "ako više kandidata ima isti broj bodova, prije treba ispisati onoga čija je oznaka prije po abecedi".

Prije nego krenemo dalje, napisat ćemo funkciju koja nam omogućava učitavanje tablice brojeva. Taj način opisan je u uvodnom dijelu liste lista. Tom istom funkcijom koristit ćemo se i u nastavku poglavljia bez dodatnog objašnjavanja.

```
def ucitaj_tablicu_brojeva(r):  
    tablica = []  
    for i in range(r):  
        redak = list(map(int, input().split()))  
        tablica.append(redak)  
    return tablica
```

Programski kod:

```
ocjene = ucitaj_tablicu_brojeva(5)  
  
bodovi = []  
for i in range(5):  
    redak = [0] * 5  
    bodovi.append(redak)  
  
for i in range(5):  
    for j in range(5):  
        br = 0  
        for k in range(5):  
            if k != i and ocjene[k][j] == ocjene[i][j]:  
                br += 1  
        redak[j] = br
```

(nastavak na idućoj stranici)

2. Lista listā

```
if br == 0:
    bodovi[i][j] = 1
# prvo pitanje
print(sum(bodovi[0]))

#drugo pitanje
maks = tko = 0
for i in range(5):
    if sum(bodovi[i]) > maks:
        maks = sum(bodovi[i])
        tko = i
print(chr(tko + 65), maks)

#treće pitanje
poredak = [0,1,2,3,4]

for i in range(5):
    for j in range(i+1, 5):
        if sum(bodovi[i]) < sum(bodovi[j]):
            bodovi[i], bodovi[j] = bodovi[j], bodovi[i]
            poredak[i], poredak[j] = poredak[j], poredak[i]
        elif sum(bodovi[i]) == sum(bodovi[j]) and poredak[i] > poredak[j]:
            bodovi[i], bodovi[j] = bodovi[j], bodovi[i]
            poredak[i], poredak[j] = poredak[j], poredak[i]
poredak[i]

for i in poredak:
    print(chr(i + 65), end = ' '))
```

Problem 7.

Zamislimo prostoriju pravokutnog oblika čiji pod možemo podijeliti na **kvadratne ploče jednake veličine**. Prostorija je prepuna zamki koje se kriju ispod ploča, no srećom imamo **upute** koje pokazuju kako se možemo sigurno kretati prostorijom. Upute se sastoje od **niza znakova** koji mogu biti ‘S’, ‘I’, ‘J’ ili ‘Z’. Ti znakovi označavaju stranu svijeta prema kojoj se treba pomaknuti (‘S’ – sjever ili jedno mjesto iznad trenutnog mesta na kojem se nalazi, ‘I’ – istok ili jedno mjesto desno, ‘J’ – jug ili jedno mjesto dolje, ‘Z’ – zapad ili jedno mjesto lijevo). Uz upute za kretanje imamo i kartu prostorije koja se sastoji od znakova za koje se zna da imaju sljedeće značenje:

- **znak ‘.’** obična ploča;
- **veliko slovo ‘B’** ploča ispod koje se nalazi blago;
- **znamenka (0..9)** kada se nađemo na toj ploči trebamo **preskočiti onoliko idućih uputa** za kretanje koliki se broj nalazi na tom polju.

Nas sada zanima koliko blaga možemo iskopati na putu opisanom u uputama (jednom kad iskopamo blago ispod neke ploče, više ga ne možemo iskopati na istom mjestu). Na početku se nalazimo u sjeverozapadnom kutu prostorije, tj. na najgornjoj lijevoj kvadratnoj ploči.

2. Lista listā

Uzeti podaci

U prvom retku nalazi se string koji predstavlja upute za kretanje. U drugom retku nalaze se dva prirodna broja **N** i **M** ($1 \leq N, M \leq 8$), redom brojevi redaka i stupaca na koje je podijeljena prostorija. U svakom od idućih **N** redaka nalazi se po **M** znakova koji predstavljaju kartu prostorije.

Izlazni podaci

U jedinom retku treba ispisati koliko blaga možemo iskopati na putu opisanom u uputama.

Opis algoritma:

Na temelju znakova iz stringa *uputa* simulirat ćemo kretanje po tablici znako-

Primjeri testnih podataka

ulaz	ulaz	ulaz
IJI	JISSIJ	IJJIIJSIZSZJJ
3 3	3 3	5 5
...B..B
BBB	B2.	.2...
.B.	BBB	BB1B2
		...1B
		.B...

izlaz	izlaz	izlaz
2	2	4

va prostorija onako kako je opisano u zadatku. Za to će nam biti potrebne četiri pomoćne varijable koje će nam služiti za pamćenje trenutnog retka (*r*) i stupca (*s*) u kojem se nalazimo, trenutnog znaka iz stringa (*x*) i količinu blaga (*blago*) koju smo do tada sakupili. Kretanje ćemo simulirati sve dok ne dođemo do kraja stringa s uputama. Na temelju trenutnog znaka iz uputa određujemo smjer kretanja korigirajući trenutnu vrijednost retka i stupca u kojem se nalazimo. Nakon toga preostaje nam provjeriti nalazi li se na trenutnom polju blago. Ako se nalazi, brojač kojim pamtimo koliko smo blaga do sad skupili povećamo za jedan, oznaku da je na tom mjestu blago zamjenimo znakom za običnu ploču. Naposljeku, izvan petlje ispišemo koliko smo sakupili blaga, što je rješenje zadatka.

Prije nego krenemo na kod, napisat ćemo funkciju koja nam omogućava učitavanje tablice znakova. Taj način opisan je u uvodnom dijelu liste lista. Tom istom funkcijom koristit ćemo se i u nastavku poglavila bez dodatnog objašnjavanja.

```
def ucitaj_tablicu_znakova(r):
    tablica = []
    for i in range(r):
        tablica.append([])
        red = input()
        for element in red:
            tablica[-1].append(element)
    return tablica
```

Simulacija

2. Lista lîstâ

Programski kod:

```
upute = input()
N, M = map(int, input().split())
prostorija = ucitaj_tablicu_znakova(N)

r = s = x = blago = 0

while x < len(upute):
    if upute[x] == 'S': r = r - 1
    if upute[x] == 'I': s = s + 1
    if upute[x] == 'J': r = r + 1
    if upute[x] == 'Z': s = s - 1

    if prostorija[r][s] == 'B':
        blago += 1
        prostorija[r][s] = '.'

    if prostorija[r][s] >= '0' and prostorija[r][s] <= '9':
        x += ord(prostorija[r][s]) - 48 + 1
    else:
        x += 1

print(blago)
```

Problem 8.

Polje meduza kvadrat je duljine stranice **N** podijeljen na NxN jednakih kvadratića. U **M** kvadratića smještene su meduze. Cilj je uhvatiti sve meduze u Polju. Bob, poznati lovac meduza, na početku se nalazi u **R**-tom retku i **S**-tom stupcu odakle počinje skakati i lovitи meduze. Bob hvata meduze poštujući ova pravila:

1. odlazi do njemu **najbliže meduze**, tj. one do koje mu treba najmanji broj skokova po Polju. Jedan **skok** pomiće ga za jedan kvadratić gore, dolje, lijevo ili desno, a kretanje je uvijek optimalno;
2. ako su dvije ili više meduze jednako udaljene od mesta na kojem se nalazi, odabire se ona koja je **bliža zadnjem retku** (čiji je redak veći). Ako je više takvih, onda se odabire ona koja je **bliža zadnjem stupcu** (čiji je stupac veći);
3. nakon što ulovi meduzu, s njezina kvadratića **ponovo određuje** meduzu koja mu je **sada** najbliža i nju sljedeću hvata.

Napiši program koji će na temelju zadanih pozicija meduza u Polju odrediti na kojoj se poziciji nalazila zadnja uhvaćena meduza i ukupan broj skokova potrebnih da se pohvataju sve meduze poštujući zadana pravila.

Ulazni podaci

U prvom retku nalaze se prirodni brojevi **N** ($1 \leq N \leq 25$), **R** ($1 \leq R \leq N$), **S** ($1 \leq S \leq N$) i **M** ($0 \leq M \leq N^2$) iz teksta zadatka. U narednih **M** redaka nalaze se po dva prirodna

2. Lista listā

broja **A**, **B** ($1 \leq A, B \leq N$), koji predstavljaju poziciju i-te meduze, pri čemu je A redak, a B stupac u Polju.

Izlazni podaci

Na izlazu treba ispisati tri broja odvojena razmakom i to: redak i stupac u kojem se nalazila zadnja uhvaćena meduza i ukupan broj skokova potrebnih da se pohvataju sve meduze.

Primjeri testnih podataka

Ulaz	Izlaz	Objašnjenje
5 2 2 3 1 4 4 5 5 3	5 3 10	Polje meduza grafički se može prikazati kao na slici (X su meduze, . prazni dijelovi): ...X.X .X.. Krenuvši s pozicije (2,2), prvo u tri skoka uhvatimo meduzu na poziciji (1,4), zatim u četiri skoka meduzu na poziciji (4,5) i u tri skoka zadnju meduzu na poziciji (5,3).

Opis algoritma:

Prvo ćemo kreirati praznu tablicu kojom ćemo vizualizirati definirano Polje meduza. Na taj ćemo način lakše simulirati kretanje Boba po Polju. Najkraći put od polazišnog kvadratiča (R, S) do odredišta (i, j) jest da se od polazišta krećemo lijevo ili desno po retku sve dok ne dođemo do stupca u kojem se nalazi odredište. Onda samo trebamo ići ili prema gore ili dolje da dođemo do odredišta. S malo matematike zaključujemo da je ukupan broj koraka koji napravimo na tom putu .

Programski kod:

```
N, R, S, M = map(int, input().split())
R = R - 1
S = S - 1
def najblizi():
    global R, S
    min = 2 * N + 1
    for i in range(N):
        for j in range(N):
            if polje[i][j] == 'X':
                if abs(i - R) + abs(j - S) <= min:
                    min = abs(i - R) + abs(j - S)
                    redak = i
                    stupac = j
```

(nastavak na idućoj stranici)

2. Lista listā

```
polje[redak][stupac] = '.'
R = redak
S = stupac
return min

polje = []
for i in range(N):
    redak = ['.'] * N
    polje.append(redak)

for i in range(M):
    A, B = map(int, input().split())
    polje[A - 1][B - 1] = 'X'

ukupno = 0
for i in range(M):
    ukupno += najblizi()

print(R + 1, S + 1, ukupno)
```

Problem 9.

Cilj je ove igre pronaći zadani riječ koja je sakrivena negdje u tablici slova veličine **pet redaka i pet stupaca**. Slova u sakrivenoj riječi moraju biti povezana **jedno za drugim** po poljima u tablici bilo horizontalno, vertikalno ili dijagonalno. Svako slovo u tablici smije se iskoristiti **najviše jednom** prilikom pronalaženja zadane riječi. Prilikom traženja zadane riječi u tablici **nije** moguć višestruki odabir puta za traženje sljedećeg slova u riječi.

Program treba ispisati originalnu tablicu slova u kojoj će **ostati** slova iz zadane riječi (u poretku kako nastaje ta riječ), a sva ostala slova trebaju biti **zamijenjena** zvjezdicama. Riječ se uvijek može pronaći, a pronađeno je rješenje uvijek jedinstveno.

Ulagni podaci:

Pet redaka s po pet velikih slova engleske abecede, pri čemu prvi redak predstavlja slova iz prvog retka tablice slova itd. U šestom retku nalazi se riječ koju tražimo sastavljena od velikih slova engleske abecede maksimalne duljine 25 slova.

Izlazni podaci:

Pet redaka s po pet ili velikih slova engleske abecede ili zvjezdica.

Primjeri testnih podataka

RB	Ulaz	Izlaz	Objašnjenje
1.	OZCPM	*****	OZCPM
	BOAER	**A**	BOAER
	XRGBH	*R*B*	XRGBH
	FHDIAI	***A*	FHDIAI
	PLKJC	****C	PLKJC
	RABAC		

2. Lista listâ



Znamo da smo kopiju liste radili na sljedeći način: kopija = a[:]. Međutim, nije moguće na isti način kreirati kopiju tablice. Naime, kad bismo napisali tmp = T[:], tmp bi postala nova lista koja ima iste elemente kao i T. Ali T u sebi ima promjenjive elemente i čuva reference na njih. Na taj će se način dobiti tmp koji je nova lista s istim referencama na redove kao i lista T, a redovi će u memoriji ostati isti objekti jer nismo napravili nove redove, nego samo nove reference na njih. Jedno rješenje kreiranja kopije tablice iskorišteno je u rješenju zadatka.

Opis algoritma:

Zadatak ćemo rješiti simulacijom. Naime, pokušat ćemo u zadanoj tablici prepoznati riječ tako što ćemo se kretati od mesta na kojem se nalazi prvo slovo u traženoj riječi. Od trenutnog slova možemo se pomaknuti u osam smjerova i provjeriti nalazi li se u nekom od tih smjerova sljedeće slovo iz riječi. Uvjeti zadatka idu nam na ruku jer garantiraju postojanje samo jedne mogućnosti. Da nije tako, bilo bi nužno rješiti zadatak rekursivnim načinom. Obilazak svih smjerova lako ćemo rješiti uvođenjem dviju pomoćnih listi oblika: $dx = [-1, -1, 0, 1, 1, 1, 0, -1]$ i $dy = [0, 1, 1, 1, 0, -1, -1, -1]$. Funkcija *trazi()* omogućit će kretanje po tablici od zadane pozicije i usporedno izgrađivanje nove riječi. Kada tijekom kretanja po tablici izgradimo neku riječ koja je iste duljine kao zadana riječ, tada smo sigurni da smo je pronašli, završavamo obilazak i vraćamo tu riječ u glavni program.

Treba uočiti da je zbog odabrane implementacije u funkciji *trazi()* bilo nužno proglašiti varijable T i tmp kao globalne. U protivnom ne bismo mogli kreirati kopiju tmp i ne bismo mogli ispisati vrijednosti iz tmp izvan funkcije.

Programski kod:

```
dx = [-1, -1, 0, 1, 1, 1, 0, -1]
dy = [0, 1, 1, 1, 0, -1, -1, -1]

T = ucitaj_tablicu_znakova(5)

rijec = input()
tmp = []
def trazi(x, y, nova):
    global T, tmp
    slovo = 0; #nasao = 0

    tmp = []
    for redak in T: tmp.append(redak[:])
    tmp[x][y] = '*'

    while 1:
        slovo += 1
        for i in range(8):
            pr = x + dx[i]; ps = y + dy[i]
            if (pr >= 0) and (pr < 5) and (ps >= 0) and (ps < 5):
                if (tmp[pr][ps] == rijec[slovo]):
                    tmp[pr][ps] = '*'
                    nova += rijec[slovo]; x += dx[i]; y += dy[i]
                    break;
        if (slovo == len(rijec) - 1):
            return nova
```

(nastavak na idućoj stranici)

2. Lista lîstâ

```
for i in range(5):
    for j in range(5):
        if T[i][j] == rijec[0]:
            if trazi (i, j, rijec[0]) == rijec:
                for i in range(5):
                    for j in range(5):
                        if tmp[i][j] == '*':
                            print(T[i][j], end = ' ')
                        else:
                            print('*', end = ' ')
                print()
```

Problem 10.

Sljedeći problem na jednom će mjestu objediniti liste lista, liste, algoritam za traženje maksimuma i sortiranje.

Bacanje diska jedna je od atletskih disciplina. U toj disciplini sudjeluje **N** natjecatelja označenih startnim brojevima od **1** do **N**. Cilj je svakog od njih baciti disk što je moguće dalje. Natjecanje je podijeljeno u **šest serija**. U svakoj seriji bacač **jednom baca** svoj disk. Nakon triju serija, uzimajući u obzir samo najbolje bacanje svakog bacača, kreira se **pri-vremena ljestvica poretka**. Prvih **N/2 najboljih natjecatelja** s te liste nastavlja natjecanje u sljedećim trima serijama. Natjecatelji koji nastavljaju natjecanje uvijek se mogu odabrati samo na jedan način. Njihovi startni brojevi u nastavku se ne mijenjaju.

Nakon šest održanih serija vrijeme je za **podjelu medalja**. Zlatnu medalju osvaja bacač koji je u šest serija bacio disk najdalje od svih natjecatelja. Srebrna medalja dodjeljuje se nekom od preostalih bacača po istom kriteriju. Osvajača brončane medalje određujemo na isti način.

Međutim, prilikom određivanja osvajača medalje (bilo zlatne, srebrne ili brončane) nekad ne možemo odabrati samo jednog osvajača jer je više njih postiglo istu daljinu zbog koje zaslužuju tu medalju. Tada će sjajniju medalju dobiti onaj među njima koji ima **veće drugo** najdalje bacanje. Ako ni tada nije moguće odrediti jednog osvajača, uspoređuje se **njihovo treće**, pa **četvrto**, **peto** i na kraju **šesto** najbolje bacanje ako bude potrebno. Ako ni tada nije moguće odrediti samo jednog osvajača medalje, tada će **svi oni** dobiti istu medalju. Bez obzira koliko se podijeli istih medalja, sve tri vrste medalja obavezno će se podijeliti na kraju natjecanja.

Ulagni podaci:

U prvom retku nalazi se prirodni broj **N** ($6 \leq N \leq 20$, **N** je parni broj). U sljedećim trima redcima nalazi se po **N** prirodnih brojeva **Di** ($0 \leq Di \leq 100$), duljine bacanja u prvoj, drugoj i trećoj seriji izražene u metrima. Pri tome vrijedi da je i-ti broj u retku duljina do koje je disk bacio natjecatelj sa startnim brojem "i".

U sljedećim trima redcima nalazi se po **N/2** prirodnih brojeva **Di** ($0 \leq Di \leq 100$), duljine bacanja u četvrtoj, petoj i šestoj seriji izražene u metrima. Pri tome vrijedi da je prvi broj u retku duljina do koje je bacio natjecatelj s najmanjim startnim brojem i tako sve do zadnjeg broja u retku koji predstavlja bačenu duljinu natjecatelja s najvećim startnim brojem u nastavku natjecanja.

Najteži za kraj



2. Lista lîstâ

Izlazni podaci:

U prvom retku treba ispisati, odvojene razmakom, uzlazno poredane startne brojeve onih natjecatelja koji su nastavili natjecanje nakon triju serija. U drugom, trećem i četvrtom retku treba ispisati startni broj osvajača zlatne, srebrne i brončane medalje. Ako ima više osvajača neke od medalja, tada ih u tom retku treba ispisati sve, odvojene razmakom i uzlazno poredane.

Primjeri testnih podataka:

ulaz	ulaz	ulaz
6	8	10
4 6 9 2 7 5	1 4 5 3 4 2 4 1	1 3 4 6 5 5 1 6 1 3
5 4 3 3 3 6	5 4 1 2 7 8 6 3	2 4 7 6 5 5 3 9 0 2
1 8 7 0 1 5	9 4 8 3 2 2 5 5	1 0 6 0 8 4 2 9 1 2
5 4 3	7 4 5 4	5 1 4 6 9
8 7 0	2 2 3 1	6 8 3 4 1
2 3 4	6 2 1 5	5 4 9 7 2
izlaz	izlaz	izlaz
2 3 5	1 3 5 6	3 4 5 6 8
3	1	8
2	3 6	5
5	5	4

Opis algoritma:

Ideja rješenja može se iščitati iz samog teksta zadatka i očito je da će nam za rješenje trebati lista lîstâ. Problem je kako to točno i optimalno kodirati. Sam je zadatak slojevit i sastoji se od nekoliko dijelova. Rješenju zadatka treba pristupiti modularno kako bismo na taj način lakše riješili pojedine dijelove i lakše tražili eventualne pogreške u kodu.

Ulazni su podaci daljine bacanja natjecatelja po serijama. Uočimo da je nakon učitavanja bolje zamijeniti retke i stupce i dalje promatrati daljine bacanja prema natjecateljima. Ovo lukavstvo nije nužno za rješenje, ali će uvelike olakšati kodiranje rješenja.

Proučimo i analizirajmo jedno od mogućih rješenja. Ako tijekom proučavanja koda naiđeš na problem, slobodno dodatno analiziraj i testiraj dio koda koji ti je nejasan koristeći se Python Shellom.

Programski kod:

Prvo ćemo napisati funkciju koja će nam omogućiti zamjenu redaka i stupaca u tablici. Najprije kreiramo novu tablicu koja će imati onoliko redaka i stupaca koliko originalna tablica ima stupaca i redaka. Naredba kopija = [[0 for j in range(redaka)] for i in range(stupaca)] radi upravo to. Uvjeri se. Sada nam je preostalo samo iz originalne tablice vrijednost na poziciji $[i][j]$ prepisati u novu tablicu na poziciju $[j][i]$.

```
def obrni(original, redaka, stupaca):  
    kopija = [[0 for j in range(redaka)] for i in range(stupaca)]  
    for i in range(len(original)):  
        for j in range(len(original[i])):  
            kopija[j][i] = original[i][j]  
    return(kopija)
```

2. Lista listā

Učitajmo podatke u listu lista koju ćemo zvati *serija*. Za to ćemo iskoristiti jedan novi način. U i-ti redak tablice *serija* zapisat ćemo daljine bacanja u i-toj seriji. Nakon učitavanja zamjenom redaka i stupaca kreiramo tablicu *bacaci*. Sada u tablici *bacaci* i-ti redak predstavlja daljine bacanja i-tog natjecatelja po serijama.

```
N = int(input())
serija = [[0] for i in range(3)]
for i in range(3):
    serija[i] = list(map(int, input().split()))
bacaci = obrni(serija, 3, N)
```

Sada trebamo odrediti najbolji rezultat za svakog natjecatelja. Ako do sada nismo uočili prednost zamjene redaka i stupaca, sada je trenutak za to. Naime, vrijednost najdaljeg bacanja i-tog bacača ustvari je maksimalna vrijednost zapisana u i-tom retku tablice *bacaci*. Njihova najdalja bacanja zapisat ćemo u posebnu listu *best_skok* kao uređene parove (*max_daljina_i_tog_bacača, i*).

```
best_skok = []
for i in range(N):
    best_skok += [(max(bacaci[i]), i)]
```

Koji su bacači prošli dalje i dobili pravo na još tri serije? Najboljih N pola bacača. Njih ćemo odrediti sortiranjem liste *best_skok* i prepisivanjem prvih N pola u novu listu *prosli_dalje*. Atribut *reverse* omogućuje sortiranje liste u padajućem poretku. Sortiranjem liste *prosli_dalje*, ovaj put u silaznom poretku, dobivamo uzlazno poredane startne brojeve onih natjecatelja koji su nastavili natjecanje nakon triju serija. Time završava prvi dio zadatka.

```
best_skok.sort(reverse = True)
prosli_dalje = []
for i in range(N // 2):
    prosli_dalje += [best_skok[i][1]]
prosli_dalje.sort()
for i in prosli_dalje:
    print(i + 1, end = ' ')
print()
```

Nastavljamo dalje. U tablicu *nastavak* učitat ćemo daljine bacanja N pola preostalih bacača u sljedećim trima serijama. Nakon učitavanja zamjenom redaka i stupaca kreiramo tablicu *finale*. Sada u listi *finale* i-ti redak predstavlja daljine bacanja i-tog natjecatelja u trima novim serijama. Kako su ti natjecatelji bacali kladivo i u prvim trima serijama, u i-ti redak tablice *finale* dodat ćemo vrijednosti iz tablice *bacaci* koje se nalaze u retku s oznakom *prosli_dalje[i]*. Tako ćemo objediniti daljine bacanja u prve i druge tri serije.

```
nastavak = [[0] for i in range(3)]
for i in range(3):
    nastavak[i] = list(map(int, input().split()))
finale = obrni(nastavak, 3, N // 2)
for i in range(len(finale)):
    finale[i] += bacaci[prosli_dalje[i]]
    finale[i].sort(reverse = True)
```

2. Lista listā

Došli smo do kraja. Preostao nam je najteži dio. Sada treba odrediti dobitnike medalja. Znamo da su redci u tablici *finale* sortirani po veličini. Poznatim algoritmom sortiranja sortiramo natjecatelje (retke) u tablici *finale* prema daljinama bacanja (stupci), a u *prosli_dalje* pamtimo koji redak odgovara kojem bacaču. Pazi, ovdje ne samo da ćemo zamjenjivati vrijednosti kada se za to ukaže potreba već cijeli redak. Kada tijekom usporedbe otkrijemo da je i-ti bacač u nekom bacanju bio lošiji od j-tog bacača, tada ćemo zamijeniti i-ti i j-ti redak u tablici *finale*.

```
for i in range(N // 2):
    for j in range(i, N // 2):
        k = 0
        while k < 6 and finale[i][k] == finale[j][k]:
            k += 1
        if k == 6 or finale[i][k] > finale[j][k]:
            continue
        finale[i], finale[j] = finale[j], finale[i]
        prosli_dalje[i], prosli_dalje[j] = prosli_dalje[j],
        prosli_dalje[i]
```

Tko su osvajači medalja? U slučajevima kada nema višestrukih dobitnika iste medalje može se napisati puno jednostavniji kod. To prepuštamo čitatelju. U ovom trenutku napisat ćemo kod koji pokriva sve slučajeve.

```
k = 0
for medalja in range(3):
    medaljas = [prosli_dalje[k] + 1]
    while k + 1 < N // 2 and finale[k] == finale[k + 1]:
        k += 1
        medaljas.append(prosli_dalje[k] + 1)
    medaljas.sort()
    print(''.join(map(str, medaljas)))
    k += 1
```

Problem 11.

Ovaj zadatak pomalo je neobičan. Postoji mnogo rješenja, a od tebe se ne traži da ispišeš najbolje, nego **što bolje** rješenje.

Dizajn algoritma

Što trebaš učiniti? Trebaš samo prošetati. Točnije, ispisati niz koraka kojim se, počevši **iz ishodišta** (točke (0, 0)), krećeš po cijelobrojnim točkama koordinatnog sustava. Svaki korak pomiče te za jednu točku u jednom od četiriju osnovnih smjerova. Korake predstavljamo znakovima U (gore), L (lijevo), R (desno) i D (dolje). Tvoja šetnja može biti bilo koje duljine, od 1 do 5000 koraka.

Pa dobro, u čemu je štos? Šetnja treba biti takva da čini **što više skretanja**, što manje puta posjećuje isto polje i što manje puta izlazi iz kvadrata kretanja (čija je veličina određena danim brojem N). Preciznije:

- Za svaki korak (osim prvog) koji **skreće**, što znači da nije jednak prethodnom koraku, osvajaš **2 puncta**.

2. Lista lîstâ

- B. Za svaki korak koji vodi u **već posjećeno polje**, ili u polje čija je absolutna vrijednost x ili y koordinate veća od N , dobivaš kaznu od **-3 puncta**.

Naravno, za korake koji zadovoljavaju i A) i B) primjenjuju se oba pravila.

Uzlazni podaci

U jedinome retku nalazi se prirodni broj N ($1 \leq N \leq 20$) koji određuje prostor kretanja korišten u pravilu B.

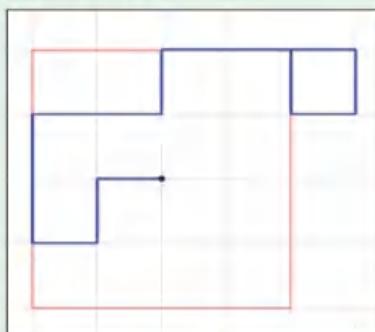
Izlazni podaci

U jedini redak ispiši svoju šetnju – niz sastavljen od najviše 5000 znakova iz skupa {U, L, R, D}, bez razmaka.

Primjeri testnih podataka

ulaz	izlaz	Broj punata
2	LDLUURURRRDLU	$9 * 2 + 3 * (-3) = 9$

Skica primjera:



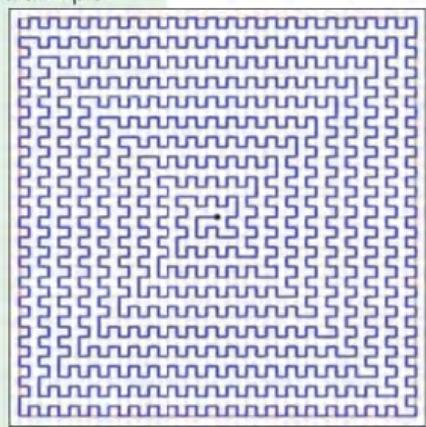
Opis algoritma:

Ovo je neobičan zadatak koji ima mnogo različitih rješenja. Većina rješenja zasniva se na određivanju opće ideje kretanja po koordinatnom sustavu i točnog kodiranja pripadajućeg rješenja. Moguća su i rješenja u kojima se simulira kretanje po tablici u koju zapisujemo jesmo li već posjetili neko polje ili nismo. Čitatelju prepustamo da samostalno razmisli i osmisli svoje rješenje za ovaj zadatak.



U nastavku slijedi opis najboljeg rješenja koje do sada znamo. Ideja je tog rješenja nacrtati "spiralu" punu skretanja. Da bi se olakšala implementacija, krećemo iz gornjeg lijevog kuta i završavamo u ishodištu, a na kraju "izvrnemo" put (tako da počinje u ishodištu). Zašto tako činimo? Nakon što obidemo vanjski rub kvadrata, svodimo problem na manji ($N - 1$) pa možemo ponavljati taj postupak dok ne dođemo do ishodišta. To je mnogo jednostavnije nego da spiralu gradimo iz središta prema van. Slika zorno pokazuje izgled opisane šetnje.

Za bilo koje od tih rješenja potrebna je vrlo pažljiva implementacija. Adrian Satja Kurđija, diplomirani inženjer matematike i osvajač srebrne medalje s međunarodne olimpijade iz informatike, osmislio je ovaj zadatak i ponudio analizu njegovih rješenja.



3. Rekurzije

3.1. Uvod

Shvatiti i prihvatići pojам rekurzije i rekurzivnih funkcija jedna je od težih zadaća koja se stavlja pred učenike srednje škole. Međutim, nakon početne zbumjenosti uočava se potencijal takva načina razmišljanja i rješavanja problema. Kakav je to rekurzivni način rješavanja problema?

3.1.1. Rekurzivni način razmišljanja

Rekurzivni način rješavanja problema svodi se na rješavanje jednostavnijih problema koji izgledaju isto kao početni problem. Ti se jednostavniji problemi onda dalje pojednostavljaju sve dok se ne dođe do trivijalnog problema čije je rješenje očito ili se može jednostavno odrediti. Svakim pojednostavljuvanjem rješavanje problema treba se bližiti kraju.

Primjer 1.

Neka je zadan prirodni broj N. Osmisli rekurzivni način određivanja potencije 2^N .

Rješenje:

Znamo da potenciju 2^N možemo dobiti tako da N puta pomnožimo 2 sa samim sobom. Međutim, znamo i da 2^N možemo zapisati kao $2^N = 2 \cdot 2^{N-1}$. Očito je da smo na taj način originalni problem razbili na jednostavniji problem istog izgleda. Znači, kada bismo na neki način saznali koliko je 2^{N-1} , tada bismo jednostavno tu vrijednost pomnožili s 2 i rješili originalni problem. Snaga je rekurzije u tome da sada možemo nastaviti razbijati i taj jednostavni problem na sve manje i manje. Pokažimo način rješavanja tog problema kroz jednu priču.



$$2^4 = 2^3 * 2$$

$$2^3 = 2^2 * 2$$

$$2^2 = 2^1 * 2$$

$$2^1 = 2$$

$$2^2 = 2 * 2 = 4$$

$$2^3 = 4 * 2 = 8$$

$$2^4 = 8 * 2 = 16$$

Nikola treba izračunati vrijednost potencije 2^4 . Zna da će ako otkrije vrijednost potencije 2^3 , to pomnožiti s 2 i lako dobiti traženi odgovor. Zato je pitao Ljiljanu zna li ona koliko je 2^3 .

Ljiljana nije odmah znala odgovor na to pitanje. Ali se dosjetila da ako sazna koliko je 2^2 , da će onda lako taj broj pomnožiti s 2 i dobiti odgovor. Zato je pitala Zvonimira zna li on koliko je 2^2 .

Zvonimir nije odmah znao odgovor na to pitanje. Ali se dosjetio da ako sazna koliko je 2^1 , da će onda lako taj broj pomnožiti s 2 i dobiti odgovor. Zato je pitao Sanju zna li ona koliko je 2^1 .

Sanja je znala da je odgovor na pitanje broj 2 jer je riječ o trivijalnom slučaju kada je potencija broja na jedan upravo taj broj. Svoj odgovor proslijedila je Zvonimiru.

Zvonimir je dobiveni odgovor (2) pomnožio s 2 i dobio 4. Taj odgovor proslijedio je Ljiljani.

3. Rekurzije

Ljiljana je dobiveni odgovor (4) pomnožila s 2 i dobila 8. Svoj je odgovor proslijedila Nikoli. Nikola je dobiveni odgovor (8) pomnožio s 2 i dobio rješenje originalnog problema: 16.

3.1.2. Rekurzivna relacija

Kada procijenimo da je neki problem rješiv rekurzivnim načinom, nužno je definirati pravilo (relaciju) po kojem se problem razbija na jednostavnije. Takvu relaciju nazivamo **rekurzivnom relacijom**. Općenito, neka je zadan problem $P(n)$ koji ovisi o parametru n . Tada ćemo rekurzivnom relacijom nazivati svaku relaciju koja $P(n)$ izražava s pomoću istog problema, ali koji sada ovisi o jednom parametru ili više parametara k koji su strogo manji od n . Npr. $P(n)=n+P(n-1)$ ili $P(n)=P(n-1)+P(n-2)+1$.

Rekurzivna
relacija

Uz rekurzivnu relaciju, nužno je definirati i uvjet završetka rekurzivnog pozivanja. To je trivijalni slučaj koji se lako računa bez rekurzivnog načina ili je već unaprijed poznat.

Uvjet
završetka

Npr. u *Primjeru 1.* rekurzivna relacija bila je oblika $2^N=2\cdot2^{N-1}$, a uvjet završetka $2^1=2$.

Primjer 2.

Osmisli rekurzivno rješenje, tj. rekurzivnu relaciju i uvjet završetka za računanje zbroja prvih n prirodnih brojeva.

Rješenje:

Označimo zbroj prvih n prirodnih brojeva sa $zbroj(n)$. Znamo da vrijedi $zbroj(n)=1+2+3+\dots+(n-1)+n$. Primijetimo da se navedeni zbroj može zapisati u obliku $zbroj(n)=[1+2+3+\dots+(n-1)]+n$ kao zbroj prvih $n-1$ prirodnih brojeva i broja n . Iz tog zapisa uočavamo da vrijedi $zbroj(n)=zbroj(n-1)+n$. Time smo dobili rekurzivnu relaciju koja opisuje rekurzivni način rješavanja tog problema.

Iz relacije je vidljivo da se parametar prilikom svakog rekurzivnog poziva smanjuje za jedan, a kako se ne može smanjivati do beskonačnosti, trebamo definirati trenutak kada ćemo prestati s rekurzivnim pozivom. Ovdje je jasno da ćemo s rekurzivnim pozivanjem prestati kada n postane jedan i da će tada $zbroj(1)$ imati vrijednost jedan. Pogledajmo jedan primjer ($n=4$):

$$zbroj(4) = zbroj(3) + 4$$

$$zbroj(3) = zbroj(2) + 3$$

$$zbroj(2) = zbroj(1) + 2$$

$$zbroj(1) = 1$$

$$zbroj(2) = 1 + 2 = 3$$

$$zbroj(3) = 3 + 3 = 6$$

$$zbroj(4) = 6 + 4 = 10$$

Primjer 3.

Faktorijsle ($n!$, n faktorijsla) su matematički pojma koji se definira sljedećom relacijom:

$n!=n*(n-1)*(n-2)*\dots*3*2*1$. Osmisli rekurzivno rješenje tog problema.

3. Rekurzije

Rješenje:

Leonardo od Pise (1170.-1250.), poznat još kao Fibonacci (*filius Bonacci* ili Bonaccijev sin) uveo je 1202. godine u svojem radu *Liber Abaci* opisani slijed brojeva.

Zadanu matematičku relaciju možemo zapisati kao $n!=n*(n-1)*(n-2)*\dots*3*2*1$. Iz tog zapisa očito je da vrijedi $n!=n*(n-1)!$. Time smo odredili pripadajuću rekurzivnu relaciju. Kako odrediti uvjet završetka? Vidimo da će se vrijednost parametra smanjivati svakim sljedećim korakom i da će sigurno u nekom trenutku doći do manjih vrijednosti, poput tri, dva i jedan. Te vrijednosti možemo ručno izračunati koristeći se originalnom relacijom. Očito je da rekurzivni pozivi trebaju prestati onog trenutka kada pokušamo izračunati vrijednost od 1! jer je to trivijalan i očit slučaj. Znači, kada n postane jedan, prestati ćemo razbijati problem na manje.

Primjer 4.

Osmisli rekurzivno rješenje za računanje n-tog Fibonaccijeva broja i odredi šesti takav broj.

Rješenje:

Kakvo bi to bilo učenje rekurzivnog načina razmišljanja da se u njemu ne spomenu Fibonaccijevi brojevi? Po definiciji, n -ti Fibonaccijev broj jednak je zbroju prethodnih dvaju brojeva, gdje prva dva broja imaju vrijednost jedan. Rekurzivna relacija izlazi iz definicije i vrijedi da je $F_n=F_{n-1}+F_{n-2}$, a rekurzivno pozivanje staje kada zatražimo izračun vrijednosti F_1 i F_2 jer znamo da je njihova vrijednost unaprijed definirana i iznosi $F_1=1$ i $F_2=1$. Odredimo vrijednost F_6 .

		$F_6 = F_5 + F_4$			
$F_5 = F_4 + F_3$			$F_4 = F_3 + F_2$		
$F_4 = F_3 + F_2$		$F_3 = F_2 + F_1$		$F_3 = F_2 + F_1$	
$F_3 = F_2 + F_1$		$F_2 = 1$	$F_1 = 1$	$F_2 = 1$	$F_1 = 1$
$F_2 = 1$	$F_1 = 1$	$F_2 = 1$		2	
2		2		3	
3				5	
				8	

Vidimo da se tijekom računanja nekoliko puta zatražilo računanje istog problema. Poslije ćemo vidjeti kako možemo izbjegići takva multipliciranja izračuna.

3.1.3. Rekurzivne funkcije u Pythonu

Izvršavanje rekurzivne relacije, tj. određivanje konačne vrijednosti za zadani početni parametar zna biti dosta zamorno. Zamislimo da na rekurzivni način trebamo izračunati 25. Fibonaccijev broj ($F_{25}=75025$). Nije neizvedivo, ali je naporno. Međutim, ako u priču uklju-

3. Rekurzije

čimo i računala, tada će rekurzivni način rješavanja problema dobiti još više na značaju.

Rekurzija je važan dio programiranja. Velik broj algoritama u svojoj osnovi ima rekurzivnu ideju s kojom učinkovito i brzo rješava problem. Općenito, rekurzija se upotrebljava u svim programima u kojima treba ispitati veliki broj slučajeva jer se time najmanje gubi na testiranju nepotrebnog ili na testiranju istih stvari. Rekurziju treba izbjegavati u slučajevima kad postoji optimalna nerekurzivna verzija programa jer su nam uvijek na prvom mjestu brzina i jednostavnost.

Rekurzivna funkcija svaka je funkcija koja poziva samu sebe. Pisanje rekurzivnih funkcija oslanja se na osnove rekurzivnog razmišljanja. Rekurzivne funkcije treba pisati tako da je svakim novim pozivom posao koji ta funkcija obavlja sve bliži kraju. Takve će rekurzivne funkcije sigurno završiti s poslom u konačno mnogo koraka.

Kako bismo napisali rekurzivnu funkciju koja će riješiti postavljeni problem, a koju će računalo moći izvršiti, nužno je odraditi sljedeće korake:

- pronaći rekurzivnu relaciju za zadani problem
- pronaći uvjet prekida rekurzivnog pozivanja funkcije
- napisati rekurzivnu funkciju.

Primjer 5.

Napiši rekurzivnu funkciju koja će računati potenciju 2^N za zadani prirodni broj N.

Rješenje:

Tražena je rekurzivna funkcija:

```
def rekurzija(N):  
    if N == 1:  
        return 2  
    else:  
        return rekurzija(N - 1) * 2
```

Znamo da je rekurzivna relacija koja rješava taj problem oblika $2^N=2\cdot2^{N-1}$, a uvjet završetka $2^1=2$. Napisati rekurzivnu funkciju jednostavno je ako se znaju te dvije stvari.

Funkcija *rekurzija* kao ulazni parametar prima vrijednost eksponenta za koji želimo izračunati potenciju. Na ulazu u funkciju uvijek provjerimo uvjet završetka. Ako je on ispunjen, završavamo izvršenje tog poziva funkcije i kao vrijednost vraćamo broj dva. Ako uvjet nije ispunjen, funkcija poziva samu sebe, ali ovaj put s ulaznim parametrom za jedan manjim nego što je ona bila pozvana.

Tako napisanu rekurzivnu funkciju možemo upotrijebiti vrlo jednostavno. Dovoljno ju je samo pozvati u glavnom programu.

```
N = int(input())  
print(rekurzija(N))
```



Priča se da je slavni matematičar Ludolph van Ceulen (1540.–1610.) proveo trećinu svojeg života računajući prvi 35 decimala broja π ! Zbog te njegove upornosti i posvećenosti broju π dugo su zvali Ludolphovim brojem. Za usporedbu, danas znamo pet triljuna znamenki koje su se 90 dana računale na posebnom računalu.

Rekurzivna funkcija

3. Rekurzije

Tijek izvršavanja
rekurzije

Ako malo modificiramo tu rekursivnu funkciju, lakše ćemo uočiti kako izgleda cijeli tijek izvršenja rekurzije. Uoči da će rekursivna funkcija koja se prva pozove posljednja završiti, nakon što se završi izvršavanje svih ostalih poziva rekursivne funkcije.

```
def rekursija(N):  
    print('Poziva se funkcija za N =', N)  
    if N == 1:  
        print('Kraj funkcije za N =', N)  
        print('Vraćena vrijednost', 2)  
        return 2  
    else:  
        koliko = rekursija(N - 1)  
        print('Kraj funkcije, N =', N)  
        print('Vraćena vrijednost', koliko * 2)  
        return koliko * 2  
  
N = int(input('N = '))  
print(rekursija(N))
```

	N = 4
	Poziva se funkcija za N = 4
	Kraj funkcije za N = 3
	Poziva se funkcija za N = 2
	Poziva se funkcija za N = 1
	Kraj funkcije za N = 1
	Vraćena vrijednost 2
	Kraj funkcije za N = 2
	Vraćena vrijednost 4
	Kraj funkcije za N = 3
	Vraćena vrijednost 8
	Kraj funkcije za N = 4
	Vraćena vrijednost 16
	16

Primjer 6.

Napiši rekursivnu funkciju koja će računati zbroj prvih n prirodnih brojeva.

Rješenje:

Već smo odredili rekursivnu relaciju i uvjet završetka. Sada je jednostavno napisati rekursivnu funkciju.

```
def F(n):  
    if n == 1:  
        return 1  
    else:  
        return F(n-1) + n
```

Primjer 7.

Napiši rekursivnu funkciju koja će računati $n!$.

Rješenje:

```
def F(N):  
    if N == 1:  
        return 1  
    else:  
        return F(N-1) * N
```

Primjer 8.

Napiši rekursivnu funkciju za računanje n -tog Fibonaccijeva broja.

3. Rekurzije

Rješenje:

```
def F(N):
    if N == 1 or N == 2:
        return 1
    else:
        return F(N-1) + F(N-2)
```

Jednostavnim upisivanjem i pokretanjem te funkcije lako ćemo utvrditi da ovo rješenje za $N > 35$ postaje jako loše, tj. dugo se izvršava.

Nikad, ali baš nikad
nemojte koristiti rekurziju
da bi izračunali n-ti
Fibonaccijev broj. Za to
ima puno ljepeših i lakših
načina. [Googlajte ih!](#)

Primjer 9.

Napiši rekurzivnu funkciju koja će određivati duljinu zadanog stringa.

Duljina stringa

Rješenje:

Prvo trebamo osmisliti pripadajuću rekurzivnu relaciju. Duljina stringa jednaka je duljini stringa bez posljednjeg znaka plus jedan, znači $duljina(s) = duljina(s[:-1]) + 1$. Za uvjet završetka možemo uzeti duljinu praznog stringa, što je nula. Pripadajuća rekurzivna funkcija sljedećeg je oblika:

```
def duljina(s):
    if s == '':
        return 0
    else:
        return duljina(s[:-1]) + 1
```

Primjer 10.

Euklidov algoritam

Napiši rekurzivnu funkciju za računanje najveće zajedničke mjere pomoću Euklidova algoritma.

Rješenje:

Euklidov algoritam kaže: ako su dva broja a i b jednakia, onda je njihova mjera taj broj (a ili b); inače je mjera jednakia mjeri brojeva $|a-b|$ i manjeg od brojeva a ili b . Kada se to zna, lako je postaviti rekurzivnu relaciju koja je oblika:

$$\text{NZM}(a,b) = \text{NZM}(|a - b|, \min(a,b)).$$

Funkciju prestajemo rekurzivno pozivati kada je $a=b$ i tada je $\text{NZM}(a,b) = a$ ili b . Pripadajuća rekurzivna funkcija oblika je:

```
def Euklid(a, b):
    if a == b:
        return a
    elif a > b:
        return Euklid(a - b, b)
    else:
        return Euklid(b - a, a)

a, b = map(int, input().split())
print(Euklid(a, b))
```

3. Rekurzije

Primjer 11.

Napiši rekurzivnu funkciju koja će određivati najveću vrijednost u zadanoj listi brojeva.

Rješenje:

U ovom slučaju rekurzivna relacija nije zadana jednom relacijom. Naime, da bismo odredili najveći element u listi, trebamo zadnji element u njoj usporediti s najvećim elementom u ostatku liste. Ako je on veći, onda će funkcija vratiti njega, a ako nije, onda će vratiti najveći element u ostatku polja. Kao uvjet završetka promatrati ćemo slučaj kada u listi ima samo jedan element koji je ujedno i najveći element te liste. Kako bismo pokazali da u rekurzivnoj funkciji nije nužno da postoji samo jedan ulazni parametar, napisat ćemo funkciju s dvama ulaznim parametrima.

```
def najveci(L, n):
    if n == 0:
        return L[0]
    else:
        tmp = najveci(L[:-1], n - 1)
        if L[n] > tmp:
            return L[n]
        return tmp

L = list(map(int, input().split()))
print(najveci(L, len(L) - 1))
```

Primjer 12.

Neka su zadani cijeli brojevi *baza* i *pot*. Napiši rekurzivnu funkciju za određivanje vrijednosti izraza $baza^{pot}$.

Rješenje:

Nešto smo slično već rješavali. Određivali smo vrijednost izraza 2^N , gdje je *N* bio prirodni broj. Sada to trebamo proširiti na skup cijelih brojeva i na opću bazu. Očito ćemo morati razlikovati dva slučaja: kada je potencija pozitivna i kada je negativna. Za pozitivnu potenciju znamo pripadajuću rekurzivnu relaciju. Promotrimo jedan primjer kada je potencija negativna, npr. 2^{-4} . Taj izraz možemo zapisati u obliku $2^{-4} = 2^{-3} \cdot 2^{-1} = 2^{-3}/2$. Iz tog primjera uočavamo pravilnost i pripadajuću rekurzivnu relaciju $baza^{pot} = baza^{pot+1}/baza$. Dodatno, ako je potencija jednaka nuli, rezultat je jedan.

Kako bismo dobili na preglednosti, obje ćemo relacije objediniti pod jednim nazivom *power()* i zapisati na jedan pregledan način.

$$power(baza, pot) = \begin{cases} power(baza, pot - 1) * baza & \text{ako je } pot > 0 \\ 0 & \text{ako je } pot = 0 \\ power(baza, pot + 1) / baza & \text{ako je } pot < 0 \end{cases}$$

Sad je jednostavno napisati pripadajuću rekurzivnu funkciju.

3. Rekurzije

```
def power(baza, pot):
    if pot == 0:
        return 1
    if pot > 0:
        return power(baza, pot - 1) * baza
    if pot < 0:
        return power(baza, pot + 1) / baza

baza, pot = map(int, input().split())
print(power(baza, pot))
```

Zadaci

- Napiši rekurzivnu funkciju koja računa sljedeće zbrojeve:

$$\text{a. } \sum_{i=1}^n \frac{1}{i} \quad \text{b. } \sum_{i=1}^n (-1)^i$$

- Implementiraj jednu od inaćica Euklidova algoritma:
 - ako je $a=b$, onda je mjera brojeva a i b jednaka a ili b
 - ako je $a < b$, onda je mjera brojeva a i b jednaka mjeri brojeva b i $a \bmod b$
 - ako je $b < a$, onda je mjera brojeva a i b jednaka mjeri brojeva b i $b \bmod a$.
- Napiši rekurzivni potprogram s jednim parametrom, prirodnim brojem n . Ako je paran, n postaje $n/2$, inače n postaje $3n+1$. Cilj je postići da n postane 1. Napiši funkciju koji će unositi prirodni broj n i vraćati koliko će se puta n promijeniti prije nego postane 1.
- Napiši rekurzivnu funkciju koji će računati minimum niza od n brojeva.
- Napiši rekurzivnu funkciju koji će iz stringa izbacivati svaki drugi znak počevši od drugog znaka.
- Napiši rekurzivni potprogram koji će pretvarati broj iz baze 10 u bazu b .

Tornjevi Hanoja

Problem

Dana su tri štapa označena s A, B, C koji stoje uspravno pričvršćeni za ravnu podlogu. Na štapu A nalazi se n diskova različitih promjera, i to tako da je disk najvećeg promjera prvi, na njemu disk manjeg promjera, a na vrhu je disk najmanjeg promjera. Sve diskove sa štapa A treba premjestiti na štap B, ali tako da u svakom trenutku možemo premjestiti samo jedan disk i nikada disk većeg promjera ne smije biti na disku manjeg promjera. Štap C služi nam kao pomoć pri premještanju. Treba osmislitи rekurzivnu funkciju čiji će ulazni parametar biti prirodni broj diskova na štapu A, a ispisivat će sva premeštanja koja trebamo napraviti da bismo preselili sve diskove sa štapa A na štap B uz dane uvjete.

3. Rekurzije



Legenda

Legenda kaže da je prije mnogo godina negdje oko grada Hanoja živio jedan car koji je tražio novog dvorskog mudraca. Budući da je i sam bio mudar, želio je pronaći što boljeg mudraca, pa je odlučio uzeti onoga tko dâ najbolje rješenje postavljenog problema (zagonetke): dana su tri štapa i n diskova različitog promjera. Svi diskovi postavljeni su na prvom štapu tako da se manji disk uvijek nalazi iznad većeg. Treba prebaciti sve diskove s izvornog štapa na ciljni štap prebacujući jedan po jedan disk i koristeći se trećim štapom kao pomoćnim, ali da se ni u jednom trenutku disk većeg promjera ne nađe na disku manjeg promjera.

Mnogi mudraci iz cijele zemlje dolazili su pred cara s raznim rješenjima, ali su rješenja bila ili nerazumljiva ili preduga. "Mora postojati jednostavniji način", razmišljao je car. Jednog je dana pred cara stigao, barem tako kaže legenda, Buda, koji je rekao da je problem tako jednostavan da se rješava sam od sebe. Svoje rješenje Buda je izložio na sljedeći način:

1. Ako postoji samo jedan disk, pomaknemo ga s izvornog štapa na ciljni štap, i to je toliko jednostavan posao da ga svaka seoska luda može uraditi;
2. Ako pak ima više od jednog diska, postupak je sljedeći:
 - premjestimo prvo $n-1$ diskova s izvornog na pomoćni štap, koristeći se ciljnim štapom kao pomoćnim
 - budući da je $n-1$ diskova na pomoćnom štalu, a najveći je i dalje ostao na izvornom, problem se svodi na točku 1, tj. treba prebaciti taj jedan disk s izvornog štapa na ciljni štap
 - potom treba $n-1$ diskova na isti način prebaciti s pomoćnog štapa na ciljni štap (sada se koristeći izvornim štapom kao pomoćnim).

Kada je Buda završio s pričom, car ga je upitao kada će konačno reći svoje rješenje. Buda se samo nasmiješio i otiašao. Legenda dalje kaže kako su budistički svećenici prihvatali izazov i počeli premještati 64 diska. Nadalje, legenda kaže kako će, kada oni to završe, svijet prestati postojati i biti zamijenjen nečim još bizarnijim i neobjašnjivim. Neki smatraju da su svećenici to odavno uspjeli učiniti.

Početni broj diskova: 4
Potez: 1 s 1 na 2
Potez: 2 s 1 na 3
Potez: 1 s 2 na 3
Potez: 3 s 1 na 2
Potez: 1 s 3 na 1
Potez: 2 s 3 na 2
Potez: 1 s 1 na 2
Potez: 4 s 1 na 3
Potez: 1 s 2 na 3
Potez: 2 s 2 na 1
Potez: 1 s 3 na 1
Potez: 3 s 2 na 3
Potez: 1 s 1 na 2
Potez: 2 s 1 na 3
Potez: 1 s 2 na 3

Pojasnimo što se događa za $n=3$.

Imamo sljedeća premještanja:

- s A na B
- s A na C
- s B na C - u ovom su trenutku dva gornja diska sa štapa A premještena na štap C
- s C na A
- s C na B
- s A na B.

Sve se to može kraće zapisati na sljedeći način:

- premjesti dva gornja diska s A na C uz pomoć B
- premjesti disk s A na B
- premjesti dva diska s C na B uz pomoć A.

Ili, općenito, za n diskova premještanje ima oblik:

- premjesti $n-1$ gornjih diskova s A na C uz pomoć B
- premjesti disk s A na B
- premjesti $n-1$ disk s C na B uz pomoć A.

Time je ustvari dobivena rekurzivna relacija, tj. našli smo način kako ćemo premjestiti n diskova ako znamo premjestiti $n-1$ diskova. Rekurzivno pozivanje staje kada je $n=1$, i u tom ćemo slučaju premjestiti jedan disk sa štapa A na štap B. Ovaj algoritam ne vraća nikakvu vrijednost u glavni program, već samo ispisuje odgovarajuća premještanja.

```
def hanoi(disk, pocetni='1', zavrnsni='3', pomocni='2'):  
    if disk > 0:  
        hanoi(disk - 1, pocetni, pomocni, zavrnsni)  
        hanoi(disk - 1, pocetni, pomocni, zavrnsni)  
        print('Potez: ' + str(disk) + ' s ' + pocetni + '  
na ' + zavrnsni)  
        hanoi(disk - 1, pomocni, zavrnsni, pocetni)  
  
n = int(input('Početni broj diskova: '))  
hanoi(n)
```

3. Rekurzije



Tko bi rekao da o ovako jednostavnom algoritmu ovisi sudbina svijeta. Ali, ima nade jer je za $n=64$ to ukupno 18446744073709551615 premještanja ili 584 000 godina uz brzinu od milijun prijenosa u sekundi.

3.2. Rekurzije u zadacima

Osmisliti i napisati dobru rekurziju koja će elegantno i učinkovito riješiti zadani problem nije uvijek lagan zadatak. Kako bismo ipak što bolje razumjeli taj pojam, u ovom ćemo se poglavljju posvetiti izvršavanju gotovih rekurzija. Njihovo izvršavanje više će podsjećati na rješavanje matematičkih zadataka, dok ćemo se informatičkim vještinama koristiti kada to bude potrebno. Krenimo redom.

Problem 1.

Odredi vrijednost rekurzije f za zadani parametar $x = 7$, tj. odredi $f(7)$ te napiši rekurzivnu funkciju i odredi vrijednost $f(512)$.

$$f(x) = \begin{cases} f(x-3)+1 & x \geq 3 \\ f(x+2)-3 & 1 \leq x < 3 \\ x^2+x+1 & x < 1 \end{cases}$$

Jednostavna
rekurzija

Rješenje:

Slijednim izvršavanjem zadanih rekurzivnih relacija doći ćemo do traženog rješenja. Na početku je vrijednost parametra x sedam. Kako je on u tom trenutku veći ili jednak tri, pozvat ćemo rekurziju primjenjujući prvu rekurzivnu relaciju $f(x-3)+1$. U sljedećem koraku rekurzije vrijednost parametra x jest četiri. Kako je on i dalje veći ili jednak tri, ponovo ćemo primijeniti prvu rekurzivnu relaciju $f(x-3)+1$. U sljedećem koraku parametar x iznosi jedan i zbog toga sada primjenjujemo drugu rekurzivnu relaciju $f(x+2)-3$. Nakon primjene te relacije x postaje tri i sada opet primjenjujemo prvu relaciju $f(x-3)+1$. U ovom je trenutku x postao nula, što je jedna od mogućih vrijednosti ($x < 1$) zbog koje dočini do prestanka pozivanja rekurzije.

$$\begin{aligned} f(7) &= f(7-3)+1 = f(4)+1 = \\ &= f(4-3)+1+1 = f(1)+2 = \\ &= f(1+2)-3+2 = f(3)-1 = \\ &= f(3-3)+1-1 = f(0) = \\ &= 0+0+1 = 1 \end{aligned}$$

Uočimo da smo tijekom određivanja vrijednosti rekurzije kada bismo uočili da nešto možemo sami izračunati, to i napravili. Npr. u drugom smo retku odmah zbrojili $1+1$. Takav način izvršavanja rekurzije prirodniji je čovjeku i našem logičkom razmišljanju. Međutim, rekurzija se na računalu izvršava na malo drugačiji način.

3. Rekurzije

Svaki put kada se u trenutnom pozivu rekurzije dogodi novi poziv, trenutna rekurzija staje s radom i čeka završetak i povratnu informaciju od rekurzije koja se iz nje pozvala.

$$\begin{aligned}f(7) &= f(7 - 3) + 1 = f(4) + 1 = 0 + 1 = 1 \\f(4) &= f(4 - 3) + 1 = f(1) + 1 = -1 + 1 = 0 \\f(1) &= f(1 + 2) - 3 = f(3) - 3 = 2 - 3 = -1 \\f(3) &= f(3 - 3) + 1 = f(0) + 1 = 1 + 1 = 2 \\f(0) &= 0 + 0 + 1 = 1\end{aligned}$$

Rekurzivna funkcija:

Kako imamo zadalu rekurzivnu relaciju i uvjet završetka, lako ćemo napisati pripadajuću rekurzivnu funkciju i dobiti da je $f(512)=167$.

```
def f(x):
    if x < 1:
        return x * x + x + 1
    elif x >= 3:
        return f(x - 3) + 1
    else:
        return f(x + 2) - 3
print('f(512)=', f(512))
>>>
f(512)= 167
```

Čitatelju prepuštamo malu modifikaciju te rekurzivne funkcije kako bismo dokazali sljedeće tvrdnje:

- rekurzija f pozvana je ukupno 175 puta
- funkcija nikad nije pozvana više od jednom s nekim parametrom.

Problem 2.

Odredi vrijednost rekurzije f za zadani parametar $x = 23$ te napiši rekurzivnu funkciju i odredi vrijednost $f(512)$.

$$f(x) = \begin{cases} f(x-6)+2 & x > 10 \\ x^2+1 & 6 \leq x < 10 \\ 2f(x+1)-1 & x < 6 \end{cases}$$

Rješenje:

$$\begin{aligned}f(23) &= f(17) + 2 = \\&= f(11) + 2 + 2 = f(11) + 4 = \\&= f(5) + 2 + 4 = f(5) + 6 = \\&= 2 * f(6) - 1 + 6 = 2 * f(6) + 5 = \\&= 2 * (6 * 6 + 1) + 5 = 79\end{aligned}$$

3. Rekurzije

Rekurzivna funkcija:

```
def f(x):
    if x >= 6 and x <= 10:
        return x * x + 1
    elif x > 10:
        return f(x - 6) + 2
    else:
        return 2 * f(x + 1) - 1

print(f(512))
>>>
233
```

Problem 3.

Odredi vrijednost rekurzije f za zadani parametar $x = 7$ te napiši rekurzivnu funkciju i odredi vrijednost $f(256)$.

$$f(x) = \begin{cases} f(x-3)+3 & x > 0, x \text{ neparan} \\ f(x+1)-1 & x > 0, x \text{ paran} \\ 0 & x \leq 0 \end{cases}$$

Rješenje:

$$\begin{aligned} f(7) &= f(4) + 3 = \\ &= f(5) - 1 + 3 = f(5) + 2 = \\ &= f(2) + 3 + 2 = f(2) + 5 = \\ &= f(3) - 1 + 5 = f(3) + 4 = \\ &= f(0) + 3 + 4 = f(0) + 7 = \\ &= 0 + 7 = 7 \end{aligned}$$

Rekurzivna funkcija:

```
def f(x):
    if x <= 0:
        return 0
    if x > 0 and x % 2 == 1:
        return f(x - 3) + 3
    if x > 0 and x % 2 == 0:
        return f(x + 1) - 1

print(f(256))
>>>
256
```

Provjerom dodatnih slučajeva dokaži slutnju da samo za $x = 1$ vrijedi da je $f(x) \neq x$, za svaki prirodni broj x .

3. Rekurzije

Problem 4.

Odredi vrijednost rekurzije f za zadani parametar $x = 5$ te napiši rekurzivnu funkciju i odredi vrijednost $f(75)$.

Rekurzivni poziv u rekurziji

$$f(x) = \begin{cases} f(f(x-3)) + 3 & x > 0 \\ x^2 - 2 & x \leq 0 \end{cases}$$

Rješenje:

Ta je rekurzija malo drugačija od svih koje smo do sada napisali. Pripadajuća rekurzivna relacija kao parametar pri pozivu koristi se vrijednošću koju dobije nakon izvršenja rekurzije za neki drugi parametar. Pogledajmo što se događa tijekom izvršenja te rekurzije.

$$f(5) = f(f(5 - 3)) + 3 = f(f(2)) + 3 = f(2) + 3 = 2 + 3 = 5$$

$$\begin{aligned} f(2) &= f(f(2 - 3)) + 3 = f(f(-1)) + 3 = f(-1) + 3 = -1 + 3 = 2 \\ f(-1) &= (-1)^*(-1) - 2 = -1 \end{aligned}$$

Uočimo da se tijekom izvršenja rekurzija dva puta pozvala s parametrom dva i dva puta s parametrom -1. Međutim nije bilo potrebno dva puta računati te vrijednosti, već samo prilikom prvog poziva, dok smo sljedeći put samo iskoristili dobivenu vrijednost.

Rekurzivna funkcija:

```
def f(x):
    if x <= 0:
        return x * x - 2
    else:
        return f(f(x - 3)) + 3

print(f(75))
>>>
77
```

Nažalost, ta se rekurzija jako sporo izvršava jer se s povećanjem zadanog parametra poveća broj nepotrebnih višestrukih poziva rekurzije za istu vrijednost. Zbog toga se i tražila vrijednost $f(75)$: naime, za veću vrijednost autor ovog teksta nije imao strpljenja dočekati kraj izvršenja.

Problem 5.

Odredi vrijednost rekurzije f za zadani parametar $x = 9$ te napiši rekurzivnu funkciju i odredi vrijednost $f(1001)$.

$$f(x) = \begin{cases} f(f(x-2)+1) & x \geq 4 \\ f(x-1)-1 & 2 \leq x < 4 \\ x+3 & x < 2 \end{cases}$$

3. Rekurzije

Rješenje:

$$\begin{aligned}f(9) &= f(f(7) + 1) = f(2 + 1) = f(3) = 2 \\f(7) &= f(f(5) + 1) = f(2 + 1) = f(3) = 2 \\f(5) &= f(f(3) + 1) = f(2 + 1) = f(3) = 2 \\f(3) &= f(2) - 1 = 3 - 1 = 2 \\f(2) &= f(1) - 1 = 4 - 1 = 3 \\f(1) &= 1 + 3 = 4\end{aligned}$$

U ovom primjeru $f(3)$ smo izračunali jednom i upotrijebili još četiri puta. Nažalost, računalo će morati svaki put iznova računati tu vrijednost.

Rekurzivna funkcija:

```
def f(x):  
    if x < 2:  
        return x + 3  
    elif x >= 4:  
        return f(f(x - 2) + 1)  
    else:  
        return f(x - 1) - 1  
  
print(f(1001))  
>>>  
2
```

Međutim, ako probamo pronaći vrijednost rekurzije za neki parni broj, otkrit ćemo da se tada ova rekurzija beskonačno izvršava. Provjeri nekoliko slučajeva i uvjeri se u tu tvrdnju.

Problem 6.

Odredi vrijednost rekurzije f za zadani parametar $x = 12$ te napiši rekurzivnu funkciju i odredi vrijednost $f(1000)$.

$$f(x) = \begin{cases} 1 + f(f(x-5)) & x > 10 \\ f(x-3)-2 & 0 < x \leq 10 \\ 1-x & x \leq 0 \end{cases}$$

Rješenje:

$$\begin{aligned}f(12) &= 1 + f(f(7)) = 1 + f(-3) = 1 + 1 - (-3) = 5 \\f(7) &= f(4) - 2 = -1 - 2 = -3 \\f(4) &= f(1) - 2 = 1 - 2 = -1 \\f(1) &= f(-2) - 2 = 3 - 2 = 1 \\f(-2) &= 1 - (-2) = 3\end{aligned}$$

3. Rekurzije

Rekurzivna funkcija:

```
def f(x):
    if x <= 0:
        return 1 - x
    elif x > 10:
        return 1 + f(f(x - 5))
    else:
        return f(x - 3) - 2

print(f(1000))
>>>
1
```

Problem 7.

Odredi vrijednost rekurzije f za zadane parametre $x = 10$ i $y = 5$ te napiši rekurzivnu funkciju i odredi vrijednost $f(1979, 2015)$.

Rekurzija s dva parametra

$$f(x,y) = \begin{cases} f(x-2, y+1)+3 & x > y \\ 5 & x = y \\ f(x+1, y-3)+2 & x < y \end{cases}$$

Rješenje:

Ovaj put rekurzija ovisi o dvama parametrima. Ideja je rješenja ista, samo oprez treba biti dvostruko veći.

$$\begin{aligned} f(10,5) &= f(8,6) + 3 = 13 + 3 = 16 \\ f(8,6) &= f(6,7) + 3 = 10 + 3 = 13 \\ f(6,7) &= f(7,4) + 2 = 8 + 2 = 10 \\ f(7,4) &= f(5,5) + 3 = 5 + 3 = 8 \\ f(5,5) &= 5 \end{aligned}$$

Rekurzivna funkcija:

```
def f(x,y):
    if x == y:
        return 5
    elif x > y:
        return f(x - 2,y + 1) + 3
    else:
        return f(x + 1,y - 3) + 2

print(f(1979,2015))
>>>
23
```

3. Rekurzije

Problem 8.

Odredi vrijednost rekurzije f za zadane parametre $x = 13$ i $y = 2$ te napiši rekurzivnu funkciju i odredi vrijednost $f(42, 65)$.

$$f(x,y) = \begin{cases} f(x-3, y+2)-1 & x > 3 \\ y+x & x = 3 \\ f(2x+1, y-4)+1 & x < 3 \end{cases}$$

Rješenje:

$$f(13,2) = f(10,4) - 1 = 7 - 1 = 6$$

$$f(10,4) = f(7,7) - 1 = 8 - 1 = 7$$

$$f(7,6) = f(4,8) - 1 = 9 - 1 = 8$$

$$f(4,8) = f(1,10) - 1 = 10 - 1 = 9$$

$$f(1,10) = f(3,6) + 1 = 9 + 1 = 10$$

$$f(3,6) = 3 + 6 = 9$$

Rekurzivna funkcija:

```
def f(x,y):
    if x == 3:
        return y + x
    elif x > 3:
        return f(x - 3,y + 2) - 1
    else:
        return f(2 * x + 1,y - 4) + 1

print(f(42,65))
>>>
81
```

Problem 9.

Neka je zadana rekurzija f . Odredi vrijednost poziva $f(f(f(7)))$.

$$f(x) = \begin{cases} f(x-2)+1 & x \geq 5 \\ f(x-1)-x & 2 < x < 5 \\ 2x & x \leq 2 \end{cases}$$

Rješenje:

Računanje ove vrijednosti nije teško i samo zahtijeva preciznost, strpljenje i uredno napisano rješenje. Prvo ćemo odrediti vrijednost poziva $f(7)$, a zatim ostatak.

$$f(7) = f(5) + 1 = 2 + 1 = 3$$

$$f(5) = f(3) + 1 = 1 + 1 = 2$$

$$f(3) = f(2) - 3 = 4 - 3 = 1$$

$$f(2) = 2 * 2 = 4$$

$$f(f(f(7))) = f(f(3)) = f(1) = 2 * 1 = 2$$

3. Rekurzije

Problem 10.

Neka je zadana rekurzija f . Odredi vrijednost poziva $f(f(f(f(f(4)))))$.

$$f(x) = \begin{cases} f(x-3)+3 & x > 3 \\ 2x+1 & x = 3 \\ x^2+2 & x < 3 \end{cases}$$

Rješenje:

$$\begin{aligned} f(f(f(f(f(4))))) &= f(f(f(f(6)))) = f(f(f(10))) = f(f(12)) = f(16) = 18 \\ f(4) &= f(1) + 3 = 6 \\ f(6) &= f(3) + 3 = 10 \\ f(10) &= f(7) + 3 = f(4) + 6 = 6 + 6 = 12 \\ f(12) &= f(9) + 3 = f(6) + 6 = 10 + 6 = 16 \\ f(16) &= f(13) + 3 = f(10) + 6 = 12 + 6 = 18 \end{aligned}$$

Zadaci za samostalan rad

Riješi sljedeće zadatke, a točnost svojih rješenja provjeri tako što ćeš napisati pripadajuću rekurzivnu funkciju.

1. Odredi vrijednost $f(9)$ ako je:

$$f(x) = \begin{cases} f(x-1)+2-f(x-2) & \text{ako je } x > 5 \\ 2 & \text{ako je } x = 5 \\ -2 & \text{ako je } x < 5 \end{cases}$$

2. Odredi vrijednost $f(8, 14)$ ako je:

$$f(x,y) = \begin{cases} f(x,y)-1 & \text{ako je } x < y \\ f(x-2,y+1)+2 & \text{ako je } x > y \\ (x+y)/2 & \text{ako je } x = y \end{cases}$$

3. Odredi vrijednost $f(12)$ ako je:

$$f(x) = \begin{cases} 2-f(x-3)-3 & \text{ako je } x > 6 \\ f(x+2)+1 & \text{ako je } 4 < x \leq 6 \\ x+4 & \text{ako je } x \leq 4 \end{cases}$$

4. Odredi vrijednost $f(10, 2)$ ako je:

$$f(x,y) = \begin{cases} f(x-2,y+2)+2 & \text{ako je } x > y \\ f(x+1,y-1)-1 & \text{ako je } x = y \\ x-y & \text{ako je } x < y \end{cases}$$

5. Odredi vrijednost $f(16)$ ako je:

$$f(x) = \begin{cases} f(x-3)-f(x-1) & x > 5 \\ 2-x-1 & x \leq 5 \end{cases}$$

6. Odredi vrijednost $f(f(f(13)))$ ako je:

$$f(x) = \begin{cases} f(x-2)+1 & \text{ako je } x > 8 \\ 2-f(x-1)-2 & \text{ako je } 5 \leq x \leq 8 \\ x-1 & \text{ako je } x < 5 \end{cases}$$

7. Odredi vrijednost $f(13, 1)$ ako je:

$$f(x,y) = \begin{cases} f(x-1,y+3)+3 & \text{ako je } x > y \\ x+f(x+4,y-2) & \text{ako je } x = y \\ 2-x+3-y & \text{ako je } x < y \end{cases}$$

8. Odredi vrijednost $f(1, 10)$ ako je:

$$f(x,y) = \begin{cases} f(x+1,y-2)+1 & \text{ako je } x < y \\ 3-x+4-y & \text{ako je } x > y \\ 2-f(x+2,y-2) & \text{ako je } x = y \end{cases}$$

3. Rekurzije

9. Odredi vrijednost $f(16)$ ako je:

$$f(x) = \begin{cases} f(f(x-2))+3 & \text{ako je } x > 10 \\ f(x+1)-1 & \text{ako je } x = 10 \\ x-4 & \text{ako je } x < 10 \end{cases}$$

10. Odredi vrijednost $f(10,3)$ ako je:

$$f(x,y) = \begin{cases} f(x-2,y+1)+x & \text{ako je } x > 5 \\ f(x-3,y+2)+y & \text{ako je } 0 \leq x \leq 5 \\ x^2+x & \text{ako je } x < 0 \end{cases}$$

11. Odredi vrijednost $f(16)$ ako je:

$$f(x) = \begin{cases} f(x-3)-f(x-1) & x > 5 \\ 2-x-1 & x \leq 5 \end{cases}$$

12. Odredi vrijednost $f(f(f(1,0),1),0)$ ako je:

$$f(x,y) = \begin{cases} f(x-2,y+1)+3 & \text{ako je } x > y \\ f(x+1,y-3)-4 & \text{ako je } x = y \\ x+2-y & \text{ako je } x < y \end{cases}$$

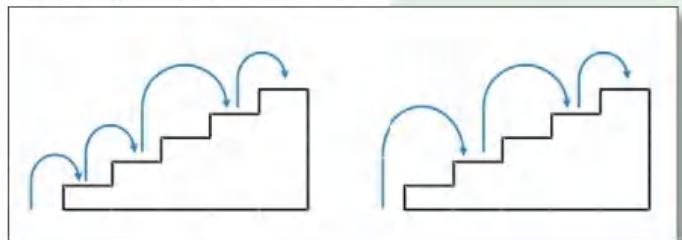
3.3. Pisanje rekurzivnih funkcija

Do sada smo naučili kako se u Pythonu izvršavaju jednostavnije rekurzivne funkcije, za koje je uglavnom bila očita ili unaprijed zadana rekurzivna relacija. Sljedeći korak jest da pokušamo sami pisati rekurzivne funkcije koje rješavaju neki problem. Kroz niz zadataka usvojiti ćemo način razmišljanja potreban da bismo mogli uspješno osmislitи takva rješenja.

Zadatak 1: Preskakivanje stuba

Mirko živi na prvom katu zgrade, a od ulaza u zgradu do njegova stana vodi N stuba. Svakim korakom Mirko se može popeti za po jednu ili za po dvije stube. Na koliko se različitih načina Mirko može popeti od ulaza u zgradu do svojeg stana? Napišite funkciju f koja kao parametar prima broj N, a vraća broj traženih načina.

Na primjer, ako je $N=5$ stuba, onda je jedan način penjanja uz stube takav da Mirko prvo napravi korak kojim priđe jednu stubu, pa nakon toga dva koraka od po dvije stube – taj način možemo zapisati kao $1+2+2$. Svi su mogući načini: $1+1+1+1+1$, $1+1+1+2$, $1+1+2+1$, $1+2+1+1$, $1+2+2$, $2+1+1+1$, $2+1+2$, $2+2+1$, tj. ima ih ukupno osam.



Rješenje:

Kod rekurzivnog rješavanja zadataka uvijek se "veliki" problem svodi na jedan ili više "manjih" problema. Vrlo je važno da su svi manji problemi potpuno **istog tipa** kao i posljazni veliki problem. Također, u svakoj rekurziji postojat će neki trivijalni, vrlo maleni problemi koje ćemo odmah znati rješiti, bez "cjepkanja" na manje probleme.

Prepoznajmo koji su problemi veliki, koji su manji, a koji su trivijalni u našem zadatku. Kada je $N=1$ ili $N=2$, odmah znamo čemu je jednako $f(N)$: $f(1) = 1$, jer je jedini način kako se Mirko može popeti na tu jednu stubu tako da napravi jedan mali korak, i $f(2) = 2$, jer se Mirko može popeti na dvije stube s pomoću dva mala ili s pomoću jednog velikog koraka. Dakle, problem je trivijalan za $N=1$ i $N=2$.

3. Rekurzije

Kako odrediti $f(N)$ za neki $N > 2$? To ćemo smatrati "velikim" problemom i svoditi ćemo ga na manje. Kako? Zapitajmo se što može biti Mirkov prvi korak.

- Ako se Mirko prvim korakom popne za jednu stubu, morat će se nakon toga popeti za još $N-1$ stubu. Na tih preostalih $N-1$ stuba Mirko se može penjati na razno-razne načine. Dakle, moramo ponovno riješiti problem iste vrste (odrediti broj načina penjanja na nekoliko stuba), ali je sada problem manji ($N-1$ stuba umjesto N). Broj načina penjanja na $N-1$ stubu jednak je $f(N-1)$.
- Ako se Mirko prvim korakom popne za dvije stube, morat će se nakon toga popeti za još $N-2$ stube. To može napraviti na $f(N-2)$ načina.

Ukupan broj načina penjanja na N stuba jednak je zbroju $f(N-1) + f(N-2)$, tj. vrijedi formula: $f(N) = f(N-1) + f(N-2)$. Vidimo da smo "veliki" problem određivanja $f(N)$ sveli na dva "manja" problema: određivanje $f(N-1)$ i $f(N-2)$.

Za uspješno rješavanje problema rekurzivne prirode vrlo je važno usvojiti sljedeću logiku 1-2-3:

1. Probleme treba na neki način poredati od manjih prema većima. U zadatku sa stubama to je bilo vrlo jednostavno: što je veći N to je veći problem.
2. Da bismo odredili rješenje nekog konkretnog problema zadane veličine, zamislimo da znamo riješiti apsolutno sve probleme koji su manji od njega. Tada je potrebno taj zadani problem nekako riješiti upotrebom gotovih rješenja manjih problema, odnosno, rješenje zadanog problema izraziti s pomoću rješenja manjih problema. U zadatku sa stubama zadani problem bio je veličine N , a njegovo rješenje $f(N)$ izazili smo s pomoću rješenja manjih problema $f(N-1)$ i $f(N-2)$.
3. Uzastopnom primjenom koraka 2 uvijek moramo u konačnici doći do trivijalnog problema. U zadatku sa stubama $f(N)$ svodi se na $f(N-1)$ i $f(N-2)$, $f(N-1)$ svodi se na $f(N-2)$ i $f(N-3)$, $f(N-2)$ svodi se na $f(N-3)$ i $f(N-4)$, ..., $f(3)$ svodi se na $f(2)$ i $f(1)$, a $f(2)$ i $f(1)$ su trivijalni problemi. Uočite da nije dovoljno samo $f(1)$ proglašiti trivijalnim nego i $f(2)$ – kad $f(2)$ ne bi bio trivijalan, svodio bi se na $f(1)$ i $f(0)$, a $f(0)$ uopće nije definiran.

Sada možemo napokon napisati i vrlo jednostavnu rekurzivnu funkciju koja rješava naš zadatak:

```
def f( N ):  
    if( N == 1 ):                      # Prvi trivijalni problem: N=1.  
        return 1;  
    elif( N == 2 ):                     # Drugi trivijalni problem: N=2.  
        return 2;  
    else:                             # "Veliki" problem svedi na dva manja.  
        return f( N-1 ) + f( N-2 );
```

Uočite: **nije potrebno** razmišljati kako se rekurzija izvršava za manje probleme ($f(N-1)$ i $f(N-2)$) ako smo se pri njezinu osmišljavanju koristili logikom 1-2-3! Samo treba paziti da pokrijemo sve trivijalne slučajeve koji će u konačnici zaustaviti izvođenje rekurzije.

Iz rekurzivne funkcije vidimo da je broj načina penjanja na N stuba zapravo jednak n -tom Fibonaccijevom broju!

 U četvrtom
ćete razredu
iz matemati-
ke učiti dokazi-
vati tvrdnje matema-
tičkom indukcijom.
Logika rješavanja re-
kurzivnih problema
gore opisanim nači-
nom u potpunosti je
identična logici doka-
zivanja matematič-
kom indukcijom!
Dobro razumijevanje
gradiva o rekurzivnim
funkcijama iz infor-
matike stoga će vam
pomoći u razumi-
vanju matematičke
indukcije i obratno.

3. Rekurzije

Zadatak 2: Sparivanje zagrada

Pogledajmo neki aritmetički izraz, na primjer: $((5+3)*(6-(2+6)))*((2+3)*4)$. Kada iz njega obrišemo sve osim zagrada, dobivamo: $((())())()$. Da smo krenuli od nekog drugog aritmetičkog izraza, dobili bismo neki drugi niz zagrada. Postavlja se pitanje: ako imamo N otvorenih i N zatvorenih zagrada, koliko različitih nizova "dobro sparenih" zagrada možemo napraviti? Niz je zagrada "dobro sparen" ako je, čitajući slijeva nadesno, u svakom trenutku broj otvorenih zagrada veći od broja zatvorenih zagrada ili mu je jednak, a ukupan je broj otvorenih i zatvorenih zagrada jednak. Na primjer, niz $((())())()$ nije niz "dobro sparenih" zagrada. Kada je $N=3$, onda postoji pet kombinacija dobro sparenih zagrada: $(((), ()()), (((), ())), (((), ((), ()))))$ i $(((), ((), ())))$.

Rješenje:

Označimo s $f(N)$ ukupan broj nizova "dobro sparenih" zagrada koji se sastoje od N otvorenih i N zatvorenih zagrada. Kada je $N=1$, onda je jedini ispravni niz $()$, pa je $f(1)=1$. Kada je $N=2$, onda imamo dva ispravna niza: $(((),))$ i $(((),))$, pa je $f(2)=2$. Pokušajte sami naći sve ispravne nizove za $N=4$. Što je N veći ispisivanje svih nizova očito postaje sve komplikiranije.

Pokušajmo stoga problem određivanja $f(N)$ svesti na manje probleme. U skladu s logikom 1-2-3 zamislimo da su nam već poznate sve vrijednosti $f(1), f(2), \dots, f(N-1)$; pokušajmo s pomoću njih odrediti $f(N)$. Svaki niz od N otvorenih i N zatvorenih zagrada očito mora započeti nekom otvorenom zagradom. Gdje se ta zagrada zatvara?

Pogledajmo ponovno primjer iz teksta zadatka: u nizu $(((),))()$ crvenom bojom označili smo mjesto na kojem je zatvorena prva zagrada. Svaki niz od N zagrada izgleda ovako: $(X)Y$, pri čemu su X i Y neki nizovi dobro sparenih zagrada. Koliko se zagrada nalazi u X , a koliko u Y ? Ako se u X nalazi K otvorenih i K zatvorenih zagrada, onda se u Y mora nalaziti $N-K-1$ otvorenih i isto toliko zatvorenih zagrada. Takav X možemo napraviti na $f(K)$ načina, a takav Y na $f(N-K-1)$ načina. Ukupno, izraz $(X)Y$ možemo napraviti na $f(K)*f(N-K-1)$ načina jer svaki od $f(K)$ izraza koji daju X možemo kombinirati za svakim od $f(N-K-1)$ izraza koji daju Y . Konačno, primjetimo da K može biti bilo koji od brojeva $0, 1, \dots, N-1$. Kada je $K=0$, onda izraz izgleda ovako $()Y$, a kada je $K=N-1$, onda izraz izgleda ovako: (X) . Ako stavimo $f(0)=1$, onda je u oba slučaja broj kombinacija i dalje $f(K)*f(N-K-1)$.

Dakle, budući da možemo odabrati bilo koji K , formula za ukupan broj izraza glasi:

$$f(N) = f(0)*f(N-1) + f(1)*f(N-2) + f(2)*f(N-3) + \dots + f(N-3)*f(2) + f(N-2)*f(1) + f(N-1)*f(0),$$

uz trivijalne slučajevе $f(0)=1, f(1)=1$.

Sada napokon možemo napisati rekurzivnu funkciju u Pythonu.

```
def f( N ):
    if( N == 0 ):                      # Prvi trivijalni problem: N=0.
        return 1;
    elif( N == 1 ):                     # Drugi trivijalni problem: N=1.
        return 1;
    else:                             # "Veliki" problem svedi na puno manjih.
        ukupno = 0;
        for K in range(0, N):
            ukupno = ukupno + f(K) * f(N-K-1);
        return ukupno;
```

3. Rekurzije

Zadatak 3: Plus i minus

Pero je na papir napisao nekoliko brojeva. Zanima ga je li moguće između tih brojeva napisati znakove + i - tako da je vrijednost dobivenog izraza jednaka nekom zadanim broju X. Na primjer, ako je Pero napisao brojeve: 5 3 6 2 4, a želi dobiti X=2, onda to može napraviti ovako: $5 - 3 + 6 - 2 - 4 = 2$. Trebamo napisati funkciju f koja će primiti broj X i brojeve koje je Pero napisao u listi L pa vraćati True ako je X moguće prikazati s pomoću brojeva iz liste, a False ako to nije moguće.

Rješenje:

Da bismo riješili zadatak, ponovno slijedimo logiku 1-2-3. Razmislimo kako poredati probleme po veličini i koji su problemi trivijalni. Očito, ako se lista L sastoji samo od jednog broja $L=[L_1]$, onda naša funkcija vraća True ako je $L_1=X$, a inače False. Čim se lista sastoji od dvaju ili više brojeva, problem trebamo svesti na manje probleme.

Pretpostavimo da se lista sastoji od N brojeva: $L=[L_1, L_2, \dots, L_N]$. U skladu s točkom 2, zamislimo da znamo riješiti zadatak za sve liste s manje od N brojeva i za svaki X. Trebamo nekako s pomoću tih rješenja napisati funkciju f(L, X). Koji ćemo predznak staviti ispred broja L_N ? Ako stavimo znak plus, onda izraz izgleda ovako:

$$L_1 ? L_2 ? L_3 ? \dots ? L_{N-1} + L_N = X,$$

pa preostaje staviti predznake između brojeva L_1, L_2, \dots, L_{N-1} tako da zbroj bude $X-L_N$:

$$L_1 ? L_2 ? L_3 ? \dots ? L_{N-1} = X-L_N.$$

Mogu li se upitnici zamijeniti s + i – tako da zbroj bude $X-L_N$? Odgovor na to pitanje dat će nam poziv funkcije $f(L[:-1], X-L[-1])$. (Sjetimo se: $L[-1]$ zadnji je element liste, a $L[:-1]$ lista svih elemenata osim zadnjeg).

Naravno, ispred L_N mogli smo staviti i minus:

$$L_1 ? L_2 ? L_3 ? \dots ? L_{N-1} - L_N = X,$$

pa preostaje staviti predznake između brojeva L_1, L_2, \dots, L_{N-1} tako da zbroj bude $X+L_N$:

$$L_1 ? L_2 ? L_3 ? \dots ? L_{N-1} = X+L_N.$$

To se može napraviti ako poziv funkcije $f(L[:-1], X-L[-1])$ vrati True. Dakle, $f(L, X)$ treba vratiti True ako bilo koji od poziva $f(L[:-1], X-L[-1])$ i $f(L[:-1], X+L[-1])$ vrati True, a treba vratiti False samo ako oba ta poziva vrate False. Program koji rješava ovaj naizgled dosta komplikiran problem na kraju je vrlo kratak!

```
def f( L, X ):
    if( len(L) == 1 ):          # Trivijalni problem: lista ima samo 1 element.
        if( L[0] == X ):
            return True;
        else:
            return False;
    else:                      # "Veliki" problem svedi na dva manja.
        return f(L[:-1], X-L[-1]) or f(L[:-1], X+L[-1]);
```

(nastavak na idućoj stranici)

3. Rekurzije

```
print( '2 ? 3 = 7 : ', f([2, 3], 7) );
print( '2 + 3 = 5 : ', f([2, 3], 5) );
print( '2 - 3 + 5 = 4 : ', f([2, 3, 5], 4) );
print( '6 - 2 + 3 + 1 = 8 : ', f([6, 2, 3, 1], 8) );
print( '6 ? 2 ? 3 ? 1 = 7 : ', f([6, 2, 3, 1], 7) );
```

Rekurzivno ispisivanje svih kombinacija

U svakom od zadataka iz prethodne celine mogli bismo poželjeti ispisati sve kombinacije koje daju rezultat: sve načine za preskakivanje stuba, sve moguće nizove dobro sparenih zagrada i sve moguće rasporede operacija plus i minus koje će kao rezultat dati X. Sada ćemo vidjeti kako se to može napraviti.

Zadatak 4: Nizovi binarnih znamenki

Svi nizovi binarnih znamenki duljine 3 jesu: 000, 001, 010, 011, 100, 101, 110, 111. Trebamo napisati funkciju $f(N)$ koja će ispisati svih 2^N nizova binarnih znamenki duljine N.

Rješenje:

Ponovno možemo primijeniti svoju logiku 1-2-3. Trivijalno je ispisati nizove duljine 1: to su jednostavno 0 i 1. Svaki niz duljine N izgleda ovako:

0X

ili ovako:

1X,

pri čemu je X bilo koji niz duljine N-1. Dakle, odabrat ćemo ili znamenku 0 ili znamenku 1 kao prvu znamenku i pozvati $f(N-1)$ koja zna napraviti sve nizove duljine N-1.

Osim parametra N funkciji f sada trebamo slati i jednu listu L u koju ćemo "priklupljati" znamenke kako ih generiramo. Na početku je lista L prazna, a svaki put kada se odlučimo za znamenku 0 ili 1 dodat ćemo je na kraj liste pa tako proširenu listu poslati u rekurzivni poziv. Uočite da ispis u rekurziji radimo tek kad dođemo do $N=0$, što znači da niz koji smo gore označili s X treba imati 0 znamenki (tj. ne treba dodati još znamenki u listu L). Provjerite što se dogodi kada ispis radimo odmah na ulasku u funkciju!

```
def f( N, L ):
    if( N == 0 ):                                # Trivijalni problem: ne treba dodavati
                                                # znamenke, nego samo ispisati.
        for i in L:
            print( i, end='' );
        print();
    else:                                         # "Veliki" problem svedi na dva manja.
        f( N-1, L + [0] );                      # Prvo napravi sve liste koje počinju s 0.
        f( N-1, L + [1] );                      # Zatim napravi sve liste koje počinju s 1.

f(4, []);
```

3. Rekurzije

Zadatak 5: Anagrami

Ana ima košaru u kojoj se nalazi N papirića. Na svakom je papiriću napisano po jedno slovo. Ana bilo kojim redom izvlači papirice iz košare i slaže ih jedan do drugog tako da

čine neku riječ sve dok ne izvuče sve papirice iz košare. Napišite funkciju $f(L)$ koja prima listu L u kojoj se nalaze sve slova na papirićima u košari i ispisuje sve moguće riječi koje Ana može dobiti izvlačenjem papirića. Na primjer, ako su na papirićima slova A, N i E, onda Ana može dobiti riječi ANE, AEN, NAE, NEA, EAN i ENA.



Jedni od najpoznatijih anagrama načinjenih od imena poznatih ličnosti:

Elvis – Lives

Clint Eastwood – Old West action

Madame Curie – Me, Radium ace

A Homer Simpson – Mr Homo

Sapiens

Rješenje:

Jedina bitna razlika u ovom zadatku u odnosu na prethodni zadatak jest da trebamo izbjegavati višestruka ponavljanja slova koja smo već iskoristili. Opisat ćemo dva načina kako to možemo napraviti.

Problem ispisivanja trivijalan je ako je lista L prazna. Ako u L imamo jedno slovo ili više njih, onda je riječ koju trebamo napraviti oblika

aX,

pri čemu je a bilo koje slovo iz liste L , a X bilo koja riječ sastavljena od slova iz liste L iz koje izbacimo slovo a. Rekurzivni poziv generirat će nam sve moguće riječi X. Ponovno kao parametar funkcije dodajemo string s u koji spremamo slova redom kojim ih dodajemo u riječ.

```
def f( L, s ):  
    if( len(L) == 0 ):  
        # Trivijalni slučaj: ispraznili smo listu, treba samo  
        # ispisati string.  
        print( s );  
    else:  
        for i in range( 0, len(L) ):  
            Lnova = L[:];  
            del( Lnova[i] );  
            # Napravi novu kopiju liste L.  
            # Iz kopije izbaci i-ti znak.  
            f( Lnova, s + L[i] );  
            # Pozovi rekurziju za manju listu i string s kojem  
            # na kraj dodaj taj i-ti znak.  
  
    f( [ 'A', 'N', 'E' ], '' );
```

Drugi je način da imamo pomoćno polje b u kojem označavamo koja smo sve slova iz liste L iskoristili, a koja ne. Kada se odlučimo za neko slovo, u tom pomoćnom polju označimo s 1 da smo to slovo "potrošili", tako da ga ne bismo ponovno iskoristili unutar rekurzivnog poziva. Kada se vratimo iz rekurzivnog poziva (tj. ispišemo sve riječi koje počinju tim slovom), to slovo ponovno označimo s 0, odnosno kao neiskorišteno. To pomoćno polje možemo slati kao dodatni parametar funkcije f ili ga upotrebljavati kao glo-

3. Rekurzije

balnu varijablu. Za ilustraciju su i to polje i string koji gradimo u donjem programu implementirani kao globalne varijable.

```
def f( L ):
    global bio, s;

    if( len(L) == len(s) ):
        print( s );
    else:

        for i in range( 0, len(L) ):
            if( not bio[i] ):
                bio[i] = True;
                s = s + L[i];
                f( L );
                s = s[:-1];

L = [ 'A' , 'N' , 'E' ];
bio = [False]*len(L);
s = '';
f( L );
```

Trivijalni problem: u stringu se već nalaze sva slova iz L.
"Veliki" problem svedi na nekoliko manjih.
Slovo L[i] još uvijek nismo prebacili u string.
Sad ćemo.
Prvo označimo da smo to slovo iskoristili.
Dodamo ga na kraj stringa s.
Pozovemo rekurziju.
Vratili smo se iz rekurzije, pa uklonimo L[i] s kraja stringa.
Polje u kojem pamtimo koje smo slovo već iskoristili. Na početku nismo nijedno.
String u koji prikupljamo slova. Na početku je prazan.

To je rješenje nešto komplikiranije, ali dosta efikasnije. Uočite da i L može biti globalna varijabla. Što bi se dogodilo da nakon povratka iz rekurzije ne uklonimo L[i] s kraja stringa s ili ne dopustimo ponovnu upotrebu L[i]?

Zadatak 6: Plus i minus (ponovo)

Riješimo ponovno Zadatak 3, ali tako da sada i ispišemo sve moguće izraze koji u zbroju daju broj X.

Rješenje:

Treba malo modificirati rješenje koje smo ranije napisali. Podsetimo se, u prvom koraku mogli smo kao zadnji znak staviti plus pa smo imali:

$$L_1 ? L_2 ? L_3 ? \dots ? L_{N-1} + L_N = X,$$

ili minus, što je vodilo na

$$L_1 ? L_2 ? L_3 ? \dots ? L_{N-1} - L_N = X.$$



Ako se na papirićima više puta javlja isto slovo, onda će nam neke riječi biti ispisane više puta. Pokušajte modificirati program tako da ispiše svaku riječ točno jednom!

3. Rekurzije

I dalje ćemo imati identične rekurzivne pozive kao ranije, ali ćemo sada imati još jednu listu Z u koju ćemo spremiti predznak za koji smo se odlučili: u prvom ćemo slučaju na početak liste Z staviti znak plus, a u drugom slučaju znak minus. Kada se rekurzija spusti do trivijalnog slučaja u kojem je lista duljine 1 i ako se broj u toj listi podudara s brojem X, onda ćemo ispisati sve brojeve iz originalne liste isprepletene sa zapamćenim predznacima iz Z.

```
def f( L, X, Z ):
    global LL, XX;
    if( len(L) == 1 ):
        if( L[0] == X ):
            for i in range(0, len(Z)):
                print( LL[i], Z[i], end=' ' );
                print( LL[-1], '=', XX );
        else:
            f( L[:-1], X-L[-1], [ '+' ] + Z );
            f( L[:-1], X+L[-1], [ '-' ] + Z );
    LL = [2, 3]; XX = 7; f(LL, XX, []);
    LL = [2, 3]; XX = 5; f(LL, XX, []);
    LL = [2, 3, 5]; XX = 4; f(LL, XX, []);
    LL = [6, 2, 3, 1]; XX = 8; f(LL, XX, []);
    LL = [6, 2, 3, 1]; XX = 7; f(LL, XX, []);
    LL = [1, 1, 1, 1]; XX = 2; f(LL, XX, []);

# Ovo su originalne vrijednosti L i X iz glavnog
# programa.
# Trivijalni problem: lista ima samo 1 element.
# Izbor je predznaka spremljenih u Z ispravan,
# ispisujemo.
# "Veliki" problem svedi na dva manja.
# Prvo ispiši sve kombinacije u kojima smo
# za zadnji predznak uzeli +.
# Onda ispiši sve kombinacije u kojima smo
# za zadnji predznak uzeli -.

# 0 načina
# 1 način: 2+3=5
# 1 način: 2-3+5=4
# 1 način: 6-2+3+1=8
# 0 načina
# 3 načina: 1-1+1+1=2,
# 1+1-1+1=2, 1+1+1-1=2
```

Unutar rekurzivnih poziva mijenjamo L i X, pa nam originalni sadržaj tih varijabli ne bi bio dostupan u trenutku ispisa – zato smo uveli globalne varijable LL i XX.

Ubrzavanje rekurzije - tehnika “memoizacija”

Podsjetimo se rekurzije za izračunavanje n-tog Fibonaccijeva broja:

```
import time;

def f( N ):
    if( N == 1 ):
        return 1;
    elif( N == 2 ):
        return 2;
    else:
        return f( N-1 ) + f( N-2 );

start = time.time(); print( f(37) ); end = time.time();
print( 'Proteklo vrijeme: ', end - start );
```

3. Rekurzije

Koliko dugo traje računanje 37. Fibonaccijeva broja? Jako dugo, na našem računalu čak 7.5 sekundi! Zbog čega to računanje traje tako dugo? Dodajmo u rekurziju jednu globalnu varijablu koja će brojati koliko se puta pozove funkcija f:

```
def f( N ):
    global broj_poziva;
    broj_poziva = broj_poziva + 1;

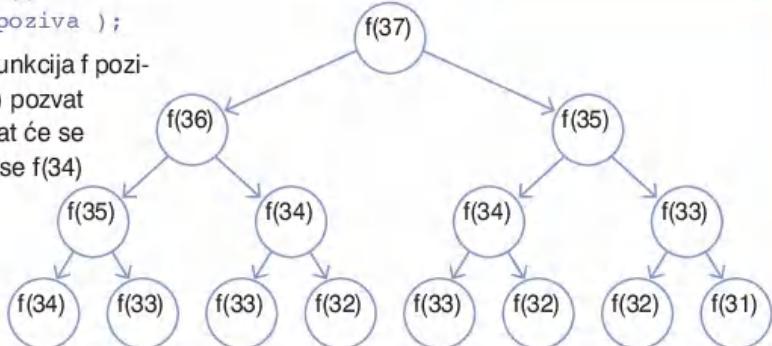
    if( N == 1 ):
        return 1;
    elif( N == 2 ):
        return 2;

    else:
        return f( N-1 ) + f( N-2 );

broj_poziva = 0; print( f(37) );
print( 'Broj poziva: ', broj_poziva );
```

Broj je poziva čak 48315633! Zbog čega se funkcija f poziva toliko puno puta? Prilikom računanja f(37) pozvat će se i f(36) i f(35). Kod računanja f(36) pozvat će se f(35) i f(34), a kod računanja f(35) pozvat će se f(34) i f(33). Imamo sljedeću sliku pozivanja:

Vidimo da se jedan te isti broj računa puno puta! Na primjer, f(35) računa se dva puta, f(34) računa se tri puta, a f(33) računa se pet puta. Probajte odrediti koliko će se puta računati f(1). Nema nikakve potrebe da računamo isti broj više puta – jednom kad izračunamo f(K), zapamtit ćemo ga u jednom pomoćnom globalnom polju. Svaki idući put kad nam zatreba računanje f(K), najprije ćemo u globalnom polju provjeriti je li već izračunat. Ako jest, nećemo pozivati rekurziju, nego ćemo samo vratiti već izračunatu vrijednost spremljenu u polju. To će jako ubrzati izračunavanje!



```
import time;

def f( N ):
    global MEMO;
    if( MEMO[N] != -1 ):
        return MEMO[N];
    if( N == 1 ):
        MEMO[N] = 1; return MEMO[N];
    elif( N == 2 ):
        MEMO[N] = 2; return MEMO[N];
    else:
        MEMO[N] = f( N-1 ) + f( N-2 ); return MEMO[N];

# Ako u polju MEMO[N] ne piše -1, to
# znači da smo ranije već izračunali
# f(N).

# Izračunamo f(N) i zapamtimo ga u
# polju MEMO.
```

(nastavak na idućoj stranici)

3. Rekurzije

```
N = 37; MEMO = [-1] * (N+1);  
# Na početku MEMO sadrži (N+1)  
# element -1.  
  
start = time.time(); print( f(N) ); end = time.time();  
print( 'Proteklo vrijeme: ', end - start );
```

Memoizacija

Sada računanje traje toliko kratko da ga ne možemo precizno izmjeriti.

Ta tehnika pamćenja i ponovne upotrebe jednom izračunatih rezultata zove se **memoizacija**. Ona se može primijeniti samo ako je ukupan broj različitih vrijednosti za koje će rekurzija ikad biti pozvana dovoljno malen da se sve te vrijednosti mogu pospremiti u globalno polje. Kod Fibonaccijevih brojeva prilikom računanja $f(37)$ rekurzija će biti pozivana samo za $f(37), f(36), \dots, f(2), f(1)$, dakle, samo za 37 različitih vrijednosti koje smo sve mogli pospremiti u polje MEMO.

Zadatak 7: Miš i sir



Miš se nalazi u gornjem lijevom kutu pravokutne ploče koja se sastoji od $R \times S$ polja. Na svakom polju nalazi se neki broj komadića sira. Miš se u svakom koraku pomiče za jedno polje prema desno ili za jedno polje prema dolje. Čim se nađe na nekom polju, miš pojede sav sir koji se na njemu nalazi. Potrebno je odrediti koliko najviše komadića sira miš može pojesti tako da završi u donjem desnom kutu ploče. Količina sira na svakom polju dostupna je u matrici SIR. Na slici je primjer jedne matrice SIR. Miš će pojести 33 komadića sira ako ide putem označenim plavom bojom. To je najveći broj komada sira koje miš može pojesti.

4	2	3	4	1
5	4	5	1	4
3	2	2	3	2
1	3	5	5	1
2	4	3	2	1

Rješenje:

Označimo s $f(r, s)$ maksimalnu količinu sira koju miš može pojesti tako da završi na polju u retku r i stupcu s . Dakle, trebamo odrediti $f(r, s)$. Ponovno primjenjujemo logiku 1-2-3. Probleme po veličini poredamo ovako: ako je $r_1 \leq r_2$ i $s_1 \leq s_2$, onda je problem određivanja $f(r_1, s_1)$ lakši od određivanja $f(r_2, s_2)$. Jedini trivijalni problem jest kad miš stoji u gornjem lijevom kutu: $f(0,0) = \text{SIR}[0][0]$.

Kako $f(r, s)$ svesti na manje probleme? Tako da promotrimo koji je bio zadnji korak koji je miša doveo na polje (r, s) . Postoje samo dva načina kako je on mogao doći na to polje: ili s polja $(r-1, s)$ ili s polja $(r, s-1)$. Ako je došao s prvog polja, do tada je maksimalno mogao pojesti $f(r-1, s)$ komada sira, a ako je došao s drugog polja, onda je maksimalno mogao pojesti $f(r, s-1)$ komada sira. Naravno, bolje je da dođe s onog polja s kojeg je mogao pojesti više sira:

$$f(r, s) = \max(f(r-1, s), f(r, s-1)) + \text{SIR}[r][s].$$

Dodajemo $\text{SIR}[r][s]$ jer će tu količinu miš pojesti nakon dolaska na polje (r, s) . U gornjoj formuli treba još paziti da miš nije u prvom retku ($r=0$) ili u prvom stupcu ($s=0$) jer tada na polje (r, s) može doći samo iz jednog smjera.

3. Rekurzije

Napišimo prvo "običnu" rekurziju koja rješava ovaj zadatak pa je ubrzajmo tehnikom memoizacije.

```
def f( r, s ):
    global SIR;

    if( r == 0 and s == 0 ):
        return SIR[r][s];

    maxi = -1;
    if( r > 0 ):

        maxi = max( maxi, f( r-1, s ) );
    if( s > 0 ):

        maxi = max( maxi, f( r, s-1 ) );
    return maxi + SIR[r][s];

SIR = [
    [4, 2, 3, 4, 1],
    [5, 4, 5, 1, 4],
    [3, 2, 2, 3, 2],
    [1, 3, 5, 5, 1],
    [2, 4, 3, 2, 1]
];
R = len(SIR); S = len(SIR[0]);
print( 'Mis moze pojesti najvise', f(R-1, S-1), 'komada sira.' );
```

Trivijalni problem:
miš je u početnom polju.

Veliki problem svodimo na najviše dva manja.

Miš je u (r, s) mogao stići s polja (r-1, s).

Miš je u (r, s) mogao stići s polja (r, s-1).

Ubrzanje tehnikom memoizacije vrlo je slično onom za Fibonaccijeve brojeve:

```
def f( r, s ):
    global SIR, MEMO;

    if( MEMO[r][s] != -1 ):
        return MEMO[r][s];

    if( r == 0 and s == 0 ):
        MEMO[r][s] = SIR[r][s];
        return MEMO[r][s];

    maxi = -1;
    if( r > 0 ):
        maxi = max( maxi, f( r-1, s ) );
    if( s > 0 ):
        maxi = max( maxi, f( r, s-1 ) );
    # Ako smo već izračunali f(r, s),
    # to će pisati u MEMO.
```

Trivijalni problem: miš je u početnom polju.

Veliki problem svodimo na najviše dva manja.

Miš je u (r, s) mogao stići s polja (r-1, s).

Miš je u (r, s) mogao stići s polja (r, s-1).

(nastavak na idućoj stranici)

3. Rekurzije

```
MEMO[r][s] = maxi + SIR[r][s];
return MEMO[r][s];

SIR = [
    [4, 2, 3, 4, 1],
    [5, 4, 5, 1, 4],
    [3, 2, 2, 3, 2],
    [1, 3, 5, 5, 1],
    [2, 4, 3, 2, 1]
];

R = len(SIR); S = len(SIR[0]);
MEMO = [[-1 for s in range(S)] for r in range(R)]

print( 'Mis moze pojesti najvise', f(R-1, S-1), 'komada sira.' );
```

Zapamti izračunatu vrijednost $f(r, s)$ u tablici MEMO.

U tablici $MEMO[r][s]=-1$ označava da $f(r, s)$ još nije izračunat.

Usporedite brzinu izvršavanja za tablicu SIR iz teksta zadatka i za neku malo veću tablicu.

Pretraživanje u dubinu (DFS) i u širinu (BFS)

Pogledajmo još jedan zadatak koji ima rekursivno rješenje.

Zadatak 8: Bojanje

DFS i BFS su dva najčešća algoritma koja se pojavljuju na natjecanjima iz programiranja. Iako zanimljivi, nisu dio redovnog nastavnog gradiva.

Na pravokutnoj ploči na nekim poljima nalazi se znak 'x', a na nekima znak '..'. Polja na kojima je 'x' smatramo "obojenima", a polja na kojima je '..' prazna su. Pero je odlučio obojiti područje na ploči koje je obrubljeno znakovima 'x'. Napišite funkciju oboji koja prima brojeve r i s koji predstavljaju redak i stupac jednog praznog polja koje se nalazi unutar područja koje treba obojiti. Raspored znakova na ploči nalazi se u globalnoj varijabli PLOCA.

.	.	.	x	x	x	x	x	x	x	x	.	.	.
.	.	x	x	.	.
.	x	x	.	.
.	x	x	x	x	.	.	x	.	.
.	.	x	.	.	.	x	.	.	x	.	.	x	.
.	.	x	.	.	.	x	.	.	x	.	x	.	.
.	.	.	x	x	x	.	.	.	x	x	.	.	.

.	.	.	x	x	x	x	x	x	x	x	x	x	.	.
.	.	x	x	x	x	x	x	x	x	x	x	x	.	.
.	x	x	x	x	x	x	x	x	x	x	x	x	.	.
.	x	x	x	x	x	x	x	x	x	x	x	x	.	.
.	.	x	x	x	x	x	x	x	x	x	x	x	.	.
.	.	x	x	x	x	x	x	x	x	x	x	x	.	.
.	.	x	x	x	x	x	x	x	x	x	x	x	.	.

Rješenje:

Na slici lijevo primjer je jedne neobojene ploče. Plavo je označeno prazno polje unutar područja koje treba obojiti. Desno je obojena ploča. Prvo ćemo napisati klasično rekursivno rješenje u skladu s logikom 1-2-3. Problem koji rješavamo jest bojanje ploče A za koju nam je zadano jedno prazno polje (r, s) unutar područja. Kako ćemo poredati probleme bojanja ploča prema težini? Bojanje ploče A veći je problem od bojanja ploče B

3. Rekurzije

ako su na ploči B već obojena sva polja kao na ploči A i još neka. Trivijalni je problem kad su obojena sva polja unutar područja.

Veliki problem svodimo na manje na sljedeći način: ako imamo ploču A i jedno prazno polje (r, s) unutar područja, onda ćemo obojiti polje (r, s) i tako dobiti novu ploču B. U toj su novoj ploči zbog bojanja polja (r, s) mogla nastati najviše četiri nova područja: jedno "gore", kojem je prazno polje (r-1, s), jedno "dolje", kojem je prazno polje (r+1, s), jedno "lijevo" s praznim poljem (r, s-1) i jedno "desno" s praznim poljem (r+1, s). Zato ćemo imati četiri rekursivna poziva koja će bojati ta četiri područja. Nema veze ako nastane manje od četiriju područja, tada će npr. prvi rekursivni poziv obojiti i "područje" gore i "područje" lijevo.

```
def oboji( r, s ):
    global PLOCA;

    PLOCA[r][s] = 'x';
    print( ''.join( PLOCA[r] ) ); # Oboji prazno polje na mjestu (r, s).

    if( r-1 >= 0 and PLOCA[r-1][s] == '.' ):
        oboji( r-1, s ); # Rekursivno oboji nova četiri područja koja su mogla nastati.

    if( r+1 < len(PLOCA) and PLOCA[r+1][s] == '.' ):
        oboji( r+1, s );

    if( s-1 >= 0 and PLOCA[r][s-1] == '.' ):
        oboji( r, s-1 );

    if( s+1 <= len(PLOCA[0]) and PLOCA[r][s+1] == '.' ):
        oboji( r, s+1 );

PLOCA = [
    list('...xxxxxxxx...'),
    list('..x.....x..'),
    list('.x.....x..'),
    list('.x....xxx...x.),
    list(..x...x...x...x.),
    list(..x...x...xx.x.),
    list(...xxx.....xx.),
];
]

oboji( 4, 12 );

for r in range(len(PLOCA)):
    print( ''.join( PLOCA[r] ) );
```

3. Rekurzije

Kojim će redom polja biti obojena? U donjoj je tablici na svakom polju naveden redni broj kojim je to polje bilo obojeno.

.	.	.	x	x	x	x	x	x	x	x	.	.	.
.	.	x	32	23	22	17	16	13	12	6	5	x	.
.	x	33	31	24	21	18	15	14	11	7	4	3	x
.	x	34	30	25	20	19	x	x	x	8	9	2	x
.	.	x	29	26	36	x	.	.	.	x	10	1	38
.	.	x	28	27	35	x	.	.	.	x	x	37	x
.	.	.	x	x	x	x	x	.	.

stog

Sada ćemo rekurzivnu funkciju pretvoriti u nerekurzivnu koja će bojati potpuno istim redoslijedom. Za to će nam trebati jedna jednostavna struktura podataka koju zovemo **stog** (engl. **stack**).

LIFO

Stog je lista na kojoj su dopušteno samo dvije operacije:

- dodavanje na kraj liste s pomoću `STACK.append(element)`;
- brisanje i dohvaćanje elementa s kraja liste s pomoću `x = STACK.pop()`;

Stog je tzv. LIFO (*last-in-first-out*) struktura: podatak koji zadnji stavimo na stog prvi će biti dohvaćen i obrisan sa stoga.

Umjesto rekurzivnog poziva parametre koje šaljemo rekurziji stavit ćemo na stog. Sav kod unutar funkcije stavit ćemo unutar petlje koja se izvršava sve dok stog nije prazan, a nakon ulaska u petlju samo ćemo dohvatiti element s vrha stoga i spremiti ga u variable `r` i `s`. Time smo dobili ovo rješenje, koje radi identično kao rekurzivno:

```
def oboji( r, s ):  
    global PLOCA;  
  
    STACK = [ ];  
    STACK.append( [r, s] );  
  
    while( len(STACK) != 0 ):  
        [r, s] = STACK.pop();  
  
        PLOCA[r][s] = 'x';  
  
        if( r-1 >= 0 and PLOCA[r-1][s] == '.' ):  
            STACK.append( [r-1, s] );  
  
        if( r+1 < len(PLOCA) and PLOCA[r+1][s] == '.' ):  
            STACK.append( [r+1, s] );  
  
        if( s-1 >= 0 and PLOCA[r][s-1] == '.' ):  
            STACK.append( [r, s-1] );  
  
    # Napravimo stog na  
    # kojem se nalazi samo  
    # par (r, s).  
  
    # Oboji prazno polje na  
    # mjestu (r, s).  
    # "Rekurzivno" oboji  
    # nova četiri područja  
    # koja su mogla nastati.
```

(nastavak na idućoj stranici)

3. Rekurzije

```
if( s+1 <= len(PLOCA[0]) and PLOCA[r][s+1] == '.' ):
    STACK.append( [r, s+1] );

PLOCA = [
    list('...xxxxxxxx...'),
    list('..x.....x...'),
    list('.x.....x...'),
    list('.x.....xxx...x.'),
    list('..x....x...x...x'),
    list('..x....x...xx.x.'),
    list('...xxx.....xx.'),
];
oboji( 4, 12 );

for r in range(len(PLOCA)):
    print( ''.join( PLOCA[r] ) );
```

Uvjerite se da ovo rješenje zaista boja točno istim redoslijedom kao i rekurzivno. Upotrebom stoga svaka se rekurzivna funkcija može na taj način pretvoriti u nerekurzivnu!

Prikazani način obilaženja praznih polja (bilo rekurzivno, bilo s pomoću stoga) zovemo **pretraživanje u dubinu** (engl. *depth first search*, DFS). Razlog je taj što polja obilazimo tako da se “zaletimo” do kud god ide u jednom smjeru (u našoj funkciji, prema gore), pa tek onda gledamo ostale smjerove.

Pogledajmo što bi se dogodilo da u svojoj funkciji umjesto stoga upotrijebimo **red** (engl. **queue**). Red je lista na kojoj su dopuštene samo dvije operacije:

- dodavanje na kraj liste s pomoću `QUEUE.append(element)`;
- brisanje i dohvatanje elementa s početka liste s pomoću `x = QUEUE.popleft()`;

Zbog efikasnosti se nećemo koristiti običnom Pythonovom listom i sporom operacijom `x = QUEUE.pop(0)`, nego operacijom `deque` koja podržava brzu operaciju `popleft`. Red je tzv. **FIFO (first-in-first-out)** struktura: podatak koji prvi stavimo na red prvi će biti dohvaćen i obrisan iz reda.

DFS

red lista

FIFO

```
from collections import deque;

def oboji( r, s ):
    global PLOCA;

    QUEUE = deque( [] );
    QUEUE.append( [r, s] );                                # Napravimo red na kojem se nalazi
                                                          # samo par (r, s).

    while( len(QUEUE) != 0 ):
        [r, s] = QUEUE.popleft();
```

(nastavak na idućoj stranici)

3. Rekurzije

```
PLOCA[r][s] = 'x';                                # Oboji prazno polje na  
if( r-1 >= 0 and PLOCA[r-1][s] == '.' ):          mjestu (r, s).  
    QUEUE.append( [r-1, s] );                      # Oboji nova četiri  
if( r+1 < len(PLOCA) and PLOCA[r+1][s] == '.' ): područja koja su  
    QUEUE.append( [r+1, s] );                      mogla nastati.  
if( s-1 >= 0 and PLOCA[r][s-1] == '.' ):  
    QUEUE.append( [r, s-1] );  
if( s+1 <= len(PLOCA[0]) and PLOCA[r][s+1] == '.' ):  
    QUEUE.append( [r, s+1] );  
  
PLOCA = [  
list('...xxxxxxxx...'),  
list('..x.....x..'),  
list('.x.....x.'),  
list('.x....xxx...x.'),  
list('..x...x...x...x.'),  
list('..x...x...xx.x.'),  
list('...xxx.....xx.'),  
];  
  
oboji( 4, 12 );  
  
for r in range(len(PLOCA)):  
    print( ''.join( PLOCA[r] ) );
```

I ta je funkcija uspjela obojiti traženo područje, ali je sada redoslijed kojim su polja obojena drugačiji:

.	.	.	x	x	x	x	x	x	x	x	.	.	.	
.	.	x	30	26	23	20	18	16	14	12	10	x	.	.
.	x	34	29	25	22	19	17	15	13	11	8	6	x	.
.	x	37	33	28	24	21	x	x	x	9	7	2	x	.
.	.	x	36	32	27	x	.	.	.	x	4	1	5	x
.	.	x	38	35	31	x	.	.	.	x	x	3	x	.
.	.	.	x	x	x	x	x	.	.	.

BFS

Takov način obilaženja praznih polja s pomoću reda zovemo **pretraživanje u širinu** (engl. *breadth first search*, BFS). Razlog je taj što polja obilazimo tako da prvo obojimo početno polje, pa sve susjede tog početnog polja, pa sve susjede svih susjeda tog početnog polja i tako dalje. Kao da smo prolili broju na plavo polje pa se ona dalje širi u koncentričnim krugovima sa središtem u tom plavom polju.

3. Rekurzije

Da sumiramo: pretraživanje u dubinu (DFS) ima ovakvu opću strukturu:

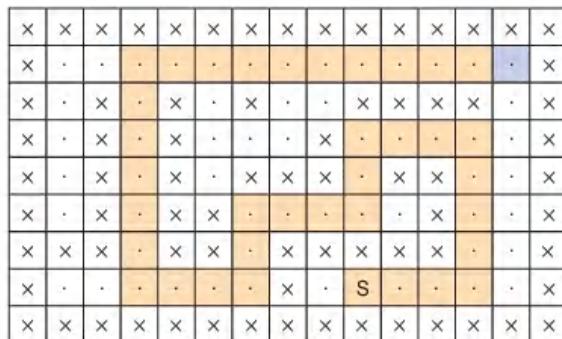
```
stavi početnu poziciju na stog;
sve dok se stog ne isprazni:
    dohvati poziciju P s vrha stoga;
    označi poziciju P kao posjećenu;
    za svaku susjednu poziciju S pozicije P:
        ako još nisi posjetio S:
            stavi poziciju S na stog;
```

Pretraživanje u širinu (BFS) ima potpuno istu strukturu, samo stog zamijenimo redom.

Ako želimo obići sve pozicije ili samo utvrditi je li moguće doći od početne do neke krajnje pozicije, to možemo napraviti i s pomoću DFS-a i s pomoću BFS-a. Velika je prednost BFS-a što s pomoću njega možemo odrediti i najkraći mogući put do krajnje pozicije.

Zadatak 9: Labirint

Zli Gargamel stavio je štrumpfa Hrgu i Štrumpfetu u labirint. Labirint je zapisan kao u pretvodnom zadatku: ‘.’ označava prazno i prohodno polja, a ‘x’ označava zid. Štrumpf Hrga na početku je u polju s koordinatama (r, s), a u polju u kojem je Štrumpfeta piše znak ‘S’. Može li Hrga doći do Štrumpfete? Koliko mu je najmanje koraka za to potrebno?



Rješenje:

Na slici je jedan primjer labirinta. Hrga je na početku u plavom polju, a najkraći put koji ga vodi do Štrumpfete dug je 36 koraka i označen je narančasto.

Slijedimo logiku pretraživanja u širinu, kao u zadatku s bojanjem. Polja ploče stavljamo u red, a zajedno s retkom i stupcem pamtimo i koliko je Hrgi trebalo koraka da dođe do tog polja. Jasno je da će mu do svakog polja koje je susjedno nekom polju (r, s) trebati jedan korak više nego što mu je trebalo do polja (r, s).

```
from collections import deque;
def labirint( r, s ):
    global PLOCA, MIN_KORAKA;
```

(nastavak na idućoj stranici)

3. Rekurzije

```
QUEUE = deque( [] );
QUEUE.append( [r, s, 0] );

while( len(QUEUE) != 0 ):
    [r, s, broj_koraka] = QUEUE.popleft();

    MIN_KORAKA[r][s] = broj_koraka;

    if( PLOCA[r][s] == 'S' ):
        return broj_koraka;

    if( r-1 >= 0 and PLOCA[r-1][s] != 'x' and MIN_KORAKA[r-1][s] == -1 ):
        QUEUE.append( [r-1, s, broj_koraka+1] );

    if( r+1 < len(PLOCA) and PLOCA[r+1][s] != 'x' and MIN_KORAKA[r+1][s] == -1 ):
        QUEUE.append( [r+1, s, broj_koraka+1] );

    if( s-1 >= 0 and PLOCA[r][s-1] != 'x' and MIN_KORAKA[r][s-1] == -1 ):
        QUEUE.append( [r, s-1, broj_koraka+1] );

    if( s+1 <= len(PLOCA[0]) and PLOCA[r][s+1] != 'x' and MIN_KORAKA[r][s+1] == -1 ):
        QUEUE.append( [r, s+1, broj_koraka+1] );

return -1;
PLOCA = [
list('xxxxxxxxxxxxxxxx'),
list('x.....x'),
list('x.x.x.x...xxxx.x'),
list('x.x.x...x....xx'),
list('x.x.x...xxx.xx..x'),
list('x.x.xx....x..x'),
list('xxx..xx..xxxxxx..x'),
list('x.....x.S....x'),
list('xxxxxxxxxxxxxxxx'),
];
R = len( PLOCA ); S = len( PLOCA[0] );
MIN_KORAKA = [ [-1 for s in range(S)] for r in range(R) ]

print( 'Hrga nalazi Strumpfetu u', labirint( 1, 13 ),
'koraka! ' );
```

Napravimo red na kojem se nalazi samo Hrgino početno polje (r, s).

Do tog polja Hrgi treba 0 koraka. To je treći element liste koji stavljamo u red.

Označi da do polja (r, s) možemo doći u broj_koraka koraka.

Jesmo li pronašli Štrumpfetu?

Napravi po jedan korak na susjedna polja u kojima još nismo bili.

Obišli smo sva dostupna polja, ali nismo pronašli Štrumpfetu :(

Napravi R*S tablicu punu brojeva -1. To znači da Hrga još nigdje nije bio.

3. Rekurzije

Ako Hrga ne može doći do Štrumpfete, bit će ispisano -1. Ako zamijenimo red stogom, onda će također biti uspješno detektirano može li Hrga doći do Štrumpfete, ali broj koraka neće nužno biti najmanji mogući (36 u primjeru sa slike). Isprobajte!

Zadaci za vježbu

1. Ispišite sve načine penjanja na N stuba ako Mirko smije raditi skokove od po jednu stubu ili od po dvije stube.
2. Učitajte brojeve N i K pa ispišite sve moguće kombinacije Lota u kojem se od kuglica označenih brojevima 1, 2, ..., N izvlači K kuglica. Napišite i funkciju koja samo prebrojava koliko takvih kombinacija ima. Na primjer, broj kombinacija standarnog Lota u kojem se od 39 kuglica izvlači njih 7 iznosi 15380937.
3. Mirko ima novčanice u nominacijama od L_1, L_2, \dots, L_N kuna. Napišite funkciju koja određuje može li koristeći se maksimalno jednom novčanicom svake nominacije platiti iznos od X kuna. Zatim modificirajte program tako da ispisuje koje novčanice treba odabrati da bi se dobio iznos od X kuna. Na primjer, ako je $L_1=4, L_2=10$ i $L_3=20$, onda Mirko može platiti iznos od 24 kune, ali ne može platiti iznos od 8 kuna. Na kraju, ubrzajte rekurziju koristeći se memoizacijom.
4. Riješite prethodni zadatak u slučaju da Mirko smije upotrijebiti po volji mnogo novčanica svake nominacije. Na primjer, uz $L_1=4, L_2=10$ i $L_3=20$ mogu se platiti iznosi od 32 i 8 kuna, ali ne može iznos od 35 kn.
5. Ispišite put kojim miš treba ići da bi pojeo maksimalnu količinu sira. U primjeru iz zadatka treba ispisati: "dolje desno desno dolje dolje desno dolje desno". Uputa: osim polja MEMO, treba imati još i polje POMAK, tako da u POMAK[r][s] piše "dolje" ako je miš u polje (r, s) došao tako da se pomaknuo prema dolje iz (r-1, s), a "desno" ako je došao tako da se iz polja (r, s-1) pomaknuo prema desno.



Može li se jedno te isto polje naći više puta na redu? Kako to izbjegići? Unutar svake od četiriju if-naredbi u funkciji labirint možemo preseliti naredbu koja postavlja MIN_KORAKA za susjedno polje i malo promjeniti uvjet u if-u.

4. Datoteke u programskom jeziku Python

U datoteke se pohranjuju svi sadržaji koji se u računalu trebaju trajno čuvati. Datoteku može stvoriti i popuniti jedan program i nakon toga njezin sadržaj može dohvatiti neki drugi program.

Datoteke u računalnim sustavima imaju dvojaku ulogu:

- one služe za trajno pohranjivanje svih oblika informacija.
- pomoću njih se može obavljati razmjena informacija između različitih programa.

Koristit ćemo se znakovnim datotekama: to su datoteke sa sufiksom **.txt**. Takvu datoteku možemo pripremiti i nekim programom za obradu teksta, npr. Notepadom ili napisati u editoru samog Pythona.

4.1. Osnovne metode za rad s datotekama

Metode za otvaranje i zatvaranje datoteka dvije su osnovne metode za rad s datotekama.

1. otvaranje datoteke

`open()`

`open(ime_datoteke, način_uporabe)`

gdje je:

`ime_datoteke` – ime datoteke, zajedno s putanjom

`način_uporabe`:

`r` – čitanje iz datoteke; datoteka se otvara samo za čitanje, a ako takva datoteka ne postoji, javit će se pogreška

`w` – pisanje u datoteku; ako postoji datoteka s navedenim imenom, preko nje će se upisati novi sadržaj, a ako ne postoji, stvorit će se nova datoteka s imenom kojeg smo naveli u `open`

`a` – dodavanje u datoteku; na već postojeću datoteku dodaje se novi sadržaj

2. zatvaranje datoteke

`close()`

Metodu za zatvaranje datoteke obavezno je navesti.

Ostale metode za rad s datotekama

Ime metode	Opis djelovanja
<code>read()</code>	Pročita cijeli sadržaj i vraća kao jedan string. Znak za novi redak pročita kao <code>\n</code> .
<code>readline()</code>	Pročita redak i vraća kao string. Kraj retka označen je s <code>\n</code> . Kada dođe do kraja datoteke, vraća se prazan string <code>"</code> .
<code>readlines()</code>	Čita se po redcima i vraća se lista stringova pojedinih redaka.

4. Datoteke u programskom jeziku Python

Ime metode	Opis djelovanja
seek()	Pokazivač se postavi na mjesto <i>n</i> . <i>n</i> je redni broj znaka u datoteci. dat.seek(0) – pokazivač se vrati na početak datoteke.
tell()	Vraća trenutni položaj pokazivača.
write()	Upisuje string datoteke. Ako je otvorena za pisanje, briše se prethodni sadržaj, ako je otvorena za dodavanje, string se dodaje na kraj datoteke.
writelines()	U datoteku upisuje listu stringova.

4.2. Primjeri u interaktivnom okruženju za open(), read() i readline()

U tekstualnu datoteku *Jurica.txt* napisan je sljedeći sadržaj:

```
Jurica.txt
Programiranje u Pythonu je jako zabavno.
Rad s datotekama je koristan.
```

```
>>> dat=open('Jurica.txt','r')
>>> dat.read()
```

read()

'Programiranje u Pythonu je jako zabavno.\nRad s datotekama je koristan.\n'

Cijeli sadržaj datoteke pročita se i vraća kao jedan string.

```
>>> dat.readline()
```

readline()

Podaci su pročitani i pokazivač se nalazi na kraju datoteke.

```
>>> dat.seek(0)
0
```

seek()

Pokazivač vraćamo na početak datoteke i možemo je ponovno čitati.

```
>>> dat.readline()
'Programiranje u Pythonu je jako zabavno.\n'
>>> dat.readline()
'Rad s datotekama je koristan.\n'
```

Datoteka se čita redak po redak.

```
>>> dat.seek(0)
0
>>> dat.readlines()
['Programiranje u Pythonu je jako zabavno.\n', 'Rad s
datotekama je koristan.\n']
```

readlines()

Pokazivač datoteke postavi se na početak, čita se datoteka i vraća lista stringova.

```
>>> dat.tell()
72
```

tell()

4. Datoteke u programskom jeziku Python

Pokazivač se nalazi na kraju datoteke.

```
>>> dat.seek(10)  
10
```

Pokazivač je pomaknut na mjesto 10. znaka u datoteci.

```
>>> dat.readlines()  
['nje u Pythonu je jako zabavno.\n', 'Rad s datotekama je  
koristan.\n']
```

Podaci su pročitani od 10. znaka do kraja datoteke.

4.3. Primjeri u interaktivnom okruženju za `open()`, `write()` i `writelines()`

`write()`

```
>>> dat=open('Jurica.txt','w')  
>>> dat.write('Korist od uporabe datoteka je velika.')  
40
```

Odgovara broju znakova koji su upisani u rečenici.

`close()`

```
>>> dat.close()
```

Datoteku je potrebno zatvoriti.

Otvorit ćemo datoteku *Jurica.txt*.

<i>Jurica.txt</i>
Korist od uporabe datoteka je velika.

Vidimo sadržaj koji smo u nju upisali.

Datoteku možemo otvoriti npr. u Notepadu (bloku za pisanje).

Pročitamo što smo u nju zapisali. Obrisani je postojeći sadržaj i upisan novi.

```
>>> dat=open('Jurica.txt','a')
```

Otvaramo datoteku u modu za dodavanje sadržaja 'a'.

```
>>> dat.write('\n')  
1
```

Prvo smo upisali novi redak.

```
>>> dat.write('Datoteke se koriste u svim programskim jezici-  
ma.')  
47
```

Dodali smo u datoteku novu rečenicu.

4. Datoteke u programskom jeziku Python

Broj 47. označava mjesto gdje se nalazi pokazivač.

```
>>> dat.close()
```

Sada datoteka ima sljedeći sadržaj.

Jurica.txt

```
Korist od uporabe datoteka je velika.  
Datoteke se koriste u svim programskim jezicima.
```

```
>>> dat=open('Jurica.txt','w')  
>>> dat.writelines('Dobar dan\n svaki dan')  
>>> dat.close()
```

writelines()

Otvorena je datoteka za pisanje.

Jurica.txt

```
Dobar dan  
 svaki dan
```

Primjer 1.

Napiši program koji će u tekstualnu datoteku *Ana.txt* upisati brojeve. Napiši program koji će u izlaznu datoteku *An2.txt* ispisati samo parne brojeve.

Ulazna datoteka <i>Ana.txt</i>	Izlazna datoteka <i>An2.txt</i>
1 3 14 18 20 19	14 18 20

```
ulaz=open("Ana.txt","r")  
izlaz=open("An2.txt","w")  
redci=ulaz.readlines()  
for s in redci:  
    b=s.split()  
    for i in range(len(b)):  
        b[i]=int(b[i])  
        if (b[i] %2 == 0):  
            izlaz.write('{} '.format(b[i]))  
ulaz.close()  
izlaz.close()
```

Najprije otvaramo datoteku u načinu za čitanje.

- 1) for s in redci: , redak pretvaramo u listu stringova
- 2) b=s.split(), svaki element liste pretvaramo u cijeli broj.

Primjer 2.

Napiši program koji će u datoteku *Juram.txt* upisati brojeve. U izlaznu datoteku *Jur1.txt* treba u prvi redak ispisati najveći, u drugi redak najmanji među učitanim brojevima, a u treći redak zbroj svih učitanih brojeva.

4. Datoteke u programskom jeziku Python

Ulazna datoteka Juram.txt	Izlazna datoteka Jur1.txt
12 -5 8 10 4	12 -5 29

```
ulaz=open("Juram.txt", "r")
izlaz=open("Jur1.txt", "w")
redci=ulaz.readlines()
for s in redci:
    b=s.split()
    for i in range(len(b)):
        b[i]=int(b[i])
    izlaz.write('{}\n {}{}\n'.format(max(b),min(b),sum(b)))
ulaz.close()
izlaz.close()
```

Primjer 3.

U svakom retku tekstualne datoteke *Daniel.txt* nalazi se određeni broj cijelih brojeva. Napiši program koji će u izlaznoj datoteci *Marko.txt* napisati zbroj svakog retka posebno.

Ulazna datoteka Daniel.txt	Izlazna datoteka Marko.txt
-14 15 -17 1	-15
-4 3 -10 22 1	12
-22 11 -12 1 0	-22
-2 3 4 12	17

```
ulaz=open("Daniel.txt", "r")
izlaz=open("Marko.txt", "w")
redci=ulaz.readlines()
for s in redci:
    b=s.split()
    for i in range(len(b)):
        b[i]=int(b[i])
    izlaz.write('{}\n'.format(sum(b)))
ulaz.close()
izlaz.close()
```

Primjer 4.

U prvom retku tekstualne datoteke *Jelena.txt* nalaze se cijeli brojevi odvojeni jednim razmakom. Napiši program koji će u prvi red izlazne datoteke *Jela1.txt* ispisati prosjek brojeva, u drugi umnožak brojeva, a u treći najmanji broj.

Ulazna datoteka Jelena.txt	Izlazna datoteka
2 4 -8 4 6 -11	-0.50 16896 -11

4. Datoteke u programskom jeziku Python

```
ulaz=open("Jelena.txt","r")
izlaz=open("Jelal.txt","w")
redci=ulaz.readlines()
s1=0
p=1
for s in redci:
    b=s.split()
    for i in range(len(b)):
        b[i]=int(b[i])
        p=p*b[i]
    pr=sum(b)/len(b)
    izlaz.write('{:.2f}\n {} \n {} \n'.format((pr),(p),min(b)))
ulaz.close()
izlaz.close()
```

Primjer 5.

Napravili smo listu s imenima, prezimenima i mjestima stanovanja. Nakon toga u datoteku smo *for* petljom upisali sadržaj.

Izgled programa:

```
listal=[[ "Jura", "Miletić", "Požega"], [ "Marko", "Miletić", "Zagreb"]
],[ "Mihael", "Lucijanić", "Zagreb"]]
print("Početna lista:")
for redak in listal:
    print(redak)
#Otvaranje datoteke za pisanje
datoteka=open("datoteka.txt","w")
#Zapisivanje u datoteku redak po redak
for element in listal:
    redak=";".join(element)
    redak=redak + "\n"
    datoteka.write(redak)
#Zatvaranje datoteke
datoteka.close()
```

Početna lista:
['Jura', 'Miletić', 'Požega']
['Marko', 'Miletić', 'Zagreb']
['Mihael', 'Lucijanić', 'Zagreb']

Jura;Miletić;Požega
Marko;Miletić;Zagreb
Mihael;Lucijanić;Zagreb

U *for* petlji glavnu listu rastavili smo na podliste. Sve elemente podliste spojili smo znakom „;“, no mogli smo upotrijebiti i bilo koji drugi znak. To radimo metodom *.join()*. Na kraju reda dodajemo „\n“ za prelazak u novi red. Poslije toga zapisujemo sadržaj u datoteku, nakon čega datoteku zatvaramo.

Ono što smo zapisali u datoteku, trebamo i pročitati. To se radi na sljedeći način:

```
lista2=[]
#Otvaranje datoteke za čitanje
datoteka=open("datoteka.txt","r")
#Čitanje iz datoteke redak po redak
for redak in datoteka:
```

(nastavak na idućoj stranici)

4. Datoteke u programskom jeziku Python

```
Pročitana lista:  
['Jura', 'Miletić', 'Požega']  
[Marko', 'Miletić', 'Zagreb']  
['Mihael', 'Lucijanić', 'Zagreb']
```

```
element=redak.rstrip()  
element=element.split(';')  
lista2.append(element)  
print()  
print("Pročitana lista:")  
for redak in lista2:  
    print(redak)
```

Najprije otvaramo datoteku u načinu za čitanje. Nakon toga *for* petljom radimo sve redak po redak:

- 1) *rstrip()* metodom uklanjamo "\n" na kraju reda
- 2) *split()* metodom dijelimo elemente unutar liste kako bismo ih kasnije mogli dohvatiti
- 3) na kraju tu listu dodajemo u glavnu listu i zatvaramo datoteku.

Primjer 6.

U datoteku *brojevi.txt* spremi sljedeće brojeve:

```
6 18 4 19  
15  
17 12 158 9  
16 17
```

Napiši program koji će u izlaznu datoteku ispisati najmanji i najveći broj u svakom retku.

```
ulaz=open("brojevi.txt","r")  
izlaz=open("rezultati.txt","w")  
redci=ulaz.readlines()  
for s in redci:  
    b=s.split()  
    for i in range(len(b)):  
        b[i]=int(b[i])  
    izlaz.write('{} {}'.format(min(b),max(b)))  
ulaz.close()  
izlaz.close()
```

Izlazna datoteka :

```
4 19  
15 15  
9 158  
16 17
```

Primjer 7.

Zadana je sljedeća ulazna datoteka:

```
1 3 9 2 7 8  
3 4 5 10 8  
11 12 15
```

4. Datoteke u programskom jeziku Python

Napiši program uz pomoć datoteka koji će učitati zadane podatke i ispisati najveći u svakom retku.

```
ulaz=open("broj.txt","r")
izlaz=open("izlaz.txt","w")
redci=ulaz.readlines()
for s in redci:
    b=s.split()
    for i in range(len(b)):
        b[i]=int(b[i])
    najveci=max(b)
    izlaz.write('{}'.format(najveci))
ulaz.close()
izlaz.close()
print('Provjeri izlaznu datoteku, izlaz')
```

Izgled izlazne datoteke:

```
9
10
15
```

Primjer 8.

U tekstualnu datoteku *Klupa.txt* spremi podatke za n brojeva. Te brojeve pročitaj iz datoteke i izračunaj i ispiši:

- a) najveći od zadanih brojeva
- b) koliko je brojeva većih od prosjeka
- c) brojeve sortirano od većeg prema manjem
- d) koliko ima brojeva djeljivih s 3 u intervalu $< -2, 7 >$
- e) umnožak neparnih brojeva.

```
dat1=open('Klupa.txt','w')
a=[]
n=int(input("Koliko brojeva:"))
for i in range(0,n):
    x=input("Unesite broj:")
    a.append(x)
dat1.writelines(e + '\n' for e in a)
dat1.close()
dat1=open('Klupa.txt','r')
a=dat1.readlines()
print(a)
dat1.close()
print("\n Zadani brojevi:")
for i in range(0,n):
    a[i]=int(a[i])
    print(a[i])
```

(nastavak na idućoj stranici)

Koliko brojeva:

```
Unesite broj:3
Unesite broj:3
Unesite broj:6
Unesite broj:-2
Unesite broj:7
```

[3\n, 3\n, -6\n, -2\n, 7\n]

Zadani brojevi:

```
3
3
6
-2
7
```

4. Datoteke u programskom jeziku Python

```
s=0
b=0
bl=0
p=1
for i in range(0,n):
    if(i==0):
        max=a[i]
    if(a[i]>max):
        max=a[i]
    s+=a[i]
    if((a[i]%3==0)and (a[i]>-2) and (a[i]<7)):
        bl+=1
    if(a[i]%2 !=0):
        p=p*a[i]
pr=s/n
for i in range(0,n):
    if a[i]>pr:
        b=b+1
print("\n Najveci broj: ",max)
print("\n Vecih od prosjeka ima: ",b)
print("\n Ispis zadanih brojeva sortirano od veceg prema manjem")
for i in range(0,n-1):
    for j in range(i+1,n):
        if a[i]<a[j]:
            t=a[i]
            a[i]=a[j]
            a[j]=t
for i in range(0,n):
    print(a[i])
print("\n Brojeva djeljivih s 3 u intervalu <-2,7> ima:",bl)
print("\n Umnozak neparnih brojeva iznosi:",p)
```

Najveci broj: 7
Vecih od prosjeka ima: 2
Ispis zadanih brojeva sortirano od veceg prema manjem
7
6
3
3
-2

Primjer 9.

U tekstualnu datoteku *Guma.txt* spremi podatke za n učenika: njihov uspjeh, ocjenu iz informatike i masu. Te podatke pročitaj iz datoteke i ispiši:

- a) najbolji uspjeh
- b) mase sortirano od manje prema većoj
- c) umnožak ocjena iz informatike
- d) koliko učenika ima uspjeh 4, ocjenu iz informatike 5 i masu jednaku 70 kg
- e) umnožak parnih masa u intervalu <60,70].

Dobivene rezultate smjesti u tekstualnu datoteku *Gum3.txt*.

```
dat1=open('Guma.txt','w')
u=[ ]
inf=[ ]
```

4. Datoteke u programskom jeziku Python

```
m=[]
n=int(input("Koliko ucenika:"))
for i in range(0,n):
    x=input("Uspjeh:")
    u.append(x)
    y=input("Informatika:")
    inf.append(y)
    z=input("Masa:")
    m.append(z)
dat1.writelines(e + '\n' for e in u)
dat1.writelines(e + '\n' for e in inf)
dat1.writelines(e + '\n' for e in m)
dat1.close()
dat1=open('Guma.txt','r')
a=dat1.readlines()
print(a)
dat1.close()
p=1
b1=0
p1=1
print("\n Zadani uspjeh:")
for i in range(0,n):
    u[i]=int(u[i])
    print(u[i])
print("\n Zadana informatika:")
for i in range(0,n):
    inf[i]=int(inf[i])
    print(inf[i])
print("\n Zadana masa:")
for i in range(0,n):
    m[i]=int(m[i])
    print(m[i])
for i in range(0,n):
    if(i==0):
        max=u[i]
    if(u[i]>max):
        max=u[i]
    p*=inf[i]
    if(u[i]==4)and( inf[i]==5) and(m[i]==70):
        b1+=1;
    if((m[i]%2 == 0)and(m[i]>60) and (m[i]<= 70)):
        p1*=m[i]
izlaz=open("Gum3.txt","w")
izlaz.write('Najbolji uspjeh = {} \n'.format ( max))
izlaz.write('Sortirane mase:\n ')
for i in range(0,n-1):
    for j in range(i+1,n):
        if m[i]>m[j]:
```

Koliko učenika: 3
Uspjeh:4
Informatika:5
Masa: 70
Uspjeh:4
Informatika:5
Masa: 70
Uspjeh: 3
Informatika:4
Masa: 72
[4\n, 4\n, 3\n, 5\n, 5\n, 4\n,
70\n, 70\n, 72\n]

Zadani uspjeh:
4
4
3
Zadana informatika:
5
5
4
Zadana masa:
70
70
72

Najbolji uspjeh = 4
Sortirane mase:
70
70
72

4. Datoteke u programskom jeziku Python

Umnozak ocjena iz informatike
= 100

Broj ucenika koji imaju uspjeh
4, ocjenu iz informatike 5 i
masu jednaku 70: 2
Umnozak parnih masa,<60,70]
= 4900

```
t=m[i]
m[i]=m[j]
m[j]=t
for i in range(0,n):
    izlaz.write('{}\n'.format(m[i]))
    izlaz.write('Umnozak ocjena iz informatike = {} \n'.format(p))
    izlaz.write('Broj ucenika koji imaju uspjeh 4, ocjenu iz
informatike 5 i masu jednaku 70: {} \n'.format(b1))
    izlaz.write('Umnozak parnih masa,<60,70] = {} \n'.format(p1))
izlaz.close()
```

Algoritamski zadaci

1. U prvom retku tekstualne datoteke nalazi se nekoliko cijelih brojeva. Napiši program koji će u izlaznu datoteku ispisati umnožak neparnih brojeva iz ulazne datoteke.
2. Unesi n brojeva u datoteku pod imenom *Jura*. Te brojeve pročitaj iz datoteke i izračunaj zbroj svih zadanih brojeva.
3. U prvom retku tekstualne datoteke nalazi se n cijelih brojeva. Napiši program koji će u prvi red izlazne datoteke ispisati zbroj parnih brojeva, u drugi red umnožak parnih brojeva, a u treći red prosjek pozitivnih brojeva.
4. U tekstualnu datoteku *Stol.txt* spremi podatke za n brojeva. Te brojeve pročitaj iz datoteke. Izračunaj i ispiši:
 - a) najmanji broj
 - b) brojeve poredane od većeg prema manjem
 - c) koliko je brojeva većih od prosjeka
 - d) umnožak neparnih brojeva u intervalu $< -4, 6 >$
 - e) koliko je od zadanih brojeva jednako 3.Dobivene rezultate smjesti u datoteku *Stol2.txt*.
5. U datoteku *Marko4.txt* spremi podatke za n brojeva. Te podatke pročitaj iz datoteke i ispiši:
 - a) prosjek najvećeg i najmanjeg broja
 - b) brojeve poredane od većeg prema manjem
 - c) samo brojeve djeljive s 5
 - d) prosjek brojeva većih od 5
 - e) broj neparnih brojeva.
6. Zadana je matrica reda n . Zadane brojeve spremi u datoteku *Proba1.dat*. Izračunaj:
 - a) najveći broj u trećem stupcu
 - b) koliko je elemenata većih od prosjeka elemenata na sporednoj dijagonali
 - c) umnožak elemenata djeljivih s 3 na glavnoj dijagonali
 - d) koliko je elemenata negativno u cijeloj matrici
 - e) zbroj brojeva jednakih -3 u trećem retku.Rezultate spremi u datoteku *Pro2.dat*.

4. Datoteke u programskom jeziku Python

7. Zadana je matrica reda n . Zadane brojeve spremi u datoteku *Pob1.dat*. Izračunaj:
- najmanji broj u drugom retku
 - koliko je elemenata manjih od 5 na glavnoj dijagonali
 - umnožak brojeva djeljivih s 5 na sporednoj dijagonali
 - zbroj pozitivnih brojeva u cijeloj matrici
 - prosjek cijele matrice.
- Rezultate spremi u datoteku *Ro2.dat*.
8. Zadana je matrica reda n . Zadane brojeve spremi u datoteku *Dar1.dat*. Izračunaj:
- najmanji broj u prvom stupcu
 - koliko je elemenata manjih od prosjeka cijele matrice
 - umnožak brojeva jednakih 5 na sporednoj dijagonali
 - zbroj parnih brojeva u trećem stupcu
 - zbroj pozitivnih brojeva u intervalu $[5, 13]$ u cijeloj matrici.
- Rezultate spremi u datoteku *Dro2.dat*.

9. Zadana je matrica reda n . Zadane brojeve spremi u datoteku *Kik1.dat*. Izračunaj:

- najmanji broj u cijeloj matrici
- prosjek elemenata većih od 3 u cijeloj matrici
- zbroj brojeva djeljivih s 3 u trećem stupcu
- broj negativnih brojeva na sporednoj dijagonali
- umnožak brojeva jednakih 5 na glavnoj dijagonali.

Rezultate spremi u datoteku *Kik2.dat*.

10. U datoteku *Go1.txt* spremi podatke za n brojeva. Te podatke pročitaj iz datoteke i ispiši:

- brojeve poredane od većeg prema manjem
- samo one brojeve koji su djeljivi s 3
- koliko je brojeva manjih od prosjeka
- umnožak parnih brojeva
- treći broj po veličini.

Rezultate spremi u izlaznu datoteku *Go2.txt*.

11. U datoteku *Le1.txt* spremi podatke za n brojeva. Te podatke pročitaj iz datoteke i ispiši:

- zbroj najvećeg i najmanjeg broja
- umnožak brojeva većih od 5
- samo parne brojeve
- brojeve poredane od većeg prema manjem
- koliko je brojeva jednakih 6.

Rezultate ispiši na zaslonu računala.

5. Korisnička grafička sučelja

Svi programi koje smo do sada napisali oslanjali su se na tekstualna sučelja. Bilo kakva komunikacija s programom tražila je unošenje naredbi tipkovnicom. Takav način komunikacije s programom smatran je zastarjelim još u ranim osamdesetim godinama prošlog stoljeća. Osim toga, programi s tekstualnim sučeljima zahtijevaju određeno korisničko predznanje. Korisnik se mora upoznati s nizom naredbi kojima se program koristi i upamtitи ih ako želi da rad na takvu programu teče glatko i bez zadrški.

Mogućnost programiranja upotrebom grafičkih korisničkih sučelja (*GUI – Graphical user interface*) otvara nam pak mogućnost pisanja programa za širu publiku. Programi s grafičkim sučeljima intuitivniji su, lakši za uporabu i korisnici se puno brže mogu naučiti raditi na njima.

5.1. Modul *tkinter*

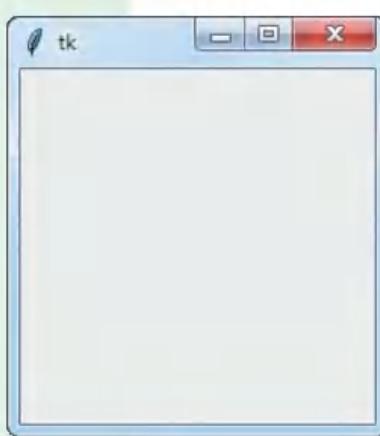
U programskom jeziku Python postoji nekoliko alata za izradu grafičkih sučelja, no jedini alat koji je dostupan u osnovnom instalacijskom paketu jest ***tkinter*** [te-ka-inter].

Tkinter je Pythonovo sučelje za Tk. Tk je alat otvorenog koda (engl. *open source*) koji se upotrebljava u brojnim programskim jezicima za izradu grafičkih sučelja. Sučelje ***tkinter*** implementirano je u modulu *tkinter.py* koji je sastavni dio Pythona.

Kako bismo mogli započeti s izradom grafičkog sučelja, moramo učitati modul ***tkinter***.

```
from tkinter import *
```

Sada smo spremni za rad. Redom ćemo uvoditi naredbe za rad s grafičkim elementima. Neke od njih bit će nužno prihvatići u obliku kakve jesu bez posebnog dodatnog objašnjenja.



5.1.1. Prozor

Osnovni element svakog grafičkog sučelja jest **prozor**. Izrada prozora u ***tkinteru*** iznimno je jednostavna. Promotrimo sljedeći dio koda.

```
from tkinter import *
prozor = Tk()
prozor.mainloop()
```

Nakon pokretanja tog koda dobivamo svoj prvi prozor.

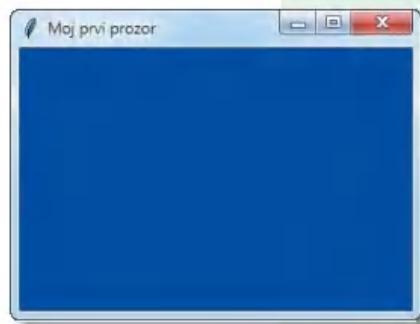
Prozor je, iako prazan, sasvim funkcionalan. Na njemu se nalaze sva obilježja operacijskog sustava u kojem je kreiran uključujući i tri standardna gumba na vrhu prozora.

5. Korisnička grafička sučelja

Objekt prozora kreirali smo naredbom `prozor = Tk()`. Na taj smo način kreirali jedan konkretni prozor koji ćemo zvati `prozor`. Posebno обратимо pozornost na zadnju liniju koda i metodu `mainloop()` koju smo pozvali. Čemu ona služi? `Mainloop` pokreće jednu beskonačnu petlju koja konstantno osluškuje sve događaje (pomicanje pokazivača miša, pritisak tipke na mišu, pritisak tipke na tipkovnici itd.) koji se događaju dok je prozor unutar fokusa.

Naravno, to nije sve što se može napraviti na prozoru. Promotrimo sljedeći, malo složeniji kod.

```
from tkinter import *
prozor = Tk()
prozor.geometry('300x200+100+100')
prozor.resizable(False, True)
prozor.config(background = 'blue')
prozor.title('Moj prvi prozor')
prozor.mainloop()
```



mainloop()

Metodom `geometry()` definiramo veličinu prozora `prozor`. Pritom se `300x200` odnosi na dužinu i širinu prozora, a `+100+100` na poziciju gornjeg lijevog kuta prozora unutar ekrana. Za početnu koordinatu `(0,0)` uzima se gornji lijevi kut ekrana.

Metoda `resizable()` specificira hoće li se prozor moći proširivati ili sužavati po širini i visini. Prvi parametar definira podešavanje po širini, a drugi po visini. Vrijednost `False` onemogućava, a vrijednost `True` dopušta podešavanje.

Metoda `title()` postavlja naslov na prozor.

Metoda `config()` omogućava podešavanje raznih opcija na widgetima. `Background` (hrv. pozadina) jedna je od njih. Svaki widget posjeduje tu metodu.

geometry()

resizable()

title()

config()

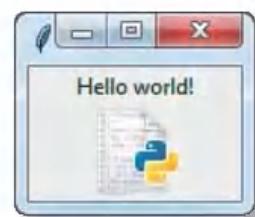
5.2. Grafički elementi (engl. Widget)

Iako smo vidjeli kako prozor možemo učiniti ljepšim, on je i dalje poprilično prazan. **Tkinter** pruža nekolicinu widgeta (visual gadgets) kojima oblikujemo izgled i sadržaj prozora. Tu je riječ najbolje prevesti kao grafički element. Ipak, nadalje ćemo se koristiti izrazom widget. Neke od njih obradit ćemo u nastavku.

5.2.1. Naljepnica (engl. Label)

Naljepnica je widget koji se upotrebljava za prikazivanje kratkog teksta ili slike. Sljedeći primjer pokazuje kako se kreira jednostavna naljepnica s kratkim tekstom i naljepnica sa slikom.

```
from tkinter import *
prozor = Tk()
labelTekst = Label(prozor, text = 'Hello world!')
slika = PhotoImage(file = 'idle.png')
(nastavak na idućoj stranici)
```



PhotoImage()

5. Korisnička grafička sučelja

```
labelTekst.pack()  
labelSlika.pack()
```

```
labelSlika = Label(prozor, image = slika)  
labelTekst.pack()  
labelSlika.pack()  
prozor.mainloop()
```

Linija koda `labelTekst = Label(prozor, text = 'Hello world!')` kreira novu naljepnicu kojoj je roditelj naš prozor i koja u sebi ima tekst "Hello world!", dok linija koda `labelSlika = Label(image = slika)` kreira naljepnicu sa slikom koju smo prethodno učitali u program linijom koda `slika = PhotoImage(file = 'idle.png')`. Jasno je da se ta slika treba nalaziti u istom direktoriju gdje se nalazi i kod programa. Treba napomenuti da **tkinter** prihvata samo slike s ekstenzijama *png* i *gif*. Prilikom kreiranja widgeta na prvom mjestu unutar zagrada treba kao prvi parametar istaknuti kojem prozoru widget pripada jer ako imamo više prozora, napraviti će se prava zbrka s izgubljenim widgetima.

Ključne su linije koda u ovom primjeru: `labelTekst.pack()` i `labelSlika.pack()`. Svaki widget posjeduje metodu `pack()` koja ih postavlja na roditeljski prozor. Metoda `pack()` postavlja widgete jedan ispod drugoga na prozor ako joj ne kažemo drugačije. Postoji još metoda za postavljanje widgeta na prozor, ali o tome ćemo kasnije.

5.2.2. Gumb (engl. Button)

Gumb je jedan od najbitnijih widgeta kod izrade grafičkih sučelja. Gumbima možemo inicirati aktivnosti na prozoru. Dodajmo u prethodni primjer sljedeće naredbe:

```
gumb = Button(prozor, text = 'OK')  
gumb.pack()
```

Dobili smo gumb na svojem prozoru!



`bind()` – povezuje događaj s funkcijom

Gumb sam po sebi ne radi ništa. Kako bismo ga aktivirali, možemo iskoristiti jednu od metoda za rad s gumbima. Na primjer, kada bismo pritiskom na gumb htjeli promjeniti pozadinsku boju prozora te promjeniti tekst unutar naljepnice, učinili bismo to na sljedeći način. Dodajmo u prethodni primjer sljedeće linije koda.

```
def gumbAkcija(event):  
    prozor.config(bg = 'yellow')  
    labelTekst.config(text = 'Dobar dan')  
    return  
...  
gumb.bind('<Button>', gumbAkcija)  
...
```



Prvim parametrom u metodi `bind()` definiramo na koji će događaj koji se dogodi na gumbu metoda reagirati. U ovom slučaju koristimo se parametrom '`<Button>`' koji označava pritisak tipke miša. Drugi je parametar u metodi ime funkcije koja se pokreće u trenutku kada se dogodi zadani događaj. Zahvaljujući tomu, pritiskom na gumb oživljavamo prozor.

Postoji još nekoliko vrsta događaja koji se mogu zadavati kao prvi parametar metodi `bind()`. Neki su od njih:

5. Korisnička grafička sučelja

Naziv parametra	Funkcija se pokreće:
<Button>	na pritisak gumba tipkom miša
<Motion>	na pokret pokazivača miša nad gumbom
<ButtonRelease>	na puštanje pritisnutog gumba
<Double-Button>	na dvostruki klik nad gumbom
<Enter>	na ulaz pokazivača nad gumb
<Leave>	nakon izlaska pokazivača s gumb
<InFocus>	nakon dolaska gumba u fokus

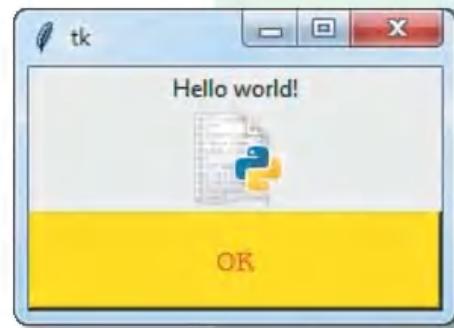
Prilikom kreiranja gumba koristili smo se atributom `text` za definiranje teksta koji će se pojaviti na gumbu. To nije jedini atribut. Upoznajmo još neke od njih.

<code>bg ili background</code>	postavlja željenu boju pozadine gumba
<code>fg ili foreground</code>	postavlja boju teksta na gumbu
<code>font</code>	postavlja željeni font na gumb
<code>height i width</code>	specificira visinu i širinu gumba u pikselima
<code>image</code>	postavlja objekt tipa <code>PhotoImage</code> na gumb
<code>padx i pady</code>	proširuje prazan prostor oko teksta na gumbu za navedenu vrijednost

Ako te attribute ne želimo postaviti prilikom kreiranja gumba, uvijek ih možemo postaviti pozivajući metodu `config`. Na primjer:

```
gumb.config(bg = 'yellow', fg = 'red', font = 'Courier',
width = 20, pady = 10)
```

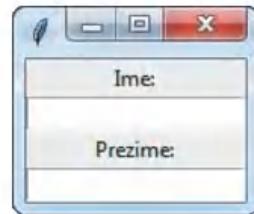
`config()`



5.2.3. Unos (engl. Entry)

Svrha widgeta Unos jest prikazivanje i modificiranje jedne linije teksta. Unos je savršen za kreiranje dijelova grafičkih sučelja u kojima korisnik mora unijeti kratke linije teksta. Napravit ćemo jedan primjer obrasca za unos imena i prezimena.

```
from tkinter import *
prozor = Tk()
labelIme = Label(prozor, text = 'Ime:')
labelPrezime = Label(prozor, text = 'Prezime:')
entryIme = Entry(prozor)
entryPrezime = Entry(prozor)
labelIme.pack()
entryIme.pack()
labelPrezime.pack()
entryPrezime.pack()
prozor.mainloop()
```



U ovaj obrazac korisnik može unijeti svoje ime i prezime. Uoči da smo uzastopnim primjenama metode `pack()` na objektima složili te objekte u prikazanom redoslijedu. Međutim, kako program može pročitati ono što je korisnik napisao i kako sam program

`pack()`

5. Korisnička grafička sučelja

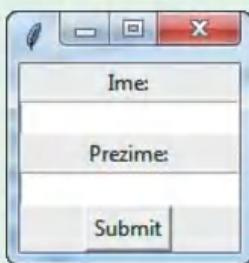
može upisati nešto u Unos? Za te aktivnosti iznimno su nam korisne metode koje widget Unos posjeduje.

- *get()* – vraća trenutni tekst koji se nalazi u Unosu u obliku stringa
- *insert(index, s)* – unosi string s u Unos na prvo mjesto prije indeksa *indeks*.
- *delete(prvi, zadnji)* – briše znakove iz Unosa počevši od prvog indeksa do *zadnji-1* indeksa

Dodajmo u prethodni primjer sljedeće naredbe:

```
def gumbSubmitAkcija(entry):
    print(entryIme.get() + ' ' + entryPrezime.get())
    entryIme.delete(0, END)
    entryPrezime.delete(0, END)
    return

...
gumbSubmit = Button(prozor, text = 'Submit')
gumbSubmit.bind('<Button>', gumbSubmitAkcija)
...
gumbSubmit.pack()
```



Na pritisak gumba Submit (zbog parametra '*<Button>*') metoda *bind()* pokreće funkciju *gumbSubmitAkcija* koja pročita sve što je napisano u oba Unosima i sadržaj ispiše na ekran. Nakon toga funkcija obriše sadržaj iz Unosa. Indeks *0* u metodi *delete()* predstavlja početni indeks stringa u Unosu, dok index *END* predstavlja poziciju iza posljednjeg znaka u stringu.

5.2.4. Tekstualni widget (engl. Text widget)

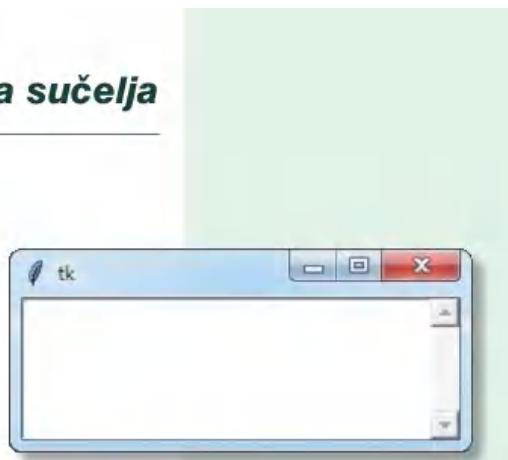
Tekstualni widget upotrebljava se kao prostor za zapisivanje više linija teksta. Radi se o dosta moćnom widgetu jer se od njega mogu napraviti prostori za pisanje raznih veličina, od širine jedne linije teksta do širine cijelog ekrana.

```
from tkinter import *
prozor = Tk()
tekst = Text(prozor, height = 2, width = 30)
tekst.pack()
prozor.mainloop()
```

Naredbom *tekst = Text(prozor, height = 2, width = 30)* kreiramo jedan tekstualni widget koji je širine dva reda teksta i duljine trideset znakova. Sav će se tekst koji je dulji od toga napisati, ali će se prikazivati samo dva posljednja retka. Po napisanom tekstu možemo se kretati strelicama na tipkovnici, ali to nije najlegantniji način. Uz tekstualni widget možemo kreirati novi widget, pomicnu traku (eng. *Scrollbar*), koji nam pomaže u kretanju po tekstu. Pogledajmo primer vezanja pomicne trake i tekstualnog widgeta.

5. Korisnička grafička sučelja

```
from tkinter import *
prozor = Tk()
tekst = Text(prozor, height = 5, width = 30)
traka = Scrollbar(prozor)
traka.config(command = tekst.yview)
tekst.config(yscrollcommand = traka.set)
traka.pack(side = RIGHT, fill = Y)
tekst.pack(side = LEFT, fill = Y)
prozor.mainloop()
```



Povezivanje dvaju widgeta odvija se u *config()* metodama jednog i drugog na navedeni način. Potom se oba widgeta postave na prozor metodama *pack()* kojima smo dodatno napisali atribute *side* i *fill*. Atribut *side* definira na kojoj će strani prozora biti widget, a atribut *fill = Y* određuje da će se widget proširiti po cijeloj visini prozora, ma koliko prozor bio velik.

Kao i kod widgeta Unos, i tekstualni widget ima mnoštvo metoda. Izdvojiti ćemo samo neke:

- *delete(index1, index2)* – metoda briše znakove widgeta od indeksa1 do indeksa2, uključujući i njih
- *get(index1, index2)* – metoda vraća tekst u obliku stringa koji se nalazi između dvaju indeksa
- *insert(index1, t)* – metoda unosi tekst t na widget počevši od mjesta *index1*

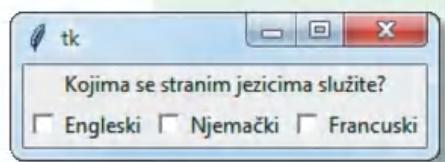
side
fill

5.2.5. Potvrđni gumb i gumb mogućnosti

Potvrđni gumb (engl. *checkbox*), poznat i kao potvrđni okvir (engl. *checkbox*) widget je koji omogućava korisniku višestruki odabir od nekoliko ponuđenih izbora. Jedan potvrđni gumb ima dva stanja: neoznačeno i označeno. Taj se widget jako često upotrebljava i u aplikacijama i na web-stranicama.

Kreirajmo jedan par potvrđnih gumba.

```
from tkinter import *
prozor = Tk()
izborEng = IntVar()
izborGer = IntVar()
izborFre = IntVar()
pitanje = Label(prozor, text = 'Kojima se stranim jezicima služite?')
pitanje.pack()
Checkbutton(prozor, text = 'Engleski', variable = izborEng).
pack(side = LEFT)
Checkbutton(prozor, text = 'Njemački', variable = izborGer).
pack(side = LEFT)
Checkbutton(prozor, text = 'Francuski', variable = izborFre).
pack(side = LEFT)
prozor.mainloop()
```



checkbox

5. Korisnička grafička sučelja

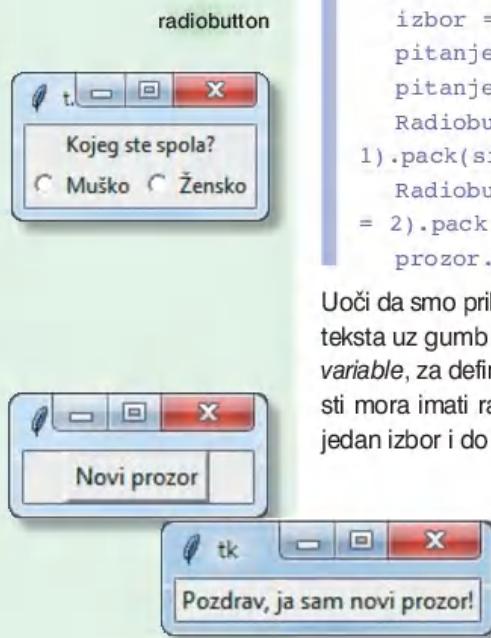
Kao što vidite, potvrđni se gumb ne pridružuje varijabli jer se izbor pojedinog potvrđnog gumba pamti u posebnoj varijabli, objektu tipa *IntVar*. Ako je potvrđni gumb označen, onda se u pripadnom *IntVar* objektu nalazi jedinica, ako nije, nalazi se nula. Vrijednosti koja je zapisana u *IntVar* objektu pristupamo uz pomoć metode *get()*. Na primjer:

```
izbor = izborEng.get()
```

Kod gumba mogućnosti (engl. *radiobutton*) može se odabrati samo jedan izbor od nekoliko ponuđenih. Uporaba gumba mogućnosti jako je slična uporabi potvrđnog gumba, uz jednu razliku. Kako je moguće odabrati samo jedan izbor, tako imamo samo jedan objekt tipa *IntVar* u kojem će se čuvati informacija o tome koji je izbor odabran.

```
from tkinter import *
prozor = Tk()
izbor = IntVar()
pitanje = Label(prozor, text = 'Kojeg ste spola?')
pitanje.pack()
Radiobutton(prozor, text = 'Muško', variable = izbor, value =
1).pack(side = LEFT)
Radiobutton(prozor, text = 'Žensko', variable = izbor, value =
2).pack(side = LEFT)
prozor.mainloop()
```

Uoči da smo prilikom kreiranja gumba mogućnosti upotrebjavali atribute *text*, za definiranje teksta uz gumb mogućnosti, *value*, za definiranje vrijednosti koju gumb mogućnosti nosi, i *variable*, za definiranje varijable u kojoj će ta vrijednost biti upisana. Svaki gumb mogućnosti mora imati različite vrijednosti u atributu *value*. Kao što vidite, može se odabrati samo jedan izbor i do njega možemo doći koristeći se metodom *get()* na objektu *izbor*.



5.2.6. Toplevel

Često pri izradi aplikacija s grafičkim sučeljima treba kreirati novi prozor iz već postojećeg prozora. To se može obaviti naredbom *Toplevel*. U sljedećem ćemo primjeru na pritisak gumba kreirati novi prozor.

```
from tkinter import *
def gumbAkcija(event):
    noviProzor = Toplevel(prozor)
    poruka = Label(noviProzor, text = 'Pozdrav, ja sam novi
prozor!')
    poruka.pack()
    noviProzor.mainloop()
    return
prozor = Tk()
gumb = Button(prozor, text = 'Novi prozor')
gumb.bind('<Button>', gumbAkcija)
gumb.pack()
prozor.mainloop()
```

5. Korisnička grafička sučelja

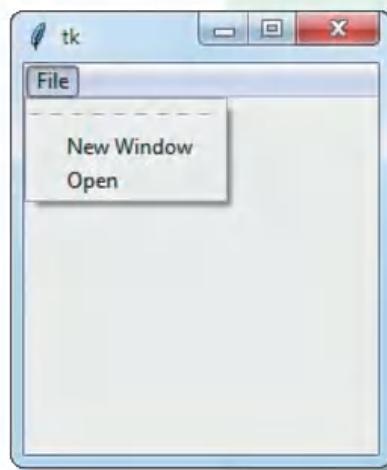
Novi prozor dijete je svojem roditeljskom prozoru i bez njega ne može postojati. Ako se zatvori prozor roditelj, zatvorit će se i prozor dijete. Primijetimo da smo pri kreiranju poruke na novom prozoru naznačili da naljepnica pripada novom prozoru. Ako to ne naznačimo, poruka će se poslije pritiska gumba za novi prozor pojaviti u roditeljskom prozoru.

5.2.7. Izbornici (engl. menu bar)

Izbornik je sastavni dio računalnih aplikacija još od početka izradijanja programa s grafičkim sučeljima. Izbornik je koristan način po spremanja raznih funkcionalnosti na prostorno štedljiv način.

```
from tkinter import *
def noviProzor():
    novi = Toplevel()
    return
def otvori():
    filedialog.askopenfilename()
    return
prozor = Tk()
menubar = Menu(prozor)
podmenu = Menu(menubar)
podmenu.add_command(label = 'New Window', command = lambda :
noviProzor())
podmenu.add_command(label = 'Open', command = lambda :
otvori())
menubar.add_cascade(label = 'File', menu = podmenu)
prozor.config(menu = menubar)
prozor.mainloop()
```

Novi izbornik kreiramo naredbom `menubar = Menu(prozor)` te ga vežemo uz prozor, zatim kreiramo još jedan izbornik naredbom `podizbornik = Menu(menubar)` i vežemo ga za prethodni izbornik. Tako smo stvorili mogućnost padajućeg izbornika. Potom dodajemo nove kategorije u podizbornik, a to su `NewWindow` i `Open`, pa ih opcijom `command` i funkcijom `lambda` povezujemo s pripadajućim funkcijama koje pokreću odgovarajuće akcije. Ovdje se možemo podsjetiti da se neke naredbe trebaju samo upotrebljavati bez dubljeg ulaska u njihovo značenje.



5.3. Upravitelji rasporeda (engl. Layout managers)

Dosada smo se upoznali s jednim načinom postavljanja widgeta na prozor. To je bila metoda `pack()`, koja je najjednostavnija za uporabu. Kod nje ne definiramo precizno gdje će se widget pojaviti na prozoru, već samo okvirno, i to u odnosu na druge widgete. Ipak neke stvari kod metode `pack()` možemo preciznije definirati.

`pack()`

Koristeći se opcijom `fill`, možemo specificirati hoće li widget ispunjavati cijelu dužinu ili širinu prozora. Na primjer:

5. Korisnička grafička sučelja



fill

```
from tkinter import *
prozor = Tk()
poruka1 = Label(prozor, bg = 'blue', text = 'Molim')
poruka2 = Label(prozor, bg = 'blue', text = 'Hvala')
poruka3 = Label(prozor, bg = 'blue', text = 'Izvoli')
poruka4 = Label(prozor, bg = 'blue', text = 'Oprosti')
poruka1.pack()
poruka2.pack(fill = X)
poruka3.pack(fill = X)
poruka4.pack()
prozor.mainloop()
```

Poruke "Hvala" i "Izvoli" proširene su na cijelu veličinu ekrana. Njihova će širina uvijek pratiti širinu prozora.

Sljedeća opcija kojom se možemo koristiti jest *side*. Njome određujemo na kojoj će se strani prozora widget smjestiti. Npr., promijenimo dio prethodnog koda u sljedeći:

side

```
poruka1.pack(side = LEFT, padx = 5, pady = 5)
poruka2.pack(side = LEFT, padx = 5, pady = 5)
poruka3.pack(side = LEFT, padx = 5, pady = 5)
poruka4.pack(side = LEFT, padx = 5, pady = 5)
```

Sve poruke teže smještanju na lijevu stranu, stoga su sve poredane u jedan red. Opcije *padx* i *pady* omogućavaju proširenje praznog prostora oko widgeta, što radi preglednost može biti jako korisno.

Sljedeća metoda postavljanja widgeta na prozor jest *place()*. Ona je sušta suprotnost načinu postavljanja *pack()*. Kod metode *place()* definiramo točnu veličinu i položaj widgeta na prozoru.

place()

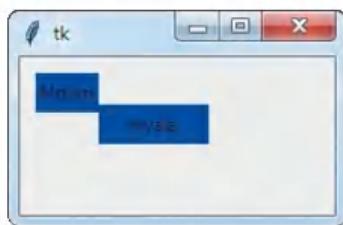


```
from tkinter import *
prozor = Tk()
prozor.geometry('200x100+100+100')
poruka1 = Label(prozor, bg = 'blue', text = 'Molim')
poruka2 = Label(prozor, bg = 'blue', text = 'Hvala')
poruka1.place(x = 10, y = 10, width = 40, height = 25)
poruka2.place(x = 50, y = 30, width = 70, height = 25)
prozor.mainloop()
```

Taj način postavljanja widgeta na prozor programeri izbjegavaju jer je izrazito nefleksibilan.

Posljednja metoda za postavljanje elemenata na prozor jest *grid()*. Grid postavlja dane mu widgete u dvo-dimenzionalnu strukturu, poput matrice.

grid()



```
from tkinter import *
prozor = Tk()
poruka1 = Label(prozor, text = 'Molim')
poruka2 = Label(prozor, text = 'Hvala')
poruka3 = Label(prozor, text = 'Izvoli')
poruka4 = Label(prozor, text = 'Oprosti')
(nastavak na idućoj stranici)
```

5. Korisnička grafička sučelja

```
poruka1.grid(row = 0, column = 0)
poruka2.grid(row = 0, column = 1)
poruka3.grid(row = 1, column = 0)
poruka4.grid(row = 1, column = 1)
prozor.mainloop()
```

Metoda `grid` prima dva atributa, `row` i `column`, u kojima određujemo koordinate pojedinog widgeta.

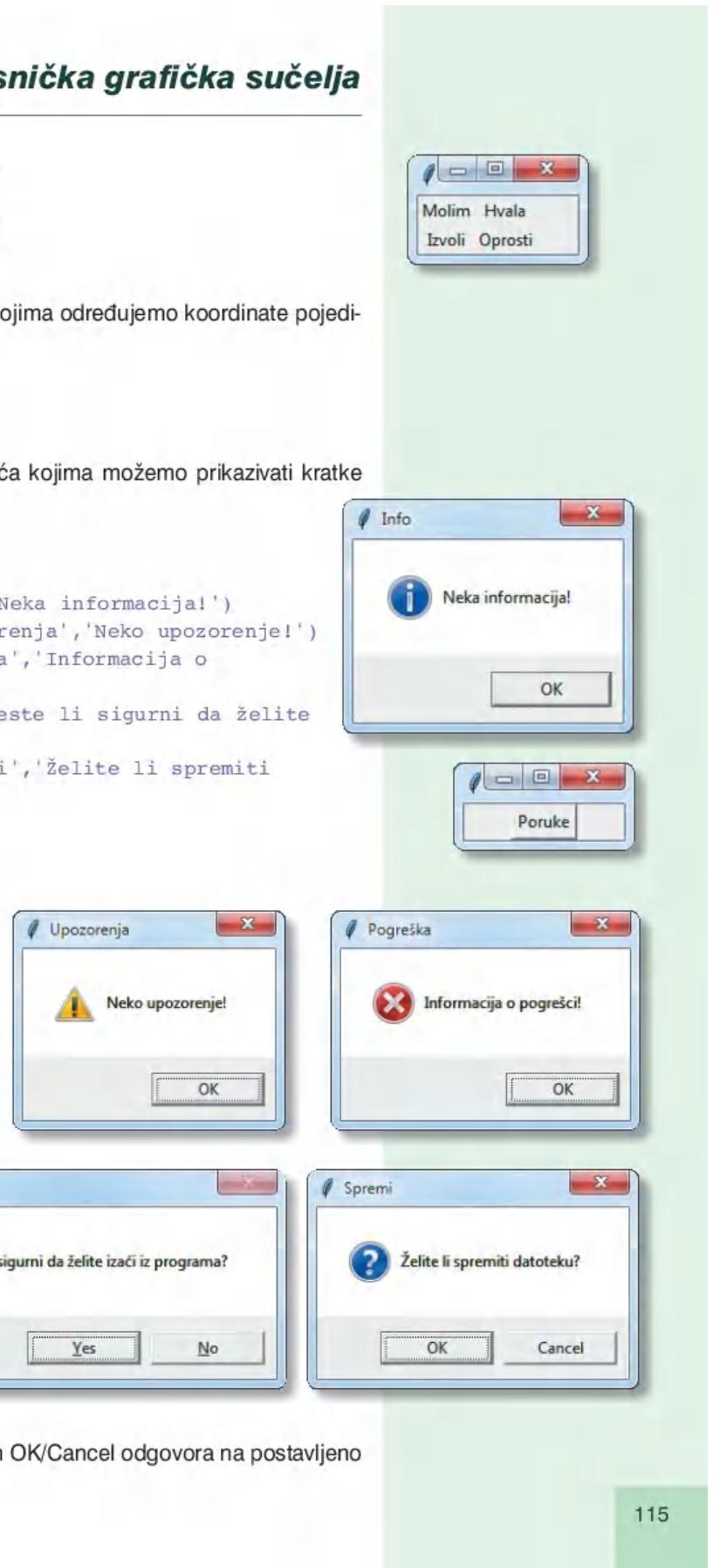
5.4. Poruke

Tkinter posjeduje nekoliko standardnih prozorčića kojima možemo prikazivati kratke poruke korisniku i zahtijevati njegovu potvrdu.

```
from tkinter import *
def gumbAkcija(event):
    messagebox.showinfo('Info', 'Neka informacija!')
    messagebox.showwarning('Upozorenja', 'Neko upozorenje!')
    messagebox.showerror('Pogreška', 'Informacija o pogrešci!')
    messagebox.askyesno('Upit', 'Jeste li sigurni da želite izaći iz programa?')
    messagebox.askokcancel('Spremi', 'Želite li spremiti datoteku?')
return
prozor = Tk()
gumb = Button(text = 'Poruke')
gumb.bind('<Button>', gumbAkcija)
gumb.pack()
prozor.mainloop()
```

Funkcijom `gumbAkcija` realizirali smo otvaranje prozorčića s pripadajućim porukama. Pri tome smo se koristili sljedećim naredbama iz paketa `messagebox`:

- sa `showinfo` otvorili smo informacijski prozorčić sa slovom *i* u plavom krugu
- sa `showwarning` otvorili smo prozorčić upozorenja s uskličnikom u žutom trokutu
- sa `showerror` otvorili smo prozorčić o pogrešci sa slovom *x* u crvenom krugu
- sa `askyesno` otvorili smo prozorčić s izborom Da/Ne odgovora na postavljeno pitanje
- sa `askokcancel` otvorili smo prozorčić s izborom OK/Cancel odgovora na postavljeno pitanje.

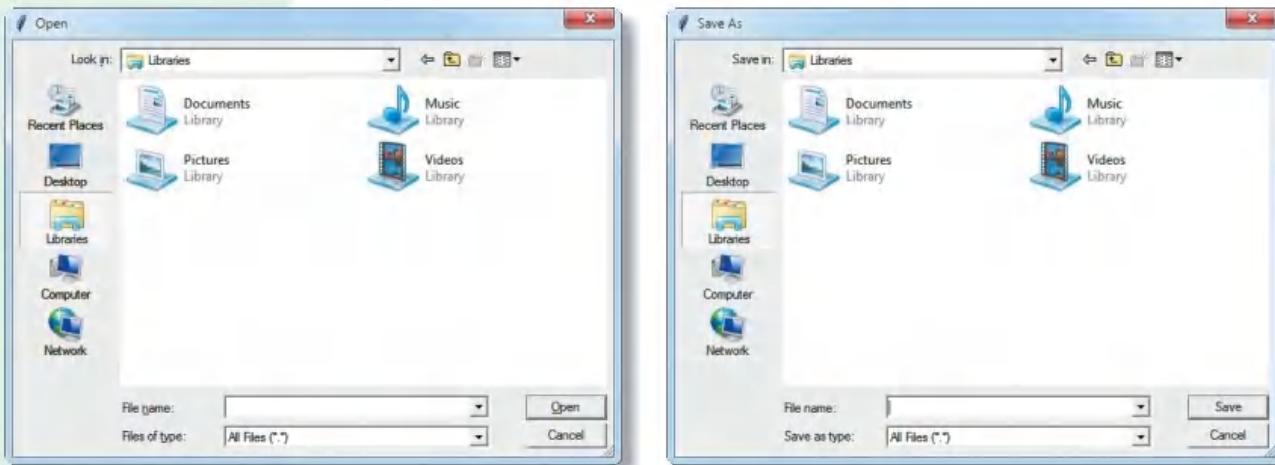


5. Korisnička grafička sučelja

Tkinter omogućava i poziv dijaloga za otvaranje datoteka ili spremanje datoteka koji su ugrađeni u operacijski sustav u kojem izvršavamo program.

```
odabranaDatoteka = filedialog.askopenfilename()
odabranaPutanja = filedialog.asksaveasfilename()
```

Ti su dijalazi posebno korisni jer nam pružaju uporabu kompletne funkcionalnosti operacijskog sustava pri otvaranju i spremanju datoteka.



Od problema do rješenja

Nakon što smo uveli neke od najznačajnijih elemenata za rad s grafičkim sučeljima, vjerojatno smo u stanju osmislići i izraditi jedan projekt. Tijekom izrade projekta trebat ćemo upotrijebiti većinu uvedenih elemenata.



5.5.1. Ideja projekta

Bingo je igra na sreću. Opišimo jednu verziju te igre koju ćemo u nastavku implementirati. Igrač na početku igre dobije listić s 15 brojeva. Svi su brojevi veći od jedan i manji od 90 (uključivo) te se mogu ponavljati. Voditelj igre iz bubenja, u kojem se nalaze svi brojevi od jedan do 90, redom izvlači po jedan od njih. Svaki put kada voditelj izvuče jedan broj, igrač provjerava ima li tog broja na njegovu listiću. Ako ima, nekako ga treba označiti. Cilj je igre što prije označiti sve brojeve na listiću. Ovisno o tome koji je po redu zadnji označeni broj na listiću, igrač dobiva pripadajuću nagradu. Pri tome vrijedi da ako igrač uspije označiti sve brojeve prije:

- 30. izvučenog broja iz bubenja, onda dobiva "SuperBingo"
- 33. izvučenog broja iz bubenja, onda dobiva "Bingo33"

5. Korisnička grafička sučelja

- 36. izvučenog broja iz bubnja, onda dobiva "Bingo36"
- 39. izvučenog broja iz bubnja, onda dobiva "Bingo39".

Ako mu to uspije nakon 40. izvučenog broja, onda dobiva "Bingo40+". U konačnici, naš bi program trebao izgledati kao što je prikazano na slikama.

Nakon što smo definirali pravila igre, krenimo u njezino implementiranje.

5.5.2. Realizacija projekta

Projekt ćemo započeti izradom početnog prozora s jednom naljepnicom koja nosi sliku igre na sreću. Početni prozor na sebi će imati jedan izbornik i prostor za unos imena, spola i potvrdu da je korisnik aplikacije punoljetan. Potvrda starosti ključni je dio jer bez toga korisnik neće moći nastaviti igru. Unos imena i spola je optionalan i nije obvezan za nastavak rada programa.

Krenimo redom. U ovom dijelu koda kreirat ćemo prozor, postaviti mu naslov i onemogućiti promjenu veličine prozora. Potom ćemo pripremiti i postaviti sliku *Bingo15_90.png* na *labelSlika* i istu tu naljepnicu metodom *pack()* postaviti na prozor.

```
prozor = Tk()
prozor.title('Dobrodošli u igru Bingo!')
prozor.resizable(False, False)
slika = PhotoImage(file = 'Bingo15_90.png')
labelSlika = Label(image = slika)
labelSlika.pack()
```

Zatim ćemo kreirati izbornik s dvjema kategorijama, Novi listić i Pravila igre. Njih ćemo povezati s funkcijama *noviListić(prozor)* i *pravilaIgre(prozor)*. Tim dvjema funkcijama posvetit ćemo se malo kasnije.

```
menubar = Menu(prozor)
igramenu = Menu(menubar)
igramenu.add_command(label = 'Novi listić', command = lambda
: noviListic(prozor))
igramenu.add_command(label = 'Pravila igre', command = lambda
: pravilaIgre(prozor))
menubar.add_cascade(label = 'Igra', menu = igramenu)
prozor.config(menu = menubar)
```

Menu

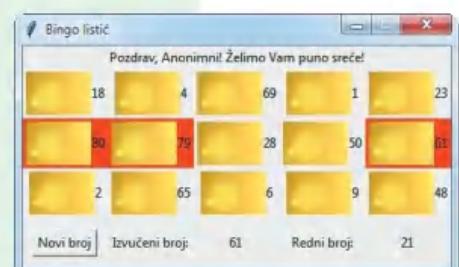
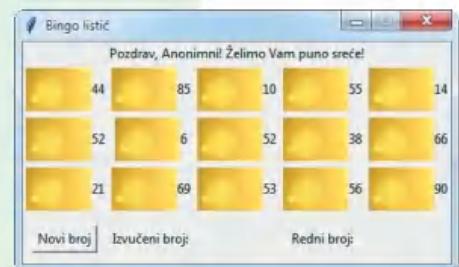
Kreiranjem Unosa za unos imena, gumba mogućnosti za odabir spola i potvrdnog gumba završit ćemo izgled glavnog prozora.

```
labelIme = Label(prozor, text = 'Ime:')
labelIme.pack(side = 'left', padx = 5, pady = 10)
entryIme = Entry(prozor)
entryIme.pack(side = 'left', padx = 5, pady = 10)
spol = IntVar()
```

Naljepnica

(nastavak na idućoj stranici)

Unos



5. Korisnička grafička sučelja

Potvrđni gumb
Gumb mogućnosti

```
Radiobutton(prozor, text = 'M', variable = spol, value =
1).pack(side = 'left')
Radiobutton(prozor, text = 'Ž', variable = spol, value =
2).pack(side = 'left')
punoljetan = IntVar()
Checkbutton(prozor, text = 'Ja sam punoljetna osoba',
variable = punoljetan).pack(side = 'left')
prozor.mainloop()
```

Daljnja funkcionalnost programa nastavlja se odabirom kategorije iz izbornika. Ako korisnik odabere kategoriju Pravila igre, pokrenut će se funkcija *pravilaIgre(prozor)* koja kao argument prima ime prozora iz kojeg pozivamo funkciju.

Funkcija *pravilaIgre()* otvara novi prozor u kojem se nalazi tekstualni widget s pomicnom trakom i tekstrom pravila igre.

Scrollbar

```
def pravilaIgre(roditelj):
    pravila = Toplevel(roditelj)
    pravila.title('Pravila igre')
    pravila.resizable(False, False)
    tekst = 'Na licu svakog listića nalazi se petnaest (15) različitih brojeva, iz niza od 1 do 90.\nBINGO \nPetnaest pogodenih brojeva u sva tri reda unutar jedne kombinacije:\ndo zaključno 30. izvučenog broja – SUPERBINGO\nod 31. do 33. izvučenog broja – BINGO 33\nod 34. do 36. izvučenog broja – BINGO 36\nod 37. do 39. izvučenog broja – BINGO 39\nod 40. izvučenog broja do ostvarenja dobitka – BINGO 40+'
    S = Scrollbar(pravila)
    T = Text (pravila, height = 5, width = 60)
    S.pack(side = 'right', fill = Y)
    T.pack(side = 'left', fill = Y)
    S.config(command = T.yview)
    T.config(yscrollcommand = S.set)
    T.insert(END, tekst)
    return
```

Ako korisnik odabere kategoriju Novi listić iz izbornika, pokrenut će se funkcija *noviListic(prozor)*. Na početku te funkcije trebamo provjeriti je li osoba punoljetna provjeravajući je li potvrđni gumb pritisnut. Ako nije pritisnut, javlja se upozorenje i funkcija završava s radom. Inače, ako je osoba punoljetna, kreiramo novi prozor s Bingo listićem.

```
def noviListic(root):
    listaIzvucenih = []
    if(punoljetan.get() != 1):
        messagebox.showwarning('Upozorenje!', 'Maloljetnim osobama zabranjeno je igranje igre na sreću!')
        return
    else:
        listic = Toplevel(root)
        listic.title('Bingo listić')
```

(nastavak na idućoj stranici)

5. Korisnička grafička sučelja

```
listic.resizable(False, False)

if(entryIme.get() == ''):
    ime = 'Anonimni'
else:
    ime = entryIme.get()
labelPozdrav = Label(listic, text = 'Pozdrav, ' +
ime + '! Želimo Vam puno sreće!')
labelPozdrav.grid(row = 0, column = 0, columnspan = 5)
```

Dodatno smo kreirali pozdravnu poruku s pomoću jedne naljepnice koju smo nazvali *labelPozdrav* i postavili je u matricu rasporeda u prvi red. Opcijom *columnspan=5* nalaščavamo da će prvi red biti širine pet stupaca.

Sada ćemo napraviti matricu provjere koja je veličine 3x5 i popunjena je nulama. Matrica provjere služit će nam kao zapis na kojem je mjestu pogoden broj u igri na sreću.

```
matricaProvjere = []
redak = []
for i in range(3):
    for j in range(5):
        redak.append(0)
    matricaProvjere.append(redak)
    redak = []
```

Budući da se Bingo listić sastoji od matrice 3x5 s brojevima, kreiramo novu matricu 3x5 koja u sebi sadržava naljepnice. Naljepnice imaju na sebi i tekst i sliku. Osobito je važna opcija *compound='left'* jer ona specificira na kojoj će strani biti smještena slika unutar naljepnice. U ovom slučaju slika je postavljena lijevo od broja.

```
labeli = []
redak = []
slikaBroj = PhotoImage(file = 'Bingo_broj.png')
for i in range(3):
    for j in range(5):
        redak.append(Label(listic, text = randint(1,
90), image = slikaBroj, compound = 'left'))
        labeli.append(redak)
        redak = []

    for i in range(3):
        for j in range(5):
            labeli[i][j].grid(row = i+1, column = j)
```

Potom pravimo gumb na kojem piše novi broj i metodom *bind()* povezujemo ga s funkcijom *noviBrojAkcija* koju ćemo napisati kasnije. Kreiramo naljepnice na kojima će nam pisati trenutno izvučeni broj i koji je redni broj izvučenog broja.

```
noviBroj = Button(listic, text = 'Novi broj')
noviBroj.bind('<Button>', noviBrojAkcija)
noviBroj.grid(row = 4, column = 0, pady = 10)
izvučeniBrojLabel = Label(listic, text = 'Izvučeni broj:')
```

(nastavak na idućoj stranici)

5. Korisnička grafička sučelja

```
    izvuceniBrojLabel.grid(row = 4, column = 1, pady =  
    10)  
    prikazBroja = Label(listic)  
    prikazBroja.grid(row = 4, column = 2, pady = 10)  
    redniBrojLabel = Label(listic, text = 'Redni broj: ')  
    redniBrojLabel.grid(row = 4, column = 3, pady = 10)  
    brojIzvucenih = Label(listic)  
    brojIzvucenih.grid(row = 4, column = 4, pady = 10)  
    listic.mainloop()  
    return
```

Funkcija *noviBrojAkcija* ima za ulogu odabir slučajnog broja i provjeru je li neki broj na listiću identičan slučajnom broju. U isto vrijeme funkcija će provjeravati je li Bingo ostvaren. Funkcija prvo provjerava je li izvučeno svih 90 brojeva. Ako jest, javlja poruku i prekida rad funkcije. Inače bira novi slučajni broj vodeći računa o tome da taj broj još nije izvučen. Pozivom metode *config()* postavlja se vrijednost izvučenog broja i redni broj na pripadajuće naljepnice. Potom funkcija provjerava je li izvučeni broj jednak kojem broju u matrici naljepnica. Ako su jednaki, onda se ta informacija bilježi u matricu provjere na odgovarajuće mjesto.

```
def noviBrojAkcija(event):  
    if(len(listaIzvucenih) >= 90):  
        messagebox.showinfo('Bingo', 'Izvučeni su svi  
brojevi')  
        return  
    else:  
        def noviBrojAkcija(event):  
            if(len(listaIzvucenih) >= 90):  
                messagebox.showinfo('Bingo', 'Izvučeni su svi  
brojevi')  
                return  
            else:  
                zastavica = 0  
                while(zastavica == 0):  
                    pom = randint(1, 90)  
                    if (pom not in listaIzvucenih):  
                        listaIzvucenih.append(pom)  
                        randomBroj = pom  
                        zastavica = 1  
  
                prikazBroja.config(text = randomBroj)  
                brojIzvucenih.config(text = len(listaIzvucenih))  
                for i in range(3):  
                    for j in range(5):  
                        if(labeli[i][j].cget('text') ==  
prikazBroja.cget('text')):
```

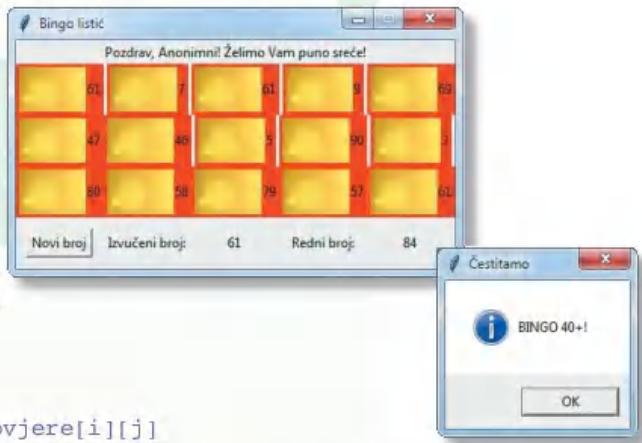
(nastavak na idućoj stranici)

5. Korisnička grafička sučelja

```
        labeli[i][j].config(bg = 'red')
        matricaProvjere[i][j] = 1
    provjeraBingo()
return
```

Na kraju funkcije poziva se funkcija *provjeraBingo()*. Ta funkcija provjerava je li ostvaren dobitak. Zbrajaju se vrijednosti iz *matricaProvjere* i ako je zbroj jednak 15, funkcija kreira BINGO poruke. Ovisno o tome do kojeg je broja Bingo izvučen, ispisuje se odgovarajuća poruka.

```
def provjeraBingo():
    if (len(listaIzvucenih) <= 30):
        suma = 0
        for i in range(3):
            for j in range(5):
                suma += matricaProvjere[i][j]
        if (suma == 15):
            messagebox.showinfo('Čestitamo', 'SUPERBINGO!')
    elif(len(listaIzvucenih) >= 31 and len(listaIzvucenih) <= 33):
        suma = 0
        for i in range(3):
            for j in range(5):
                suma += matricaProvjere[i][j]
        if (suma == 15):
            messagebox.showinfo('Čestitamo', 'BINGO 33!')
    elif(len(listaIzvucenih) >= 34 and len(listaIzvucenih) <= 36):
        suma = 0
        for i in range(3):
            for j in range(5):
                suma += matricaProvjere[i][j]
        if (suma == 15):
            messagebox.showinfo('Čestitamo', 'BINGO 36!')
    elif(len(listaIzvucenih) >= 37 and len(listaIzvucenih) <= 39):
        suma = 0
        for i in range(3):
            for j in range(5):
                suma += matricaProvjere[i][j]
        if (suma == 15):
            messagebox.showinfo('Čestitamo', 'BINGO 39!')
    else:
        suma = 0
        for i in range(3):
            for j in range(5):
                suma += matricaProvjere[i][j]
        if (suma == 15):
            messagebox.showinfo('Čestitamo', 'BINGO 40+!')
```



6. Koordinatna grafika

6.1. Kornjačina grafika

Crtanje i rad s grafičkim funkcijama zanimljiv je i neizbjegjan dio svakog programskog jezika. Tako i programski jezik Python ima nekoliko programskih paketa koji se mogu upotrijebiti za razne grafičke potrebe, a mi ćemo se upoznati s modulom **turtle** koji dolazi zajedno s distribucijom programskog paketa Python. Kornjačina grafika originalno je dio programskog jezika Logo, a kreirali su je Wally Feurzeig i Seymour Papert 1966. godine. Pomicanjem, kretanjem kornjače pera realiziraju se grafički elementi. Pretpostavljamo da ste se u dosadašnjem školovanju susreli s programskim jezikom Logo pa znate o čemu je ovdje riječ.

Modul *turtle* podržava rad i na proceduralni i objektno-orientirani način. Objektno-orientirano sučelje koristi se dvjema klasama: klasom *TurtleScreen* i klasom *RawTurtle* (*RawPen*). U proceduralnom načinu mogu se upotrebljavati sve funkcije koje su izvedene iz metoda klase *screen* i *turtle* i imaju isti naziv kao odgovarajuća metoda.

U sljedećoj tablici prikazane su metode sadržane u modulu *turtle*:

Metoda	Objašnjenje
<code>forward(d), fd(d)</code>	pomiče pero za <i>d</i> jedinica naprijed
<code>backward(), back(), bk()</code>	pomiče pero za <i>d</i> jedinica unatrag
<code>right(kut), rt(kut)</code>	zakreće pero za <i>kut</i> stupnjeva udesno
<code>left(kut), lt(kut)</code>	zakreće pero za <i>kut</i> stupnjeva ulijevo
<code>goto(x,y), setpos(x,y), setposition(x,y)</code>	pomiče pero na točku s koordinatama (x,y)
<code>setx(x)</code>	postavlja prvu koordinatu na <i>x</i> , druga ostaje nepromijenjena
<code>sety(y)</code>	postavlja drugu koordinatu na <i>y</i> , prva ostaje nepromijenjena
<code>setheading(kut), seth(kut)</code>	usmjerava pero tako da pokazuje u smjeru kuta <i>kut</i>
<code>home()</code>	postavlja pero u početni položaj
<code>circle(r [, extend] [, korak])</code>	crti kružnicu polumjera <i>r</i> , ako želimo samo dio kružnog luka, navest ćemo parametar <i>extend</i> , a parametar <i>korak</i> upotrebljava se za dobivanje pravilnih mnogokuta
<code>dot(d,b)</code>	crti točku promjera <i>d</i> i boje <i>b</i>
<code>undo()</code>	poništava zadnju akciju pera, može se upotrijebiti višekratno
<code>speed(b)</code>	postavlja brzinu crtanja (0 bez animacije, 1 najsporije, 10 najbrže)
<code>position(), pos()</code>	vraća trenutnu poziciju (x,y) pera
<code>towards(x,y)</code>	vraća kut koji zatvara trenutna pozicija pera i pozicija s koordinatama (x,y)
<code>xcor()</code>	vraća x koordinatu trenutne pozicije pera
<code>ycor()</code>	vraća y koordinatu trenutne pozicije pera
<code>heading()</code>	vraća trenutnu orientaciju pera

6. Koordinatna grafika

Metoda	Objašnjenje
<code>distance(x,y)</code>	vraća udaljenost između točke (x,y) i trenutne pozicije pera
<code>distance(A)</code>	vraća udaljenost između točke A i trenutne pozicije pera
<code>pendown(), pd(), down()</code>	pero se spušta i ostavlja trag
<code>penup(), pu(), up()</code>	pero se podiže i ne ostavlja trag
<code>pensize(d), width(d)</code>	pero ima debljinu d jedinica
<code>pen()</code>	vraća ili postavlja attribute pera (debljina, boja,...)
<code>isdown()</code>	vraća vrijednost True ako je pero spušteno, u protivnom vraća vrijednost False
<code>color()</code>	očitava i vraća boju pera i ispune
<code>pencolor()</code>	očitava i vraća boju pera
<code>fillcolor()</code>	očitava i vraća boju ispune
<code>filling()</code>	vraća stanje ispune (True/False)
<code>begin_fill()</code>	početak ispune
<code>end_fill()</code>	kraj ispune
<code>stamp()</code>	kopira – ostavlja otisak pera na trenutnoj poziciji
<code>reset()</code>	brše sve crteže u grafičkom prozoru, postavlja pero u početni položaj i sve attribute postavlja na početne vrijednosti
<code>clear()</code>	brše crteže u grafičkom prozoru, pero ostaje nepromijenjeno
<code>tracer(True/False)</code>	uključuje/isključuje animaciju
<code>write(s)</code>	ispisuje tekst na mjestu gdje se nalazi pero, dodatno se mogu odrediti parametri fonta i poravnanje
<code>showturtle(), st()</code>	postavlja pero da bude vidljivo
<code>hideturtle(), ht()</code>	skriva pero, ne prikazuje ga
<code>isvisible()</code>	vraća vrijednost True ako je pero vidljivo, u protivnom vraća vrijednost False
<code>onclick(funkcija,gumb_misa)</code>	na odgovarajući klik miša izvršava se funkcija
<code>onrelease(funkcija,gumb_misa)</code>	slično metodi onclick(), samo za otpuštanje gumba miša
<code>textinput(title, prompt)</code> <code>numinput(title, prompt, inicijalna, min,max)</code>	omogućava unos podataka u grafičkom načinu rada
<code>bye()</code>	zatvara grafički prozor
<code>exitonclick()</code>	klik na prozor (ekran) pokreće metodu bye()
<code>title(naslov)</code>	<i>naslov</i> grafičkog prozora
<code>mainloop()</code>	mora biti zadnja naredba radi pravilnog izvršavanja programa

Kreiranje grafičkog prozora i rad u njemu mogući su i u interaktivnom modu. Nakon što napišemo:

```
>>> from turtle import *
naizgled se ništa neće dogoditi, no nakon
>>> title('Grafika')
```

```
from turtle import *
kreiranje grafičkog
prozora
```

6. Koordinatna grafika

otvorit će se prozor veličine 600x600 piksela. Ishodište koordinatnog sustava (točka (0,0)) točno je u sredini prozora. Pero se nalazi na početnoj poziciji, odnosno u ishodištu i usmjereno je prema pozitivnom smjeru osi x.

Sad ćemo se kroz niz primjera pobliže upoznati s radom i uporabom nekih od navedenih metoda.

Primjer 1.

Na svakom vrhu kvadrata otisnite pero u drugoj boji i orijentaciji.

```
from turtle import *
pensize(10)
boja = ["green", "brown", "blue", "red"]
pu()
for b in boja:
    color(b)
    forward(100)
    left(90)
    stamp()
mainloop()
```

Primjer 2.

Nacrtajte jednakostranični trokut kod kojeg je svaka stranica druge boje

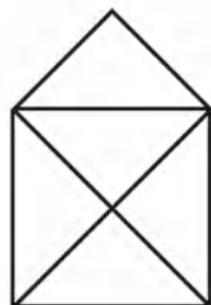
```
from turtle import *
boja = ["purple", "magenta", "maroon"]
pensize(5)
for b in boja:
    color(b)
    fd(100)
    left(120)
mainloop()
```



Primjer 3.

Sjećate li se kako se nacrt „kućica“ u jednom potezu? Vjerujemo da ste to probali barem jednom. Evo jednog rješenja tog problema.

```
from turtle import *
from math import *
pensize(2)
a = 100
fd(a)
lt(135)
fd(a*sqrt(2))
rt(135)
fd(a)
lt(135)
fd(a/2*sqrt(2))
lt(90)
```



(nastavak na idućoj stranici)

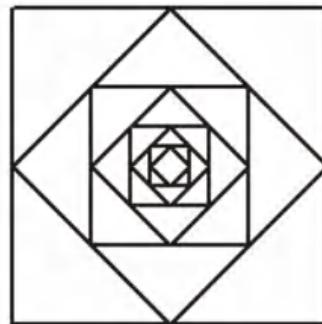
6. Koordinatna grafika

```
fd(a/2*sqrt(2))
lt(45)
fd(a)
lt(135)
fd(a*sqrt(2))
rt(135)
fd(a)
ht()
mainloop()
```

Primjer 4.

U kvadrat stranice a upišite kvadrat kojem su vrhovi polovišta stranica prvog kvadrata. Postupak se nastavlja sve dok je stranica kvadrata veća od 20. Odredite zbroj svih površina tako dobivenih kvadrata.

```
from turtle import *
from math import *
def kvadrat(a):
    pensize(2)
    for i in range(4):
        fd(a)
        rt(90)
a = numinput('Ulaz', 'Duljina stranice kvadrata(20, 400): ', 50, 20, 400)
p = a*a
kvadrat(a)
while a > 20:
    pu()
    fd(a/2)
    rt(45)
    a = a *sqrt(2)/2
    p = p+a**2
    pd()
    kvadrat(a)
ht()
s = 'p = '+str(p)
pu()
goto(0,20)
write(s, font = ('Verdana', 14, 'bold'))
mainloop()
```

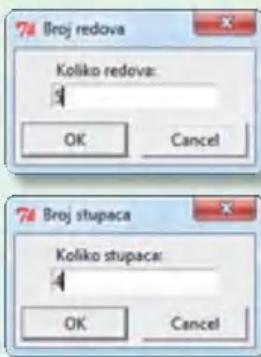


Primjer 5.

Ispišite zelene točke promjera 30 raspoređene u tablici od m redova i n stupaca.

```
from turtle import *
title('Prvi')
dd = 30
pocx = 0
pocy = 0
m = numinput('Broj redova', 'Koliko redova: ')
(nastavak na idućoj stranici)
```

6. Koordinatna grafika

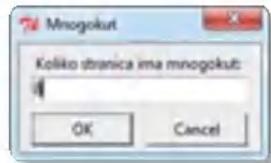


```
n = numinput('Broj stupaca','Koliko stupaca: ')
for x in range(int(n)):
    for y in range(int(m)):
        dot(20,'green')
        pu()
        goto (pox+x*dd,poy+y*dd)
        pd()
dot(20,'green')
pu
mainloop()
```



Primjer 6. Crtanje pravilnog n-terokuta

```
from turtle import *
n = int(numinput('Mnogokut','Koliko stranica ima mnogokut: ',3,3,10))
kut = 360.0 / n
for i in range(n):
    forward(100)
    right(kut)
mainloop()
```



Primjer 7. Crtanje pravilnog n-terokuta metodom circle()

```
from turtle import *
n = int(numinput('Mnogokut','Koliko stranica ima mnogokut: ',3,3,10))
circle(100,360,n)
mainloop()
```

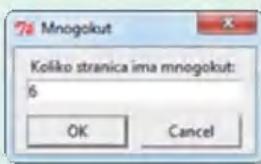


Primjer 8. Ispunjén (obojen) pravilni n-terokut

```
from turtle import *
n = int(numinput('Mnogokut','Koliko stranica ima mnogokut: ',3,3,10))
color('red','red')
kut = 360.0 / n
begin_fill()
for i in range(n):
    forward(100)
    right(kut)
end_fill()
mainloop()
```

Primjer 9. Ispunjén, obojen pravilni n-terokut metodom circle()

```
from turtle import *
color('orange','orange')
begin_fill()
n = int(numinput('Mnogokut','Koliko stranica ima mnogokut: ',3,3,10))
circle(100,360,n)
end_fill()
ht()
mainloop()
```



6. Koordinatna grafika

Primjer 10. RGB kvadrati

```
from turtle import *
def kvadrat(a):
    for x in range(4):
        fd(50)
        rt(90)
    return
boja = ('red','green','blue')
for i in range(3):
    color(boja[i],boja[i])
    begin_fill()
    kvadrat(50)
    end_fill()
    pu()
    fd(50)
    pd()
    ht()
mainloop()
```



Primjer 11.

Crtanje niza jednakostraničnih trokuta koji imaju jedan zajednički vrh i kojima je stranica svakog novog trokuta za 20% dulja od stranice prethodnog trokuta. Duljinu stranice prvog trokuta i broj trokuta zadaje korisnik.

```
from turtle import *
def trokut(a):
    for i in range (3):
        fd(a)
        lt(120)
    return
n=int(numinput('','Koliko trokuta: ',3,3,10))
a=numinput('','Duljina stranice prvog trokuta: ',5,5,100)
pu()
goto(-200,-200)
pd()
ht()
pensize(2)
for i in range(n):
    trokut(a)
    a=1.2*a
mainloop()
```

Primjer 12. Ispis „trokutaste“ spirale

```
from turtle import *
m=numinput('Ulaz','Početna duljina stranice (5, 100): ', 50,
5, 100)
n=int(numinput('Ulaz','Broj ponavljanja (5, 50): ', 20, 5,
100))
pensize(2)
```

(nastavak na idućoj stranici)

6. Koordinatna grafika

```
for i in range (int(n)):
    fd(m)
    lt(120)
    m=m+10
ht()
mainloop()
```



„Trokutasta“ spirala može se ispisati i rekurzivnom funkcijom. Podsjetimo se: rekurzijom se naziva funkcija koja poziva samu sebe. Kod rekurzije je bitno odrediti uvjet izlaska iz rekurzije i korak rekurzije.

Primjer 13. Ispis „trokutaste“ spirale rekurzivnom funkcijom

```
from turtle import *
def trokut(m):
    if m>0:
        fd(m)
        lt(120)
        trokut(m-10)
m = numinput('Ulaz', 'Početna duljina stranice (5, 100): ',
50, 5, 600)
pu()
goto(-200,-300)
pd()
pensize(2)
trocuk(m)
ht()
mainloop()
```

Za vizualizaciju rekurzivne funkcije možemo se koristiti fraktalima. Fraktali su geometrijski oblici poput pravokutnika, kružnica i trokuta, ali s posebnim svojstvom samosličnosti. To znači da fraktal može biti podijeljen u sitne dijelove od kojih je svaki reducirana kopija veće cijeline.

Kochova pahuljica

Kochova pahuljica, koju je 1904. predstavio švedski matematičar Niels Fabian Helge von Koch, jedna je od prvih opisanih fraktalnih krivulja.

Kochova krivulja nastaje kada liniju određene duljine podijelimo na tri jednakaka dijela, a potom srednji dio zamijenimo dvjema novim linijama jednakih duljina. Isti postupak ponavlja se sa svakom od novonastalih linija. Na analogan način dobiva se i Kochova pahuljica počevši od oblika trokuta.

Primjer 14.

Nacrtajte prvih šest iteracija Kochove pahuljice.

```
from turtle import *
def k(a, n):
    if (n == 0):
        fd(a)
    else:
        k(a/3.0, n-1)
```

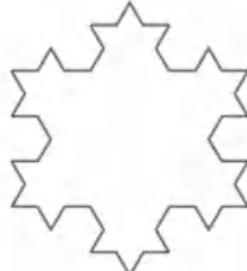
(nastavak na idućoj stranici)

6. Koordinatna grafika

```
lt(60)
k(a/3.0, n-1)
rt(120)
k(a/3.0, n-1)
lt(60)
k(a/3.0, n-1)

def pahuljica(a, n):
    for step in range(3):
        k(a, n)
        rt(120)

pu()
pensize(2)
goto(-200,100)
pd()
for i in range(6):
    pahuljica(300,i)
    clear()
ht()
mainloop()
```



Primjer 15. Olimpijski krugovi

```
from turtle import *
r = 40
pensize(4)
boje = ['blue', 'black', 'red', 'yellow', 'green']
pu()
goto (-50,0)
pd()
z = 0
kraj = 3
for i in range(2):
    for j in range (kraj):
        color (boje[z])
        circle(r)
        z = z+1
        pu()
        fd(85)
        pd()
    pu()
    goto(-15,-50)
    pd()
    kraj = kraj-1
ht()
mainloop()
```

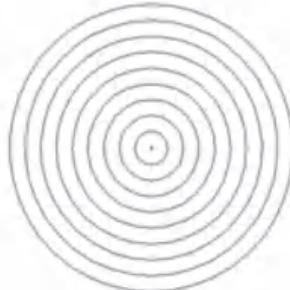


Krugovi u raznim bojama koje simboliziraju Olimpijske igre predstavljaju pet različitih kontinenta. Crni krug predstavlja Afriku, žuti Aziju, plavi Evropu, crveni Ameriku i zeleni Australiju.

6. Koordinatna grafika

Primjer 16. Deset koncentričnih kružnica

```
from turtle import *
title('Koncentrične kružnice')
pu()
for i in range (1,200, 20):
    rt(90)
    fd(i)
    rt(270)
    pd()
    circle(i)
    pu()
    home()
ht()
mainloop()
```



Primjer 17. Meta – deset raznobojnih koncentričnih krugova

```
from turtle import *
title('Meta')
boja = ('black','brown','pink','darkblue','green','red','blue',
', 'orange','yellow','white')
pu()
j = 0
for i in range (200,1, -20):
    rt(90)
    fd(i)
    rt(270)
    pd()
    color(boja[j],boja[j])
    j = j+1
    begin_fill()
    circle(i)
    end_fill()
    pu()
    home()
ht()
mainloop()
```



Primjer 18.

Nacrtajte pravokutni trokut kojem se zadaju duljine kateta.

```
from turtle import *
from math import *
a = numinput('Kateta a','Duljina katete a: ',3,10,400)
b = numinput('Kateta b','Duljina katete b: ',3,10,400)
opposite = 60
adjacent = 80
color('#008000','#008000')
begin_fill()
fd(a)
lt(90)
(nastavak na idućoj stranici)
```

6. Koordinatna grafika

```
fd(b)
tan = b / a
kut = atan(tan) * 180 /pi
kut2 = 90 - kut
okret = 180 - kut2
lt(okret)
c = sqrt(a * a + b * b)
fd(c)
end_fill()
pu()
home()
goto(0,-20)
write('B')
fd(a+10)
write('C')
lt(90)
fd(b+10)
write('A')
ht()
mainloop()
```

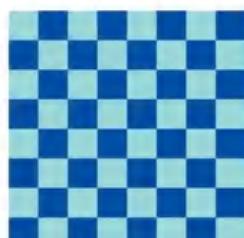


Primjer 19.

Nacrtajte šahovsku ploču koja se sastoji od $n \times n$ kvadrata

```
from turtle import *
def kvadrat(a):
    for x in range(4):
        fd(a)
        rt(90)
    return
boja = ('cyan','blue')
n = int(numinput('Šahovska ploča','Koliko od koliko kvadrata:',3,3,20))
for i in range(n):
    for j in range (n):
        color(boja[(i+j)%2],boja[(i+j)%2])
        begin_fill()
        kvadrat(15)
        end_fill()
        pu()
        fd(15)
        pd()

    pu()
    home()
    rt(90)
    fd(15*(i+1))
    rt(270)
    pd()
    ht()
mainloop()
```

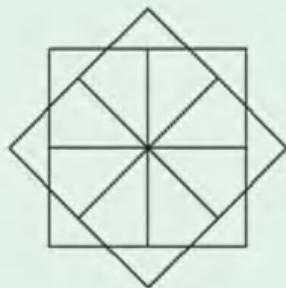


6. Koordinatna grafika

Primjer 20. Zarotirani kvadrati

Nacrtani kvadrat stranice a zaročira se za zadani kut . Postupak se nastavlja sve dok ne obide puni krug.

Primjer je prikazan na slici, stranica je duljine 100, a kut je 45° .



```
from turtle import *
from math import *
def kvadrat(a):
    pensize(2)
    for i in range(4):
        fd(a)
        rt(90)
a = numinput('Ulaz','Duljina stranice kvadrata(20, 400): ', 50, 20, 400)
kut = int(numinput('Ulaz','kut rotacije u stupnjevima (1, 360): ', 90, 1, 360))
for i in range (int(360/kut)):
    kvadrat(a)
    rt(kut)
ht()
mainloop()
```

Premda postoje specijalizirani alati za prikaz podataka dobivenih nekim istraživanjem, u tu svrhu možemo se poslužiti i mogućnostima koje pruža Python.

Primjer 21.

Stupčasti grafikon

Učitane podatke prikažite stupčastim grafikonom.

```
from turtle import *
def bar(v):
    begin_fill()
    lt(90)
    fd(v)
    write(" " + str(v))
    rt(90)
    fd(40)
    rt(90)
    fd(v)
    lt(90)
    end_fill()
    fd(10)
n = int(numinput('','Koliko brojeva: ',5,3,10))
p = []
for i in range(n):
    x = int(numinput('','Zadaj broj: '))
    p.append(x)
color("blue", "cyan")
pu()
goto(-200,0)
pd()
```

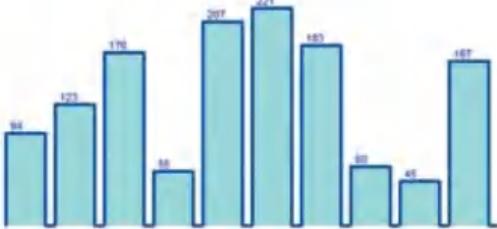
(nastavak na idućoj stranici)

6. Koordinatna grafika

```
pensize(3)
for a in p:
    bar(a)
ht()
mainloop()
```

Grafički prikaz realiziran na taj način ne bi bio upotrebljiv kad bi podaci bili jako mali ili jako veliki. Da bi se taj problem riješio, potrebno je „skalirati“ podatke, odnosno prilagoditi prikaz podacima.

Jedno je od rješenja problema da odredimo najveći podatak (M) koji treba biti prikazan. Ako za njega odredimo da će stupac biti visok N piksela, sve ostale podatke prikazujemo u tom omjeru. Implementacija opisanog prikazana je u sljedećem primjeru.



Primjer 22.

```
from turtle import *
def bar(v,ma):
    begin_fill()
    lt(90)
    fd(v*600/ma)
    write(" " + str(v))
    rt(90)
    fd(40)
    rt(90)
    fd(v*600/ma)
    lt(90)
    end_fill()
    fd(10)
n = int(numinput(' ','Koliko brojevâ: ',5,3,15))
p = []
for i in range(n):
    x = int(numinput(' ','Zadaj broj: '))
    p.append(x)
M = max(p)
color("blue","cyan")
pu()
goto(-350,-300)
pd()
pensize(3)
for a in p:
    bar(a,M)
ht()
mainloop()
```

Za potrebe analize ponekad su praktičniji grafikoni koji prikazuju udio svake vrijednosti u cjelini. Primjer takva grafikona donosi se u sljedećem primjeru.

6. Koordinatna grafika

Primjer 23. Pravokutni grafikon koji prikazuje udio svake vrijednosti u cjelini.

```
from turtle import *
def bar(v,ma,i):
    boje = ('red','green','blue','magenta','maroon','cyan','brown',
    'orange','pink','violet','gray','purple','black','yellow','navy')
    color(boje[i])
    begin_fill()
    lt(90)
    fd(100)
    rt(90)
    fd(v*600/(ma))
    rt(90)
    fd(100)
    rt(90)
    fd(v*600/ma)
    end_fill()
    bk(v*600/ma)
    lt(180)
n = int(numinput('','Koliko brojeva: ',5,3,15))
p = []
for i in range(n):
    x = int(numinput('','Zadaj broj: '))
    p.append(x)
M = max(p)
s = sum(p)
color("blue", "cyan")
pu()
goto(-300,0)
pd()
pensize(3)
i = 0
for a in p:
    bar(a,s,i)
    i+= 1
ht()
mainloop()
```



Za prikaz udjela svake vrijednosti u cjelini češće se upotrebljava kružni grafikon.

Primjer 24. Kružni grafikon koji prikazuje udio svake vrijednosti u cjelini

```
from turtle import *
from random import *
def krug(kut,tekst):
    begin_fill()
    circle(100,kut)
    x = xcor()
    y = ycor()
    goto(x,y)
    goto (p,q)
    end_fill()
(nastavak na idućoj stranici)
```

Kružni grafikon

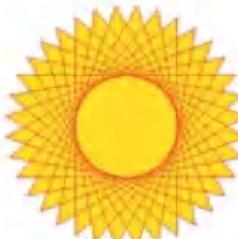
6. Koordinatna grafika

```
goto(x,y)
n = int(numinput('','Koliko brojeva: ',5,3,10))
pp = []
s = 0
for i in range(n):
    x = int(numinput('','zadaj broj: '))
    pp.append(x)
    s = s+x
p,q = 0,100
pd()
pensize(3)
b = 0
boja = ['red','green','orange','violet','brown','navy','magenta','blue','pink','yellow']
k = 0
for a in pp:
    color(boja[b],boja[b])
    b = b+1
    k = 360/s*a
    t = x/s*100
    krug(k,t)
ht()
mainloop()
```

Sljedeća dva primjera pokazuju kako se lako mogu dobiti vrlo atraktivni crteži.

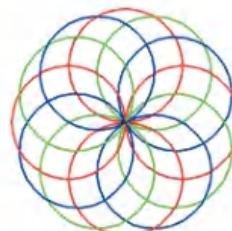
Primjer 25.

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    fd(250)
    lt(130)
    if abs(pos()) < 1:
        break
end_fill()
mainloop()
```



Primjer 26.

```
from turtle import *
pensize(5)
for i in range(4):
    for c in ["red", "green", "blue"]:
        pencolor(c)
        circle(100)
        rt(30)
exitonclick()
```



U grafičkom modu određene funkcije mogu se dodijeliti tipkama miša. Za to se upotrebjava metoda `onclick()` koja registrira aktiviranje jedne od triju tipaka miša i na temelju toga pokreće pridruženu funkciju u Pythonu. Ljeva tipka miša označena je brojem jedan, srednja brojem dva, a desna tipka miša brojem tri. Metoda `onclick()` element je podklase `turtle`. Sljedeći primjer pokazuje upotrebu te metode.

`onclick()`

6. Koordinatna grafika

Primjer 27.

Pritiskom na lijevu tipku miša pero se pomakne za 100 naprijed i ostavi trag, a pritiskom na desnu tipku miša pero se zarotira za 90° udesno.

```
from turtle import *
def naprijed(x,y):
    t1.pd()
    t1.fd(100)
    return
def okreni(x,y):
    t1.pd()
    t1.rt(90)
    return
t1 = Turtle()
t1.onclick(naprijed,1)
t1.onclick(okreni,3)
```

Određene funkcije možemo dodijeliti ne samo tipkama miša već i pojedinim tipkama tipkovnice. Za to se upotrebljava metoda *onkey()* koja je element podklase *screen*. Da bi se osiguralo konstantno praćenje i evidencija koja je tipka pritisnuta, nužna je upotreba metode *listen()*.

Primjer 28.

Kursorskim strelicama pridijeljene su funkcije kretanja naprijed, zakretanja udesno i kretanja naprijed, kretanja nazad te zakretanja uljevo i kretanja naprijed.

```
from turtle import *
t = Screen()
def naprijed():
    fd(30)
    return
def lijevo():
    left(60)
    fd(30)
    return
def desno():
    right(60)
    fd(30)
    return
def nazad():
    bk(30)
    home()
t.onkey(naprijed,"Up")
t.onkey(lijevo,"Left")
t.onkey(desno,"Right")
t.onkey(nazad,"Down")
t.listen()
```

U grafičkom modu turtle programskog jezika Python istovremeno se može raditi s više pera, što pokazuje sljedeći primjer.

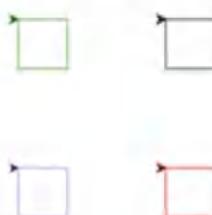
6. Koordinatna grafika

Primjer 29.

Četiri kvadrata, svaki svoje boje, crtaju se istovremeno.

```
from turtle import *
t1 = Turtle()
t2 = Turtle()
t3 = Turtle()
t4 = Turtle()
t1.pencolor('blue')
t1.pencolor('green')
t3.pencolor('red')
t4.pencolor('violet')
t1.pu()
t2.pu()
t3.pu()
t1.goto(0,150)
t2.goto(150,150)
t3.goto(150,0)
t1.pd()
t2.pd()
t3.pd()
for i in range(4):
    t1.fd(50)
    t1.rt(90)
    t2.fd(50)
    t2.rt(90)
    t3.fd(50)
    t3.rt(90)
    t4.fd(50)
    t4.rt(90)
mainloop()
```

istovremenih rad s
više pera

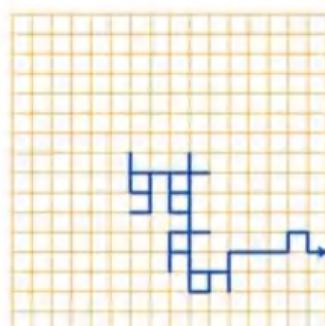


Primjer 30.

Nacrtajte kvadratnu mrežu po kojoj se pero kreće slučajnim izborom sve dok ne dođe do ruba mreže.

```
from turtle import *
from random import *
title ('Šetnja')
color ('orange')
x = -160
for y in range (-160, 160 + 1, 20):
    pu()
    goto (x, y)
    pd()
    fd (320)
y = 160
right (90)
for x in range (-160, 160 + 1, 20):
    pu()
```

modul random



(nastavak na idućoj stranici)

6. Koordinatna grafika

```
        goto (x, y)
        pd()
        fd (320)
pensize(3)
color ('blue')
pu()
goto(0, 0)
pd()
x = y = 0
while (abs(x) < 160 and abs(y) < 160):
    r = randint(0, 3)
    if r == 0:
        x +=20
        setheading(0)
        fd(20)
    elif r == 1:
        y -=20
        setheading(270)
        fd(20)
    elif r == 2:
        x -= 20
        setheading(180)
        fd(20)
    elif r == 3:
        y +=20
        setheading(90)
        fd(20)
mainloop()
```

6.2. Boje u Pythonu

Boja	string	colormode(255)	colormode(1.0)
'black'	'#000000'	(0, 0, 0)	(0.0, 0.0, 0.0)
'white'	'#ffffff'	(255, 255, 255)	(1.0, 1.0, 1.0)
'red'	'#ff0000'	(255, 0, 0)	(1.0, 0.0, 0.0)
'blue'	'#0000ff'	(0, 0, 255)	(0.0, 0.0, 1.0)
'yellow'	'#ffff00'	(255, 255, 0)	(1.0, 1.0, 0.0)
'cyan'	'#00ffff'	(0, 255, 255)	(0.0, 1.0, 1.0)
'magenta'	'#ff00ff'	(255, 0, 255)	(1.0, 0.0, 1.0)
'maroon'	'#800000'	(128, 0, 0)	(0.5, 0.0, 0.0)
'green'	'#00ff00'	(0,255,0)	(0.0, 1.0, 0.0)
'purple'	'#800080'	(128,0,128)	(0.5, 0.0, 0.5)
'navy'	'#000080'	(0,0,128)	(0.0, 0.0, 0.5)
'brown'	'#996611'	(153,102,31)	(0.60, 0.4, 0.12)
'orange'	'#faa01f'	(250, 160, 31)	(0.98, 0.625, 0.12)
'pink'	'#fa0ab2'	(250, 10, 178)	(0.98, 0.04, 0.7)
'violet'	'#9966b2'	(153, 102, 178)	(0.60, 0.4, 0.7)

Do sad smo se već koristili bojama i u izradi korisničkog grafičkog sučelja i u upoznavanju osnovnih metoda sadržanih u modu *turtle* pa ćemo se podsjetiti kako se dolazi do željenih boja i vidjeti kako možemo stvoriti zanimljive efekte njihovom upotrebom.

Kombiniranjem triju osnovnih boja, crvene (Red), zelene (Green) i plave (Blue), nastaju sve boje koje se prikazuju na zaslonu monitora (RGB model). Pri određivanju boje svakog piksela zaslona za svaku od triju osnovnih boja koristi se po jedan bajt, što znači da je boja svakog piksela određena s tri bajta. Zastupljenost pojedine osnovne boje (crvene, zelene, plave) iskazuje se decimalnim brojem u rasponu od 0 do 255, odnosno hek-

6. Koordinatna grafika

sadekadski od 0 do ff. Ako se ti brojevi podijele s 255, dolazimo do prikaza brojevima iz intervala od 0.0 do 1.0. Plava boja u sebi ima 0 crvene i 0 zelene te 255 plave, što se može zapisati (0, 0, 255) ili (0.0, 0.0, 1.0), odnosno kao heksadekadski string '#0000ff'. Ovisno o tome želimo li raditi s cijelim ili decimalnim brojevima, odabrat ćemo odgovarajući RGB prikaz metodom **colormode()**. Odaberemo li cijele brojeve, upotrijebit ćemo colormode(255), odnosno colormode(1.0) za decimalne brojeve.

colormode()

Standardne boje mogu se zadati i kao string (ili string heksadekadski zapisanih vrijednosti RGB komponenti).

Način određivanja osnovnih boja u Pythonu prikazan je sljedećom tablicom.

Primjer 31.

Ispišite dužine u 15 različitih standardnih boja (bijela je izostavljena).

```
from turtle import *
boje = ('black', 'red', 'green', 'blue', 'yellow', 'magenta',
'cyan', 'brown', 'orange', 'pink', 'violet', 'gray', 'maroon',
'purple', 'navy')
j = 0
y = -300
pensize(13)
for i in boje:
    color(i)
    pu()
    goto(-300,y)
    write(boje[j], font = ('times', 12, 'bold'))
    goto (-300, y+30)
    pd()
    goto (200,y+30)
    y = y+40
    j = j+1
mainloop()
```



Primjer 32.

Nacrtajte N raznobojnih krugova pravilno raspoređenih po kružnici polumjera r = 80 i povezanih sa središtem.

```
from turtle import *
from random import *
boja = ['red','magenta','cyan','brown','green','blue','gray',
'orange','navy','violet','yellow','pink','black']
def krug():
    b = randint(0,10)
    bi = (boja[b])
    color(bi)
    fd(80)
    rt(90)
    dot(20,bi)
    lt(90)
(nastavak na idućoj stranici)
```



6. Koordinatna grafika

```
bk(80)
def crtanje():
    n = randint(4,20)
    for i in range(n):
        krug()
        rt(360/n)
crtanje()
ht()
mainloop()
```

Primjer 33. Spirala boja

```
from turtle import *
boja = ["blue", "orange", "purple", "cyan", "red", "violet"]
reset()
tracer(0, 0)
for i in range(50):
    color(boja[i % 6])
    pd()
    fd(2 + i * 5)
    lt(45)
    pensize(i)
    pu()
mainloop()
```



Ukupan broj boja (nijansi) koje možemo prikazati na ovaj način jest $256^3 \times 256^3 = 16777216$.

Sivu boju, primjerice, možemo dobiti miješanjem crne i bijele, a ovisno u udjelu pojedine od njih nastaju različite nijanse sive boje. U colormode(1.0) crna je boja zapisana kao (0.0, 0.0, 0.0), a bijela kao (1.0, 1.0, 1.0), što znači da je razlika udjela svake komponente 1. Ako želimo nacrtati n kvadrata koji su svaki u svojoj nijansi sive boje, dovoljno je da tu razliku = 1 podijelimo na n dijelova i pri svakom novom kvadratu udio svake komponente povećamo za $1/n$, što je vidljivo u sljedećem primjeru.

Primjer 34.

linearna
interpolacija

```
from turtle import *
def kvadrat(a):
    for i in range(4):
        lt(90)
        fd(a)
    return()
colormode(1.0)
c = z = p = 0
a = 600
pu()
goto(-250,0)
pd()
n = int(numinput('Ulaz','Koliko kvadrata: ',5,5,52))
a = a/n
```

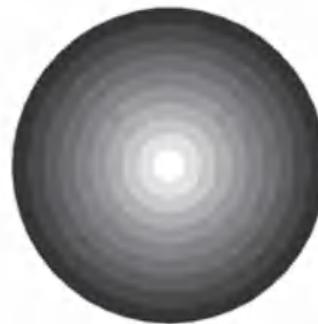
6. Koordinatna grafika

```
for i in range(n):
    color(c,z,p)
    begin_fill()
    kvadrat(a)
    end_fill()
    c = c+1/n
    z = z+1/n
    p = p+1/n
    pu()
    fd(a)
    pd()
ht()
pu()
color(0,0,0)
goto(-200, 150)
s = str(n) + ' nijansi sive'
write(s, font = ('times', 14, 'bold'))
```



Primjer 35. Deset koncentričnih krugova u sivim nijansama

```
from turtle import *
title('Koncentrični krugovi sive nijanse')
colormode(1.0)
pu()
j = 1
for i in range (200,1, -20):
    rt(90)
    fd(i)
    rt(270)
    pd()
    color(j/10,j/10,j/10)
    j = j+1
    begin_fill()
    circle(i)
    end_fill()
    pu()
    home()
ht()
mainloop()
```



Sličnim principom kao iz prethodnog primjera možemo dobiti različite nijanse prijelaza boja između plave i crvene. Jedina je razlika što se u tom slučaju udio crvene komponente treba smanjivati, a udio plave povećavati. Zelena ostaje nepromijenjena.

Primjer 36.

Ispišite n krugova u različitim nijansama boja između crvene i plave raspoređenih tako da tvore krug.

```
from turtle import *
from math import *
colormode(1.0)
c = 1
(nastavak na idućoj stranici)
```

6. Koordinatna grafika

```
z = p = 0
a = 100
pu()
goto(0,-150)
pd()
n = int(numinput('','Koliko krugova: ',5,5,52))
a = a*pi/n
for i in range(n):
    seth(360*i/n)
    color(c,z,p)
    begin_fill()
    circle(a/2)
    end_fill()
    c = c-1/n
    p = p+1/n
    pu()
    fd(2*a)
    pd()
ht()
pu()
```



Primjer 37.

Učitani broj ispišite tako da mu je svaka znamenka ispisana drugom nijansom ovisno o njezinoj vrijednosti (0 – najsvjetlije, 9 – najtamnije).

```
from turtle import *
broj = textinput('','Zadaj broj: ')
colormode(1.0)
pu()
ht()
x = -200
n = len(broj)
i = 0
while i < n:
    b = broj[i]
    nb = ord(b)-ord('0')
    p = 0.6
    z = 0
    c = 1-nb/10
    color(c,z,p)
    i = i+1
    write(b,font = ('Verdana',20,'bold'), move = True)
mainloop()
```

90517386

Postupci prikazani u prethodnim dvama primjerima ustvari predstavljaju linearu interpolaciju između dviju boja. Često se javlja potreba za interpolacijom četiriju boja ili, bolje rečeno, između dvaju parova boja. Tim postupkom bavi se bilinearna interpolacija.

bilinearna
interpolacija

Ideja je bilinearne interpolacije sljedeća. Zamislimo da imamo kvadrat, u donjem lijevom kutu neka se nalazi boja *a*, u donjem desnom kutu boja *b*, u gornjem lijevom kutu boja *c* i u gornjem desnom kutu boja *d*. Podijelimo kvadrat na *n* x *n* manjih kvadratića, od

6. Koordinatna grafika

kojih ćemo svaki obojiti bojom koja će biti boja na prijelazu između zadanih boja. Odmah je jasno da će prijelazi između kvadratiča biti manje uočljivi što je n veći. Odredimo linearnu interpolaciju za donji red, tj. između boja a i b, i za gornji red, tj. između boja c i d. Time smo odredili tzv. vodoravne interpolacije. Sad još preostaje da za svaki „stupac“ napravimo linearnu interpolaciju između boja dobivenih horizontalnom interpolacijom.

Udio crvene boje pri horizontalnoj linearnej interpolaciji računat će se po formuli:

$$c = b_i * cb + (1 - b_i)ca$$

pri čemu je

$$b_i = \frac{1}{n},$$

cb udio crvene boje u boji b,

ca udio crvene boje u boji ca za donji red

i analogno

$$c = b_i * cd + (1 - b_i)cc$$

za gornji red.

Analognim postupkom linearne interpolacije između tako dobivenih boja dolazimo do formule kojom računamo udio crvene boje:

$$c = (1.0 - bi) * (1 - bj) * ca + bi * (1 - bj) * cb + (1 - bi) * bj * cc + bi * bj * cd$$

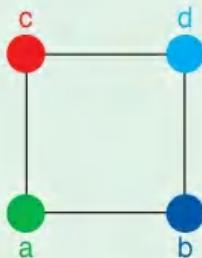
Isti postupak radi se i za udio zelene, odnosno plave boje.

Primjer 38.

Nacrtajte kvadrat sačinjen od 20x20 kvadratiča s bilinearnom interpolacijom boja. Druga slika pokazuje kako bi izgledala interpolacija da se uzelo 100x100 kvadratiča.

```
from turtle import *
def kvadrat(a):
    for i in range(4):
        fd(a)
        lt(90)
    return
n = 20
ca = 0
za = 1
pa = 0
cb = 0
zb = 0
pb = 1
cc = 1
zc = 0
pc = 0
cd = 0
zd = 1
pd = 1
```

(nastavak na idućoj stranici)



6. Koordinatna grafika

```
x = -300
y = -300
a = 5
for i in range(n):
    bi = i/n
    for j in range (n):
        bj = j/n
        c = (1.0-bi)*(1-bj)*ca+bi*(1-bj)*cb+(1-
bi)*bj*cc+bi*bj*cd
        z = (1.0-bi)*(1-bj)*za+bi*(1-bj)*zb+(1-
bi)*bj*zc+bi*bj*zd
        p = (1.0-bi)*(1-bj)*pa+bi*(1-bj)*pb+(1-
bi)*bj*pc+bi*bj*pd
        color(c,z,p)
        pu()
        goto(x+i*a,y+j*a)
        begin_fill()
        kvadrat(a)
        end_fill()
    ht()
mainloop()
```

6.3. Grafički prikaz matematičkih funkcija

Za svaku realnu funkciju treba promatrati uređeni par $(x, f(x))$ kojem smo u koordinatnoj ravnini pridružili točku $T(x, y)$. Skup tako dobivenih točaka predstavlja graf funkcije f .

U sljedećih nekoliko primjera vidjet ćemo kako se može dobiti graf linearne i kvadratne funkcije te trigonometrijskih funkcija.

Primjer 39.

Nacrtajte graf linearne funkcije $f(x) = ax + b$, koeficijente a i b zadaje korisnik.

```
from turtle import *
def duzina (x1, y1, x2, y2):
    pu()
    goto (x1, y1)
    pd()
    goto (x2, y2)
    pu()

def oznaka_tocke (x, y, label):
    pu()
    goto (x, y)
    pd()
    write (label)
    pu()

def podjela_osi (x, y, okomito):
    if okomito :
```

(nastavak na idućoj stranici)

6. Koordinatna grafika

```
duzina (x, y + 5, x, y - 5)
else:
    duzina (x - 5, y, x + 5, y)

def numeriranje (x, y, okomito, text):
    if okomito:
        oznaka_tocke (x - 20, y - 20, text)
    else:
        oznaka_tocke (x + 20, y, text)

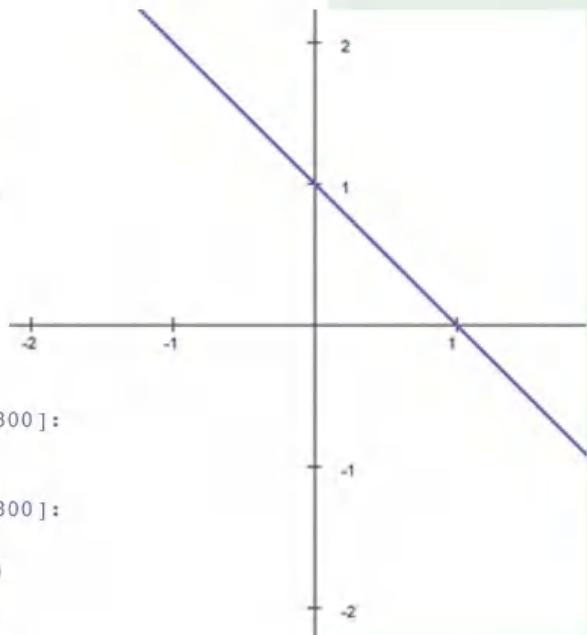
def osi():
    duzina (-400, 0, 400, 0)
    duzina (0, 400, 0, -400)

    for x in [-300, -200, -100, 100, 200, 300]:
        podjela_osi (x, 0, True)
        numeriranje (x+15, 0, True, x/100)
    for y in [-300, -200, -100, 100, 200, 300]:
        podjela_osi (0, y, False)
        numeriranje (0, y-10, False, y/100)

def crtanjefunkcije (fn, od, do, korak):
    pu()
    x = od
    y = fn (x)
    sX = x * 100
    sY = y * 100
    goto (sX, sY)
    pd()
    while x < do:
        x = x + korak
        y = fn (x)
        sX, sY = x * 100, y * 100
        goto (sX, sY)
    pu()

def fja (x):
    return (a*x +b)
title ('Graf linearne funkcije')
a = numinput ('','Koeficijent smjera: ')
b = numinput ('','Odsječak na osi y: ')
osi()
pencolor ('purple')
pensize(2)
crtanjefunkcije (fja, -4.5, 4.5, 0.01)
mainloop()
```

Problem smo rastavili na nekoliko bitnih elemenata pa se stoga program sastoji od nekoliko korisničkih funkcija. Prva je od njih funkcija *duzina* koja spaja dvije točke kojima se nacrtaju koordinatne osi. *Podjela_osi* i *numeriranje* funkcije su kojima se na koordinatnim osima označavaju i numeriraju dijelovi, a u skladu sa skaliranjem koje je nužno da bi se na grafu funkcije mogli uočiti najvažniji elementi. Funkcija *fja* omogućava korisniku da definira funkciju čiji graf želi nacrtati.



graf linearne funkcije

6. Koordinatna grafika

Primjer 40.

Promijeni funkciju $f(x)$ tako da nacrti graf kvadratne funkcije $f(x) = x^2 - x - 2$.

graf kvadratne funkcije

```
from turtle import *
from math import *

def duzina (x1, y1, x2, y2):
    pu()
    goto (x1, y1)
    pd()
    goto (x2, y2)
    pu()

def oznaka_tocke (x, y, label):
    pu()
    goto (x, y)
    pd()
    write (label)
    pu()

def podjela_osi (x, y, okomito):
    if okomito :
        duzina (x, y + 5, x, y - 5)
    else:
        duzina ( x - 5, y, x + 5, y)

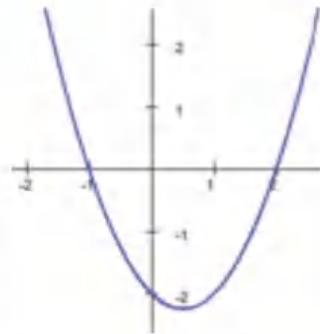
def numeriranje ( x, y, okomito, text):
    if okomito:
        oznaka_tocke ( x - 20, y - 20, text)
    else:
        oznaka_tocke ( x + 20, y, text)

def osi():
    duzina (-400, 0, 400, 0)
    duzina ( 0, 400, 0, -400)

    for x in [-300,-250, -200,-150, -100,-50,50, 100,
150,200,250, 300]:
        podjela_osi ( x, 0, True)
        numeriranje ( x+15, 0, True, x//50)

    for y in [-300,-250, -200,-150, -100,-50,50, 100,
150,200,250, 300]:
        podjela_osi ( 0, y, False)
        numeriranje ( 0, y-10, False, y//50)

def crtanjefunkcije ( fn, od, do, korak):
    pu()
    x = od
    y = fn (x)
    sx = x * 50
    sy = y * 50
    goto (sx, sy)
    pd()
```



(nastavak na idućoj stranici)

6. Koordinatna grafika

```
while x < do:
    x = x + korak
    y = fn ( x )
    sX, sY = x * 50, y * 50
    goto ( sX, sY )
    pu()

def fja (x):
    return (x*x-x -2)

title ('Graf funkcije')
osi()

pencolor ('purple')
pensize(2)
crtanjefunkcije ( fja, -4, 4, 0.01)
mainloop()
```

Primjer 41.

Graf trigonometrijskih funkcija sinus i kosinus

```
from turtle import *
from math import *

def duzina (x1, y1, x2, y2):
    pu()
    goto (x1, y1)
    pd()
    goto (x2, y2)
    pu()

def oznaka_tocke (x, y, label):
    pu()
    goto (x, y)
    pd()
    write (label)
    pu()

def podjela_osi (x, y, okomito):
    if okomito :
        duzina (x, y + 5, x, y - 5)
    else:
        duzina ( x - 5, y, x + 5, y)

def numeriranje ( x, y, okomito, text):
    if okomito:
        oznaka_tocke ( x - 20, y - 20, text)
    else:
        oznaka_tocke ( x + 20, y, text)

def osi():
    duzina (-400, 0, 400, 0)
    duzina ( 0, 400, 0, -400)

    for x in [-300, -200, -100, 100, 200, 300]:
        podjela_osi ( x, 0, True)
```

(nastavak na idućoj stranici)

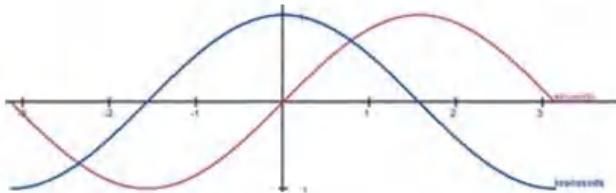
6. Koordinatna grafika

graf funkcija
 $\sin(x)$ i $\cos(x)$

```
numeriranje ( x+15, 0, True, x//100)
for y in [-300, -200, -100, 100, 200, 300]:
    podjela_osi ( 0, y, False)
    numeriranje (0, y-10, False, y//100)

def crtanjefunkcije ( fn, od, do, korak):
    pu()
    x = od
    y = fn (x)
    sX = x * 100
    sY = y * 100
    goto (sX, sY)
    pd()
    while x < do:
        x = x + korak
        y = fn ( x )
        sX, sY = x *
        goto (sX, sY)
    pu()

    pensize(2)
    title ('Sinusoida i kosinusoida')
    osi()
    pencolor ('maroon')
    crtanjefunkcije (sin, -pi, pi, 0.01)
    write('sinusoida')
    pencolor ('blue')
    crtanjefunkcije (cos, -pi, pi, 0.01)
    write('kosinusoida')
    ht()
    mainloop()
```



Funkcija $y = f(x)$ zadana je u parametarskom obliku ako su x i y zadani u eksplisitnom obliku kao funkcije neke pomoćne varijable t koju zovemo parametrom.

Primjer 42.

Nacrtajte graf funkcije *bicorn* zadane parametarski

```
from turtle import *
from math import *
from turtle import *
from math import *
pensize(2)
color ('maroon')
t = 0.01
a = 100
pu()
x = 0
y = a*(cos(t))**2*(2+cos(t))/(3+(sin(t))**2)
goto(x,y)
pd()
while t <=2*pi:
    x = a*sin(t)
    y = a*(cos(t))**2*(2+cos(t))/(3+(sin(t))**2)
    goto(x,y)
(nastavak na idućoj stranici)
```

$$y = \frac{x = a \sin t}{a \cos^2 t \cdot (2 + \cos t)} \quad 3 + \sin^2 t$$



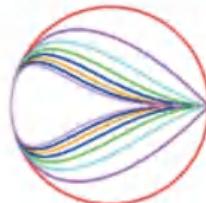
6. Koordinatna grafika

```
y = a*(cos(t))**2*(2+cos(t))/(3+(sin(t))**2)
goto(x,y)
t = t+0.01
ht()
```

Primjer 43.

Nacrtajte graf teardrop krivulje zadane jednadžbom $\begin{cases} x=\cos t \\ y=\sin t \cdot \sin^m\left(\frac{t}{2}\right) \end{cases}$ za sve vrijednosti eksponenta m sa segmenta $[0,7]$.

```
from turtle import *
from math import *
pensize(2)
boja = ['red','magenta','cyan','green','blue','orange','navy','violet']
t = 0.01
a = 100
pu()
m = 0
x = a*cos(t)
y = a*sin(t)*(sin(t/2))**m
goto(x,y)
pd()
for m in range (8):
    t = 0.01
    color(boja[m])
    while t <=2*pi:
        x = a*cos(t)
        y = a*sin(t)*(sin(t/2))**m
        goto(x,y)
        t = t+0.01
ht()
```

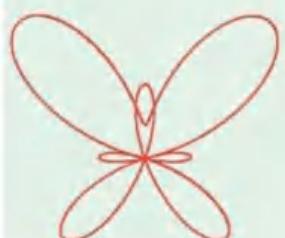


Primjer 44.

Nacrtajte graf funkcije Butterfly zadane:

$$\begin{cases} x=\sin t \left[e^{\cos t} - 2\cos(4t) + \sin^5\left(\frac{1}{15}t\right) \right] \\ y=\cos t \left[e^{\cos t} - 2\cos(4t) + \sin^5\left(\frac{1}{15}t\right) \right] \end{cases}$$

```
from turtle import *
from math import *
pensize(2)
color ('red')
t = 0.001
a = 50
pu()
x = a*sin(t)*(exp(cos(t))-2*cos(4*t) + (sin (t/12))**5)
y = a*cos(t)*(exp(cos(t))-2*cos(4*t) + (sin (t/12))**5)
goto(x,y)
pd()
while t <=2*pi:
    x = a*sin(t)*(exp(cos(t))-2*cos(4*t) + (sin (t/12))**5)
    y = a*cos(t)*(exp(cos(t))-2*cos(4*t) + (sin (t/12))**5)
    goto(x,y)
    t = t+0.001
ht()
```



Kao što je pokazao prošli primjer, za različite oblike i pojave u prirodi postoje matematičke krivulje koje ih opisuju.

6. Koordinatna grafika

Primjer 45.

Nacrtajte srce rabeći matematičke formule:

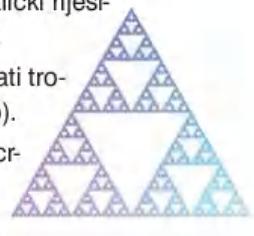
$$\begin{cases} x=16\sin^3 t \\ y=13\cos t - 5\cos(2t) - 2\cos(3t) - \cos(4t) \end{cases}$$

```
from turtle import *
from math import *
pensize(20)
color ('red')
t = 0.01
a = 10
pu()
x = a*16*(sin(t))**3
y = a*(13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t))
goto(x,y)
pd()
while t <=2*pi:
    x = a*16*(sin(t))**3
    y = a*(13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t))
    goto(x,y)
    t = t+0.01
ht()
```



Zadaci za samostalni rad

1. Napišite program koji će nacrtati pravokutnik stranica a i b (a i b zadaje korisnik nakon pokretanja programa).
2. Napišite program koji će učitati duljinu stranice prvog kvadrata i broj stranica i nacrtati kvadratnu spiralu (analognu trokutastoj spirali iz Primjera 11.).
3. Napišite program koji će nacrtati „djeticinu s četiri lista“ kao na slici.

4. Napišite program koji će nacrtati zvijezdu.
5. Rekursivnom funkcijom nacrtaj n stepenica širine i visine d .
6. Napišite program koji će nad svakom stranicom kvadrata a nacrtati jednakostranični trokut.
7. Napišite program koji će nad svakom stranicom pravokutnog trokuta zadanih katetama a i b nacrtati kvadrat.
8. Napišite program koji će nacrtati prvi n koraka kvadratne spirale.
9. Napišite program koji će nacrtati n kvadrata koji imaju zajednički donji lijevi vrh, od kojih je prvi stranica a , a svaki sljedeći ima stranicu za 10% veću od prethodnog.
10. Napišite program koji će nacrtati znak opasnosti od radioaktivnosti.
11. Napišite program koji će nacrtati jing-jang.
12. Napišite program koji će grafički riješiti sustav linearnih jednadžbi.
13. Napišite program koji će crtati trokut Sierpińskog (slika desno).
14. Napišite program koji će nacrtati graf funkcije astroid (slika desno).



6. Koordinatna grafika

Od problema do rješenja

U ovom čemu poglavljiju prezentirati dva zanimljiva problema kroz čija ćeemo rješenja pokazati kako se na zanimljiv način može koristiti i iskoristiti grafika u Pythonu.

Problem 1.

Svi znamo što je broj π . U nekom trenutku saznali smo da je trenutno poznato pet triljuna znamenki tog broja. Pred sebe ćemo postaviti sljedeći problem: Kako grafički prikazati decimale tog broja?

Puno je različitih ideja i načina na koje možemo odgovoriti na postavljeno pitanje. Ovdje ćemo pokazati dva rješenja.

Za početak, promatrat ćemo samo prvih 1000 decimala broja π .

Prvo ćemo nacrtati pravokutnu mrežu koja će biti podijeljena na $r \times s$ kvadrata zadane širine d piksela. Pri tome mora vrijediti da je $r \cdot s \leq 1000$. Svakom kvadratu pridružit ćemo jednu od znamenki, ali tako da je prva znamenka broja pridružena prvom kvadratu u prvom retku, druga drugom kvadratu u prvom retku i tako sve do $r \cdot s$ znamenke, koja će biti pridružena s -tom kvadratu u r -tom retku. Npr. za $r=5$ i $s=6$ vrijedi:

Svakoj znamenci pridružit ćemo jednu nijansu osnovne boje po želji. Neka to bude crvena boja. Na kraju, svaki ćemo kvadrat obojiti onom nijansom crvene boje koja je pridružena toj znamenici.

1	4	1	5	9	2
6	5	3	5	8	9
7	9	3	2	3	8
4	6	2	6	4	3
3	8	3	2	7	9

Programski kod:

```
import random
from turtle import *

def kvadrat(a):
    for x in range(4):
        fd(a)
        rt(90)
    return

colormode(1.0)
boja = [(0.5,0.0,0.0),(0.55,0.0,0.0),
         (0.6,0.0,0.0),(0.65,0.0,0.0),
         (0.7,0.0,0.0),(0.75,0.0,0.0),
         (0.8,0.0,0.0),(0.85,0.0,0.0),
         (0.9,0.0,0.0),(1.0, 0.0,0.0)]
```

(nastavak na idućoj stranici)



3.141592653589793
23846264338327950
28841971693993751
05820974944592307816
406286208998628034825342
117067982148086513282306
647093844609550582231725
359408128481117450284102
701938521105559644622948
954930381964428810975665
933446128475648233786783
165271201909145648566923
460348610454326648213393
607260249141273724587006
606315588174881520920962
829254091715364367892590
360011330530548820466521
384146951941511609433057
270365759591953092186117
381932611793105118548074
462379962749567351885752
724891227938183011949129
833673362440656643086021
394946395224737190702179
860943702770539217176293
176752384674818467669405
132000568127145263560827
785771342757789609173637
178721468440901224953430
146549585371050792279689
258923542019956112129021
960864034418159813629774
771309960518707211349999
998372978049951059731732
816096318595024459455346
908302642522308253344685
035261931188171010003137
838752886587533208381420
617177669147303598253490
428755468731159562863882
353787593751957781857780
532171226806613001927876
6111959092164201989

6. Koordinatna grafika

```
r = int(input('Broj redaka: '))
s = int(input('Broj stupaca: '))
d = int(input('Dimenzija kvadrata: '))

pi = ['14159265358979323846264338327950288419716939937510',
      '58209749445923078164062862089986280348253421170679',
      '82148086513282306647093844609550582231725359408128',
      '48111745028410270193852110555964462294895493038196',
      '44288109756659334461284756482337867831652712019091',
      '45648566923460348610454326648213393607260249141273',
      '72458700660631558817488152092096282925409171536436',
      '78925903600113305305488204665213841469519415116094',
      '33057270365759591953092186117381932611793105118548',
      '07446237996274956735188575272489122793818301194912',
      '98336733624406566430860213949463952247371907021798',
      '60943702770539217176293176752384674818467669405132',
      '00056812714526356082778577134275778960917363717872',
      '14684409012249534301465495853710507922796892589235',
      '42019956112129021960864034418159813629774771309960',
      '5187072113499999837297804995105973173281609631859',
      '50244594553469083026425223082533446850352619311881',
      '71010003137838752886587533208381420617177669147303',
      '59825349042875546873115956286388235378759375195778',
      '18577805321712268066130019278766111959092164201989']

znamenka = 0
for i in range(r):
    for j in range(s):
        z = int(pi[znamenka // 50][znamenka % 50])
        print(z)

        color(boja[z], boja[z])
        begin_fill(); kvadrat(d); end_fill()

        pu(); fd(d); pd()

        znamenka += 1

        pu(); home(); rt(90)
        fd((d+1)*(i+1)); rt(270); pd(); ht()

mainloop()
```

6. Koordinatna grafika

Pokažimo neka od dobivenih umjetničkih djela. Možda se u nekom od njih krije tajna poruka koju nam priroda kroz π želi poslati, a možda i ne. Pogledajmo!



10x100

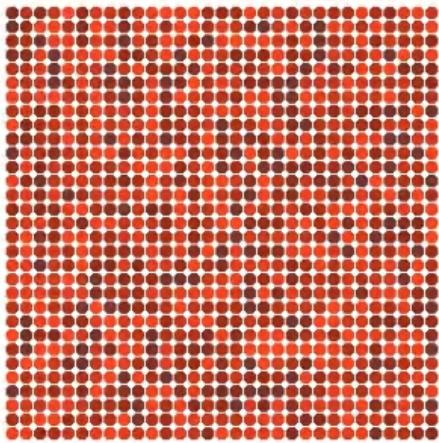
Možda će slike biti jasnije ako umjesto kvadrata upotrijebimo krugove!

Programski kod:

```
znamenka = 0
for i in range(r):
    for j in range (s):
        print(znamenka, znamenka // 100, znamenka % 100)
        z = int(pi[znamenka // 100][znamenka % 100])

        dot(d,boja[z])

    pu()
    fd(d)
    pd()
    znamenka += 1
pu()
home()
rt(90)
fd(d*(i+1))
rt(270)
pd()
ht()
mainloop()
```



Kao primjer izabrali smo umjetničko djelo koje se sastoji od 31x31 krugova.

Ovo su bile samo neke od mogućnosti i samo neki od mogućnih primjera. Pozivamo čitatelja da bude kreativan i osmisli svoj način za grafičko prikazivanje decimala broja π .

Problem 2.

Benfordov zakon donosi jednu jako zanimljivu tvrdnju. Ako imamo jako puno brojeva koji su prikupljeni iz nekog prirodnog izvora, tada se zna da će oko 30.1% tih brojeva početi znamenkom jedan, a samo oko 4.9% znamenkom devet. Pri tome zakon donosi i točne postotke pojavljivanja svih ostalih znamenki:

1	2	3	4	5	6	7	8	9
30.1%	17.6%	12.5%	9.7%	7.9%	6.7%	5.8%	5.1%	4.6%

Tu fascinantnu činjenicu uočio je i prezentirao 1938. fizičar Frank Benford nakon što je proučio gotovo 20.229 skupina različitih podataka. Benfordovu zakonu podliježu podaci prikupljeni iz prirodnih izvora i podaci iz stvarnog života.



10x10



23x10

6. Koordinatna grafika



Danas se zakon primjenjuje u otkrivanju prijevara u ekonomskim poslovnim vodama. U SAD-u je zakon prihvaćen kao pravovaljani dokaz u sudskim postupcima i njime se koristi za dokazivanje poreznih prevara.

Fibonacci brojevi jedan su od takvih nizova brojeva za koje se Benfordov zakon pokazuje istinitim. Promotrimo prvih 100, 1000 i 10000 Fibonaccijevih brojeva i prebrojimo koliko se puta u njima jedinica pojavljuje kao vodeća znamenka, koliko puta dvojka i tako sve do devetke. Vrijedi sljedeća tablica:

Z	1	2	3	4	5	6	7	8	9
Benfordov postotak	30.1%	17.6%	12.5%	9.7%	7.9%	6.7%	5.8%	5.1%	4.6%
Broj pojavljivanja za N = 100	29	18	13	9	9	6	5	7	4
Broj pojavljivanja za N = 1000	300	177	125	96	80	67	57	53	45
Broj pojavljivanja za N = 10000	3011	1762	1250	968	792	668	580	513	456

Uočimo da je podudaranje stvarnih i očekivanih vrijednosti uistinu fascinantno.

Programski kod:

U ovom rješenju prisjetit ćemo se algoritma za određivanje Fibonaccijevih brojeva i primjera 21. iz prošlog poglavlja.

```
import sys
sys.setrecursionlimit(10000) # povećavamo broj rekurzivnih pozivanja
from turtle import *

def f(N):
    global MEMO;
    if(MEMO[N] != -1):
        return MEMO[N];
    if(N == 1):
        MEMO[N] = 1; return MEMO[N]
    elif(N == 2):
        MEMO[N] = 2; return MEMO[N]
    else:
        MEMO[N] = f(N - 1) + f(N - 2)
    return MEMO[N]

def bar(v,ma):
    begin_fill()
    lt(90)
    fd(v*600/ma)
    write(" " + str(v))
    rt(90)
    fd(40)
    rt(90)
    fd(v*600/ma)
    lt(90)
```

(nastavak na idućoj stranici)

6. Koordinatna grafika

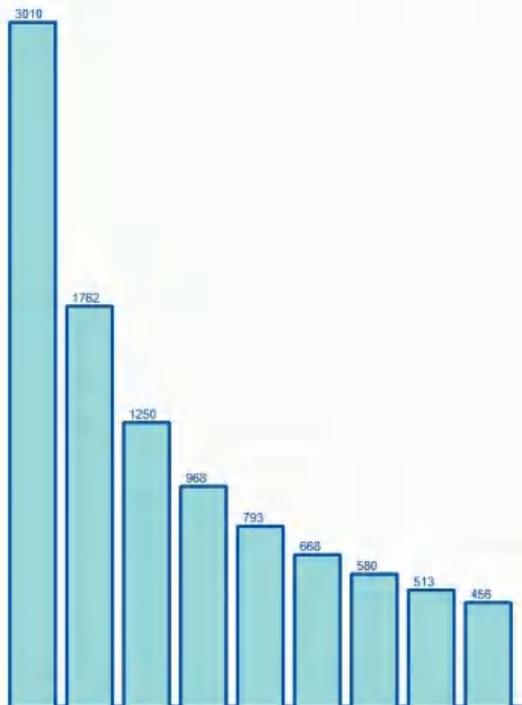
```
end_fill()
fd(10)

znamenke = [0] * 9
MEMO = [-1]*(10001);

for i in range(1,10001):
    x = f(i)
    znamenke[int(str(x)[0])-1] += 1

print(znamenke)
M = max(znamenke)
color("blue", "cyan")
pu()
goto(-350,-300)
pd()
pensize(3)
for a in znamenke:
    bar(a,M)
ht()
mainloop()
```

Dobivene vrijednosti za prvih 10000 brojeva grafički smo prikazali kao na slici.



7. Složenost algoritama

7.1. Što je složenost algoritama?

brzina izvršavanja i
količina memorije

vremenska
složenost

Kako napredujemo s učenjem programiranja u Pythonu, postajemo sposobni rješavati sve komplikirane i raznovrsnije zadatke. Uz nekoliko iznimki, do sada nam je osnovni cilj bio napisati rješenje koje na bilo koji način daje odgovor na neki postavljeni problem. Međutim, nisu sva rješenja ista (naravno, uspoređujemo samo ona rješenja koja daju točne odgovore). Računala imaju dva osnovna ograničenja: brzinu izvršavanja pojedinih naredbi i količinu memorije koja im je na raspolaganju. Suvremena računala mogu izvršavati milijarde elementarnih operacija (poput zbrajanja dvaju brojeva) u samo jednoj sekundi i u memoriji čuvati milijarde brojeva. Iako se ta ograničenja čine jako velikima, već smo u poglavljiju o rekurzijama uspjeli osjetiti njihove posljedice, pa smo npr. na izračunavanje 37. Fibonnacijske brojeve u jednom slučaju morali čekati čak desetak sekundi. No, tamo smo napisali i drugo rješenje koje je dalo odgovor gotovo trenutno. Očito je da su bolja rješenja ona koja do točnog odgovora dolaze izvršavanjem manjeg broja naredbi i korištenjem manje količine memorije.

Zbog toga je vrlo važno da naučimo unaprijed procijeniti koliko će se dugo rješenje (algoritam) koje napišemo izvoditi jednom kada ga pokrenemo i koliko će memorije zauzimati tijekom izvođenja. U ovom ćemo se poglavljju ponajprije baviti procjenom trajanja izvršavanja algoritma. Analizom algoritma moći ćemo odrediti **vremensku složenost** svojeg rješenja i prije nego što ga pokrenemo. Prije nego što točno definiramo što taj pojam znači, pogledajmo za motivaciju dva primjera.

Primjer 1: Zbroj prvih N brojeva

Zadatak je vrlo jednostavan: napisati funkciju koja prima broj N, a vraća zbroj prvih N prirodnih brojeva.

Prvo rješenje koje nam pada na pamet jest sljedeće:

```
def zbroj_1( N ):
    suma = 0;
    for i in range(1, N+1):
        suma = suma + i;
    return suma;
```

Međutim, možemo iskoristiti poznatu formulu $1+2+\dots+N = \frac{1}{2}N(N+1)$ i s pomoću nje napisati drugo rješenje:

```
def zbroj_2( N ):
    suma = N*(N+1)//2;
    return suma;
```

7. Složenost algoritama

Koja je razlika? Napišimo mali program koji će izmjeriti koliko dugo traje izvršavanje svake od funkcija za $N=1\ 000\ 000$, $N=2\ 000\ 000$, ..., $N=10\ 000\ 000$.

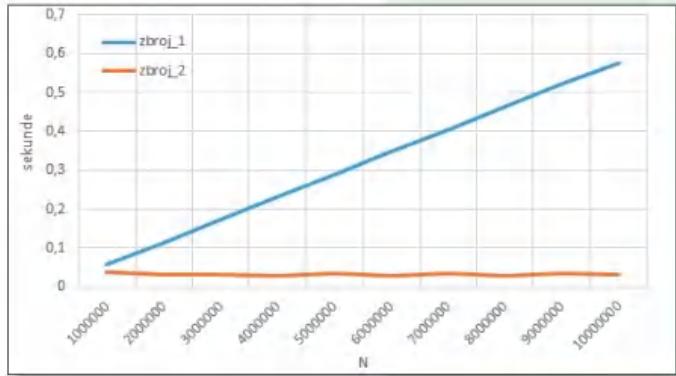
```
for N in range(1000000, 10000001, 1000000):
    print('N=', N)
    t1=time.clock(); k=zbroj_1( N ); t2=time.clock()
    print('zbroj_1:', t2-t1, 'sekundi' )
    t1=time.clock(); k=zbroj_2( N ); t2=time.clock()
    print('zbroj_2:', t2-t1, 'sekundi' )
```

import time

time.clock()

Na svojem smo računalu dobili sljedeći rezultat:

N	zbroj_1	zbroj_2
1000000	0.0585	0.000037
2000000	0.1138	0.000032
3000000	0.1744	0.000033
4000000	0.2336	0.000027
5000000	0.2891	0.000036
6000000	0.3468	0.000029
7000000	0.4053	0.000036
8000000	0.4643	0.000028
9000000	0.5225	0.000034
10000000	0.5772	0.000031



Na grafu smo prikazali kako se vrijeme izvođenja programa (na y-osi) odnosi prema parametru N koji mjeri „veličinu ulaznih podataka“, odnosno „veličinu problema“ (na x-osi). Kod funkcije $zbroj_1$ vrijeme izvođenja raste „po pravcu“ kako raste N . Kod funkcije $zbroj_2$ vrijeme je izvođenja manje-više konstantno i uopće ne ovisi o tome koliko je velik N . Očito je da funkcija $zbroj_2$ predstavlja puno bolje rješenje.

Ta razlika nastaje zbog različitog broja elementarnih operacija koje se izvode u svakoj od funkcija. Funkcija $zbroj_2$ uvijek, neovisno o broju N , izvršava točno četiri elementarne operacije: jedno množenje, jedno zbrajanje, jedno dijeljenje i jedno kopiranje izračunate vrijednosti s desne strane u varijablu sume. Zapišimo taj ukupan broj ovako:

$$T_{zbroj_2}(N) = 4.$$

Funkcija $zbroj_1$ složenija je: na početku imamo jedno kopiranje vrijednosti 0 u varijablu sume, a zatim se za svaku od N vrijednosti varijable izvede jedno zbrajanje i jedno kopiranje izračunate vrijednosti s desne strane u varijablu sume. Dakle,

$$T_{zbroj_1}(N) = 1 + 2N.$$

Ako nacrtamo grafove funkcija T_{zbroj_1} i T_{zbroj_2} , dobit ćemo sliku vrlo sličnu gornjoj! Zapravo, sliku vrlo sličnu gornjoj dobit ćemo i npr. kad je $T_{zbroj_2}(N)=1$ ili $T_{zbroj_2}(N)=23$ te $T_{zbroj_1}(N) = 3N+4$ ili $T_{zbroj_1}(N) = 7N-5$. Ono što je jedino bitno jest da je $T_{zbroj_2}(N)$ konstantan, tj. da ne ovisi o N , a da je $T_{zbroj_1}(N)$ nekakva **linearna funkcija** u ovisnosti o N .

7. Složenost algoritama

Još jedan način promatranja jest sljedeći: ako udvostručimo N, vrijeme izvođenja funkcije `zbroj_1` također će se otprilike udvostručiti, dok se vrijeme izvođenja za `zbroj_2` neće promijeniti.

Primjer 2: Broj pojavljivanja slova u riječi

Zadatak je napisati funkciju koja će primiti string koji se sastoji samo od malih slova i vratiti broj pojavljivanja slova koje se javlja najviše puta u stringu. Na primjer, ako funkcija primi string 'ananas', treba vratiti broj 3.

Prvo rješenje koje nam pada na pamet jest sljedeće: redom za svako slovo primljenog stringa prebrojimo koliko se puta ono ponavlja u riječi:

```
def prebroji_1( s ):
    max_ponavljanja = 0;
    for i in range(0, len(s)):
        broj_ponavljanja = 0;
        for j in range(0, len(s)):
            if( s[i] == s[j] ):
                broj_ponavljanja = broj_ponavljanja + 1;
        if( broj_ponavljanja > max_ponavljanja ):
            max_ponavljanja = broj_ponavljanja;
    return max_ponavljanja;
```

Nakon malo razmišljanja vidimo da ćemo npr. u riječi 'ananas' nepotrebno tri puta brojati broj pojavljivanja slova 'a' i dva puta broj pojavljivanja slova 'n'. Umjesto toga, možemo za svako od 26 malih slova engleske abecede prebrojiti koliko se puta pojavljuje:

```
def prebroji_2( s ):
    sva_slova = 'abcdefghijklmnopqrstuvwxyz';
    max_ponavljanja = 0;
    for i in range(0, len(sva_slova)):
        broj_ponavljanja = 0;
        for j in range(0, len(s)):
            if( s[j] == sva_slova[i] ):
                broj_ponavljanja = broj_ponavljanja + 1;
        if( broj_ponavljanja > max_ponavljanja ):
            max_ponavljanja = broj_ponavljanja;
    return max_ponavljanja;
```

`random.choice()`

Napišimo program koji na slučajan način stvara riječi duljina 1000, 2000, ..., 10000 i pogledajmo koliko dugo traje izvršavanje naših dviju funkcija. Funkcija `random.choice` na slučajan će način odabrat i vratiti jedan od znakova u stringu koji dobije kao parametar.

```
for N in range(1000, 10001, 1000):
    s = '';
    for i in range(0, N):
```

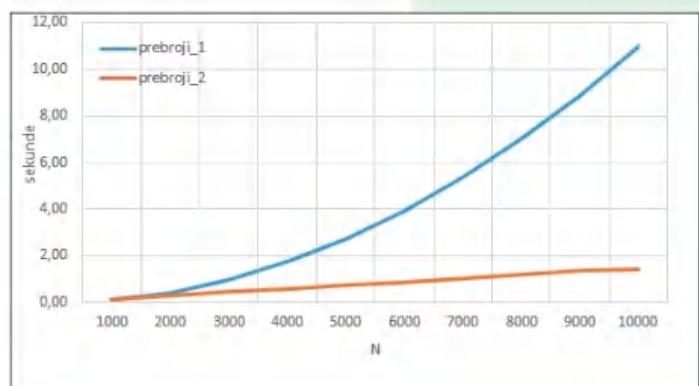
7. Složenost algoritama

```
s = s + random.choice( 'abcdefghijklmnopqrstuvwxyz' );

print( 'N=' , N );
t1=time.clock(); k=prebroji_1(s); t2=time.clock();
print('prebroji_1', t2-t1, 'sekundi');
t1=time.clock(); k=prebroji_2(s); t2=time.clock();
print('prebroji_2' , t2-t1, 'sekundi');
```

Naizgled vrlo mala razlika u funkcijama prebroji_1 i prebroji_2 rezultira drastično različitim vremenima izvršavanja:

N	prebroji_1	prebroji_2
1000	0.1065	0.0028
2000	0.4265	0.0053
3000	0.9731	0.0090
4000	1.7313	0.0113
5000	2.7134	0.0149
6000	3.9133	0.0170
7000	5.3545	0.0203
8000	7.0203	0.0235
9000	8.8728	0.0270
10000	10.9715	0.0289



Prebrojimo broj operacija u objema funkcijama. Neka je N duljina stringa. U funkciji prebroji_1 tijelo vanjske petlje izvest će se N puta. U tom tijelu imamo jedno kopiranje nule u varijablu broj_ponavljanja i unutarnju petlju. Tijelo unutarnje petlje ukupno će se izvesti N^2 puta: prvo se za $i=0$ „izvrte“ sve vrijednosti $j=0, 1, \dots, N-1$, pa za $i=1$ ponovno sve vrijednosti $j=0, 1, \dots, N-1$ itd. Dakle, operacija uspoređivanja unutar if-naredbe izvest će se ukupno N^2 puta. Koliko će se puta povećati varijabla broj_ponavljanja? To ovisi o istinitosti uvjeta u if-naredbi. Kod analize algoritma gotovo se uvijek promatra **najgori mogući slučaj** – dakle, pretpostaviti ćemo da će se dvije operacije (zbroj s 1 i kopiranje) izvesti svaki od N^2 puta. Na kraju, unutar vanjske petlje imamo još ukupno N uspoređivanja u if-u i N kopiranja unutar tijela. Ukupan je broj operacija, dakle,

$$T_{\text{prebroji_1}}(N) = 1 + N + N^2 + 2N^2 + N + N = 3N^2 + 3N + 1.$$

Vidimo da bi vrijeme izvršavanja funkcije prebroji_1 trebalo **kvadratično** ovisiti o duljini stringa N . Graf kojim smo prikazali vrijeme izvršavanja zaista izgleda kao parabola. Analiza funkcije prebroji_2 gotovo je identična – jedina bitnija razlika jest ta što se vanjska petlja ne izvodi N puta, nego uvijek 26 puta:

$$T_{\text{prebroji_2}}(N) = 2 + 26 + 26N + 26 \cdot 2N + 26 + 26 = 78N + 80.$$

Za potrebe grafičkog prikaza funkcije prebroji_2, vremena iz tablice pomnožili smo s 50 kako bi se na grafu bolje vidjelo da ona rastu „po pravcu“. Funkcija prebroji_2 zapravo je 50 puta brža nego što to slika prikazuje!

najgori mogući slučaj

7. Složenost algoritama

Kvadratično vs. linearno	Ako vrijednost N udvostručimo, tada će se vrijeme izvođenja za prebroji_1 otprilike učetverostručiti, a vrijeme izvođenja za prebroji_2 samo udvostručiti. Na temelju te pretpostavke i vremena koja ste dobili za manje vrijednosti N na svojem računalu procijenite koliko će trajati izvođenje svake od funkcija za N=20000, a zatim pozovite obje funkcije za stringove te duljine i uvjerite se je li vaša procjena bila dobra!
$T_A(N)$	U prethodnim primjerima vidjeli smo kako možemo odrediti $T_A(N)$ – broj elementarnih operacija algoritma A – i na temelju tog broja procijeniti brzinu izvršavanja u ovisnosti o N – veličini ulaznih podataka. Pri tome bitnu kvalitativnu razliku i ključnu informaciju o brzini izvođenja algoritma daje samo dominantni član u izrazu $T_A(N)$. Tako za $T_A(N)=\text{konstanta}$ govorimo o algoritmima s konstantnom složenosti, za $T_A(N)=a \cdot N + b$ o algoritmima s linearom složenosti, a za $T_A(N)=a \cdot N^2 + b \cdot N + c$ o algoritmima s kvadratičnom složenosti.
konstantna složenost linearna složenost kvadratična složenost	Često se upotrebljava tzv. notacija veliko-O (engl. big-O notation). Stroga matematička definicija kaže sljedeće: algoritam A ima složenost $O(f(N))$ ako postoji neka konstanta c takva da za broj elementarnih operacija $T_A(N)$ potrebnih da algoritam izračuna odgovor za ulazne podatke veličine N vrijedi

$$T_A(N) \leq c \cdot f(N)$$

počevši od nekog N nadalje. Tako imamo:

- zbroj_2 složenosti je $O(1)$ jer je $T_{\text{zbroj_2}}(N) = 4 \leq c \cdot 1$ za $c=4$, za sve $N \geq 1$
- zbroj_1 složenosti je $O(N)$ jer je $T_{\text{zbroj_1}}(N) = 1 + 2N \leq c \cdot N$ za $c=3$, za sve $N \leq 1$
- prebroji_2 složenosti je $O(N)$ jer je $T_{\text{prebroji_2}}(N) = 78N + 80 \leq c \cdot N$ za $c=79$, za sve $N \geq 2$
- prebroji_1 složenosti je $O(N^2)$ jer je $T_{\text{prebroji_1}}(N) = 3N^2 + 3N + 1 \leq c \cdot N^2$ za $c=4$, za sve $N \geq 4$.

Vidimo da kod određivanja složenosti u izrazu $T_A(N)$ uvijek možemo zanemariti sve članove osim onog koji ima najveću potenciju od N. Općenito:

- ako je $T_A(N)=\text{konstanta}$, onda algoritam ima složenost $O(1)$
- ako je $T_A(N)=a \cdot N + b$, onda algoritam ima složenost $O(N)$
- ako je $T_A(N)=a \cdot N^2 + b \cdot N + c$, onda algoritam ima složenost $O(N^2)$.

Neke često korištene funkcije složenosti i njihova imena prikazani su u donjoj tablici. Funkcije su porедane od najmanje prema većoj složenosti, pa je uvijek bolje pokušati napraviti algoritam čija je složenost pri vrhu tablice.

$O(1)$	konstantna složenost
$O(\log(N))$	logaritamska složenost
$O(N)$	linearna složenost
$O(N \cdot \log(N))$	log-linearna složenost
$O(N^2)$	kvadratična složenost
$O(N^3)$	kubična složenost
$O(2^N)$	eksponencijalna složenost

7. Složenost algoritama

Općenito, za bilo koji prirodni broj p složenost $O(N^p)$ zovemo polinomijalna, a složenost koja je $O(p^N)$ ili gora, na primjer $O(N \cdot 2^N)$ ili $O(N!)$, zovemo eksponencijalna. Algoritmi koji imaju eksponencijalnu složenost izrazito su spori u usporedbi s polinomijalnim – usporedite koliko brzo raste funkcija poput $f(N)=2^N$ u usporedbi s $f(N)=N^2$ ili čak $f(N)=N^5$ tako da nacrtate odgovarajuće grafove.

Još prethodne godine susreli smo se s jednim algoritmom koji ima logaritamsku složenost – riječ je o binarnom pretraživanju unutar (sortiranog) niza. Podsjetimo se kako je izgledala funkcija koja vraća indeks na kojem se element x nalazi u uzlazno sortiranoj listi L (odnosno -1 ako se x ne nalazi u L):

```
def binarno_pretrazivanje( L, x ):
    d = 0; g = len(L)-1; indeks = -1;
    while( d <= g ):
        s = (d+g) // 2;
        if( x == L[s] ):
            indeks = s;
            break;
        elif( x < L[s] ):
            g = s-1;
        elif( x > L[s] ):
            d = s+1;
    return indeks;
```

binarno pretraživanje

Ako lista L ima $N=2^k$ elemenata, tijelo unutarnje petlje izvest će se najviše k puta, dakle, najviše $\log_2 N$ puta. Razlog je taj što se interval $[d, g]$ unutar kojeg tražimo indeks elementa x u svakom koraku raspolovi. Za bilo koji N unutarnja petlja izvrši se manje od $\log_2 N + 1$ puta. Broj je elementarnih operacija unutar petlje konstantan, recimo, manji je od 20 (prebrojite ih točno za vježbu!). Stoga je ukupan broj operacija u funkciji `binarno_pretrazivanje` manji od $20 \cdot \log_2 N$ pa ta funkcija ima složenost $O(\log N)$.

Za kraj ove sekcije, upozorimo na važan, ali pomalo skriveni detalj u Pythonu. Naime, ugrađene Pythonove funkcije kojima se koristimo u radu s listama, stringovima, rječnicima imaju složenosti koje ne smijemo zanemariti prilikom pisanja učinkovitog koda.

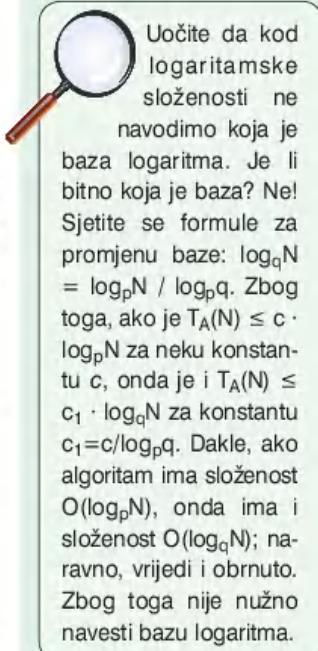
Primjer 3: Pythonova funkcija `insert`

Napravimo mali eksperiment: napišimo dvije funkcije koje trebaju stvoriti listu brojeva 1, 2, 3, ..., N . Prva će funkcija listu stvoriti tako da dodaje broj po broj na kraj liste, a druga će funkcija dodavati broj po broj na početak liste. Očekujete li da će funkcije imati isto ili različito vrijeme izvršavanja?

Funkcije se razlikuju samo u parametru koji šaljemo funkciji `L.insert`:

```
def dodaj_kraj( L, N ):
    for i in range( 1, N+1 ):
        L.insert( len(L), i );
```

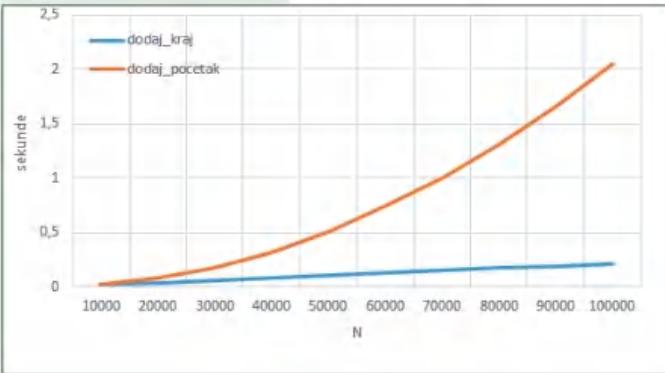
`L.insert()`



7. Složenost algoritama

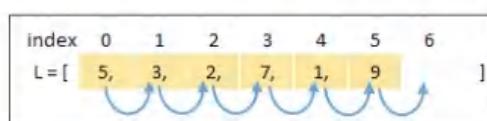
```
def dodaj_pocetak( L, N ):
    for i in range( 1, N+1 ):
        L.insert( 0, i );
```

Međutim, vrijeme izvršavanja drastično je različito:



N	dodaj_kraj	dodaj_pocetak
10000	0.0022	0.0235
20000	0.0041	0.0823
30000	0.0063	0.1798
40000	0.0083	0.3234
50000	0.0105	0.5108
60000	0.0127	0.7376
70000	0.0154	1.0070
80000	0.0173	1.3137
90000	0.0190	1.6650
100000	0.0214	2.0505

(Ponovno smo vremena za `dodaj_kraj` zbog jasnijeg prikaza na grafu pomnožili s 30.) U čemu je problem s dodavanjem na početak? Svaki broj u listi mora se pomaknuti za jedno mjesto „udesno“ kako bi napravio mesta i smjestio novi broj na indeks 0.



Ako želimo ubaciti novi element na početak liste s gornje slike, prvo se broj 9 treba pomaknuti s indeksa 5 na indeks 6; zatim se broj 1 mora pomaknuti s indeksa 4 na indeks 5 i tako dalje, sve do broja 5, koji se na kraju mora pomaknuti s indeksa 0 na indeks 1. Sva ta pomicanja imaju za posljedicu da poziv `L.insert(0, i)` ima složenost $O(k)$, gdje je k trenutna duljina liste L . Kako lista L raste, k se povećava od 1 do N . Ukupan broj elementarnih operacija potrebnih da bi se napravila lista duljine N možemo stoga procijeniti na

$$T_{\text{dodaj_pocetak}}(N) = 1 + 2 + \dots + N = \frac{1}{2} N(N+1) = \frac{1}{2} N^2 + \frac{1}{2} N,$$

složenost funkcije
`insert()`

pa je ukupna složenost ovog algoritma $O(N^2)$. No, ako novi broj uvijek dodajemo na kraj liste, svi preostali brojevi mogu ostati na svojim starim indeksima, pa poziv `L.insert(len(L), i)` ima složenost $O(1)$. Zbog toga funkcija `dodaj_kraj` ima složenost od samo $O(N)$.

Dakle, složenost funkcije `L.insert()` ne ovisi samo o duljini liste N nego i o mjestu na koje se novi element ubacuje. U najgorem slučaju, njezina će složenost biti $O(N)$. Vidimo da, ako je moguće, uvijek treba izbjegavati dodavanje (i brisanje) elemenata liste koji nisu na njezinu kraju. Slično, funkcije poput `count` ili `find` također imaju složenost $O(N)$,

7. Složenost algoritama

što je logično – potrebno je „protrčati“ kroz cijelu listu da bi se prebrojio ili pronašao zadani element.



Popis očekivanih složenosti za sve važnije funkcije u Pythonu možete pronaći na linku: <https://wiki.python.org/moin/TimeComplexity>

7.2. Algoritmi za sortiranje

Prehodne godine naučili smo sortirati zadatu listu L s pomoću insertion_sorta.

```
def insertion_sort( L ):
    for i in range(0, len(L)):
        for j in range(i+1, len(L)):
            if( L[i] > L[j] ):
                temp = L[i]; L[i] = L[j]; L[j] = temp;
```

insertion sort

Sada znamo odrediti složenost ovog algoritma. Neka lista L ima N elemenata. Kada je $i=0$, if-naredba će se izvršiti N puta; kada je $i=1$ izvršit će se $N-1$ puta; kada je $i=2$ izvršit će se $N-2$ puta i tako dalje. Unutar if-a imamo u najgorem slučaju četiri elementarne operacije: jedno uspoređivanje i tri kopiranja vrijednosti. Dakle, ukupan je broj operacija

$$T_{\text{insertion_sort}}(N) = 4(N + (N-1) + \dots + 2 + 1) = 2N(N+1) = 2N^2 + 2N,$$

pa insertion_sort ima složenost $O(N^2)$. Opisat ćemo dva algoritma koji imaju bolju složenost. Oba će algoritma biti rekurzivna.

Merge sort

Ponovno ćemo primijeniti logiku 1-2-3 iz poglavlja o rekurzijama da osmislimo funkciju merge_sort koja će uzlazno sortirati listu brojeva. Trivijalan je problem sortirati praznu listu ili listu koja ima samo $N=1$ element. Problem sortiranja liste s $N>1$ elemenata svest ćemo na dva problema sortiranja manjih lista (svaka će od njih imati $N/2$ ili $N/2+1$ elemenata). To radimo ovako: početnu listu L koju treba sortirati podijelimo na dvije polovice, L_1 i L_2 :

merge sort

```
L=[ 5, 3, 2, 7, 1, 9, 3, 7, 4, 8 ]
```

Zatim ćemo pozvati merge_sort dva puta, jednom da sortira L_1 , a drugi put da sortira L_2 . Naravno, po logici 1-2-3 možemo prepostaviti da će rekurzivni poziv uspješno sortirati obje liste. Tako će svaka polovica liste L postati sortirana:

```
L=[ 1, 2, 3, 5, 7, 3, 4, 7, 8, 9 ]
    p1           p2
```

Sada je potrebno spojiti (otuda dolazi naziv „merge-sort“) dvije već sortirane polovice u jednu veliku sortiranu listu S . To se može napraviti vrlo učinkovito: za svaku od polovica uvedimo indekse p_1 i p_2 koji će označavati na kojem smo elementu stali. Na početku su

7. Složenost algoritama

oba indeksa na prvim elementima svojih polovica. Usporedimo elemente na koje pokazuju indeksi pa manji od njih kopiramo na kraj liste S , a pripadni indeks povećamo za 1.

$L = [\quad 1, \quad 2, \quad 3, \quad 5, \quad 7, \quad 3, \quad 4, \quad 7, \quad 8, \quad 9 \quad]$	$S = [\quad 1 \quad]$
p_1	p_2

Taj postupak ponavljamo sve dok jedan od indeksa, p_1 ili p_2 , ne „izleti“ izvan svoje polovice liste. Tako nakon još jednog koraka imamo ovakvo stanje:

$L = [\quad 1, \quad 2, \quad 3, \quad 5, \quad 7, \quad 3, \quad 4, \quad 7, \quad 8, \quad 9 \quad]$	$S = [\quad 1, \quad 2 \quad]$
p_1	p_2

a nakon još jednog ovakvo (ako su elementi iznad p_1 i p_2 isti, svejedno je koji kopiramo u S):

$L = [\quad 1, \quad 2, \quad 3, \quad 5, \quad 7, \quad 3, \quad 4, \quad 7, \quad 8, \quad 9 \quad]$	$S = [\quad 1, \quad 2, \quad 3 \quad]$
p_1	p_2

Nastavljanjem tog postupka iz svoje polovice liste prvi će izletiti indeks p_1 :

$L = [\quad 1, \quad 2, \quad 3, \quad 5, \quad 7, \quad 3, \quad 4, \quad 7, \quad 8, \quad 9 \quad]$	$S = [\quad 1, \quad 2, \quad 3, \quad 3, \quad 4, \quad 5, \quad 7, \quad 7 \quad]$
p_1	p_2

Kada jedan od indeksa „izleti“ izvan svoje polovice liste, sve elemente koji su preostali u drugoj polovici redom dodajemo na kraj liste S . Napravljena na taj način lista S očito će biti ista kao sortirana lista L !

Sada možemo napisati kod za funkciju `merge_sort`. Radi što manje količine kopiranja dijelova liste L , nećemo zaista stvoriti nove liste L_1 i L_2 , nego ćemo ih reprezentirati indeksima: lista L_1 sastoji se od svih elemenata liste L na indeksima između st i mid (dakle, kao da je $L_1=L[st:mid]$), a lista L_2 sastoji se od elemenata liste L koji su na indeksima između mid i en (dakle, kao da je $L_2=L[mid:en]$). Takvim smo se trikom već koristili u poglavlju o rekurzijama.

```
def merge_sort( L, st, en ):
    # Trivijalni problem: lista L ima 0 ili 1 element.
    if( st+1 >= en ):
        return;

    # Netrivijalni problem: dijelimo listu L na dvije polovice.
    mid = (st+en) // 2;

    # Sortiramo prvu polovicu liste: od indeksa st do indeksa mid.
    merge_sort( L, st, mid );

    # Sortiramo drugu polovicu liste: od indeksa mid do indeksa en.
    merge_sort( L, mid, en );
```

7. Složenost algoritama

```
# Spajamo dvije sortirane polovice u novu sortiranu listu S.  
S = []; p1 = st; p2 = mid;  
while( p1 < mid and p2 < en ):  
    if( L[p1] < L[p2] ):  
        S.append( L[p1] );  
        p1 = p1 + 1;  
    else:  
        S.append( L[p2] );  
        p2 = p2 + 1;  
  
# Barem u jednoj od polovica došli smo do kraja.  
# Sve elemente iz preostale polovice dodajemo na kraj liste S.  
while( p1 < mid ):  
    S.append( L[p1] );  
    p1 = p1 + 1;  
  
while( p2 < en ):  
    S.append( L[p2] );  
    p2 = p2 + 1;  
  
L[st:en] = S;
```

Iz glavnog programa tu funkciju pozivamo ovako: `merge_sort(L, 0, len(L))`.

Kako odrediti složenost rekurzivne funkcije? Formula za ukupan broj operacija $T_{\text{merge}}(N)$ također će biti rekurzivna! Radi jednostavnosti, prepostavimo da je $N=2^k$, za neki prirodni broj k . Na početku (if i izračunavanje mid) imamo ukupno pet elementarnih operacija. Nakon toga pozivamo `merge_sort` za liste L_1 i L_2 , svaka od kojih ima $N/2=2^{k-1}$ elemenata. Broj operacija u svakom od tih poziva jest $T_{\text{merge}}(N/2)$. Nakon toga ide spajanje lista L_1 i L_2 . Svako od ukupno N dodavanja elemenata u listu S troši ukupno tri elementarne operacije (dodajemo uvijek na kraj s `append!`). Dakle, na stvaranje liste S trošimo $3N$ elementarnih operacija. Na kraju, možemo uzeti da kopiranje iz S u L ponovo troši N elementarnih operacija. Kada sve zbrojimo, dobivamo:

$$T(N) = 2T(N/2) + 4 \cdot N + 5, \text{ odnosno, } T(2^k) = 2T(2^{k-1}) + 4 \cdot 2^k + 5.$$

Sada ćemo umjesto $T(2^{k-1})$ uvrstiti $T(2^{k-1}) = 2T(2^{k-2}) + 4 \cdot 2^{k-1} + 5$, pa onda u dobiveno umjesto $T(2^{k-2})$ uvrstiti $T(2^{k-2}) = 2T(2^{k-3}) + 4 \cdot 2^{k-2} + 5$ i tako dalje, sve dok ne dođemo do $T(2^0) = T(1) = 2$. Račun daje (vidi desno):

$$\begin{aligned} T(N) &= 2T(2^{k-1}) + 4 \cdot 2^k + 5 \\ T(2^k) &= 2(2T(2^{k-2}) + 4 \cdot 2^{k-1} + 5) + 4 \cdot 2^k + 5 \\ &= 22 T(2^{k-2}) + 2 \cdot 4 \cdot 2^k + (1 + 21) \cdot 5 \\ &= 22(2T(2^{k-3}) + 4 \cdot 2^{k-2} + 5) + 2 \cdot 4 \cdot 2^k + (1 + 21) \cdot 5 \\ &= 23 T(2^{k-4}) + 3 \cdot 4 \cdot 2^k + (1 + 21 + 22) \cdot 5 \\ &= \dots \\ &= 2^{k-1} T(2^0) + (k-1) \cdot 4 \cdot 2^k + (1 + 21 + 22 + \dots + 2^{k-2}) \cdot 5 \\ &= 2k + (k-1) \cdot 4 \cdot 2^k + (2k-1-1) \cdot 5 \\ &= 4k \cdot 2^k - \frac{1}{2} \cdot 2k - 5 \\ &= 4N \cdot \log_2(N) - \frac{1}{2} N - 5 \end{aligned}$$

složenost merge
sort algoritma

7. Složenost algoritama



Uočite da je, na primjer, za $N=100\ 000$ broj N^2 jednak deset milijardi, a broj $N \log_2 N$ samo otrliki miliun i pol! Na računalu koje izvršava milijardu elementarnih operacija u sekundi zato očekujemo da insertion_sort treba (reda veličine) desetak sekundi za sortiranje niza od 100 000 brojeva, dok će ih merge_sort sortirati praktički trenutno.

Složenost algoritma merge_sort jest $O(N \log N)$ jer će član $4N \cdot \log_2(N)$ biti puno veći od $-1/2N - 5$ za velike N -ove. Može se pokazati da to vrijedi i u slučaju kad N nije potencija broja 2.

Očekujemo da će ovaj algoritam moći brzo sortirati puno veće liste nego insertion_sort.

Algoritam merge_sort ipak ima jedan nedostatak u odnosu na insertion sort. Naime, za merge_sort potrebna je dodatna pomoćna lista S , dok insertion_sort ne zahtijeva nikakvu pomoćnu memoriju. Sljedeći algoritam za sortiranje, quick_sort, neće zahtijevati nikakvu pomoćnu memoriju, a (u praksi) će imati istu složenost kao i merge_sort. Zbog toga se u primjeni najčešće i upotrebljava quick_sort, pa se tako i Pythonova ugrađena funkcija za sortiranje liste L , $L.sort()$, koristi upravo tim algoritmom.

Quick sort

quick sort

pivot

Ideja ovog algoritma ponovno je rekurzivna, ali malo drugačija nego kod merge_sorta. Ponovno, trivijalni su problemi sortiranje prazne i liste s jednim elementom, a liste duljine $N > 1$ sortiramo tako da problem svedemo na sortiranje dviju kraćih lista. Kod quick_sorta prvi element liste zovemo pivot.

pivot
$L = [5, 3, 2, 7, 1, 9, 3, 7, 4, 8]$

Zamislimo da nekako uspijemo premjestiti pivot na mjesto na kojem će se nalaziti u konačnoj, sortiranoj listi, da sve elemente koji su manji od pivota nekako preselimo lijevo od njega, a sve elemente koji su veći od pivota preselimo desno od njega. Za gornju listu to bi moglo izgledati ovako:

pivot
$L = [3, 3, 2, 4, 1, 5, 9, 7, 7, 8]$

Da sortiramo cijelu listu, sada je potrebno sortirati još samo lijevu listu L_1 i desnu listu L_2 . To ćemo napraviti rekurzivnim pozivom funkcije quick_sort.

Preostalo je još samo osmisiliti kako premjestiti pivot na konačno mjesto. To se učinkovito radi ovako:

1. Uvedemo dva indeksa, „lijevi“ / i „desni“ d .
2. Na početku je lijevi indeks na drugom elementu liste koju sortiramo, a desni na zadnjem elementu.
3. Lijevi indeks pomicat ćemo jedno po jedno mjesto udesno, a desni indeks jedno po jedno ulijevo.
4. Lijevo od / uvjek se moraju nalaziti elementi koji su manji od pivota, a desno od d elementi koji su veći od pivota.

7. Složenost algoritama

5. Ako nađemo na element na indeksu l koji je veći od pivota i na element na indeksu d koji je manji od pivota, pomicanje ćemo zaustaviti. Dva elementa na kojima su se našli indeksi l i d međusobno ćemo zamijeniti pa će ponovno vrijediti svojstvo iz točke 4. Tada nastavljamo pomicanje.
6. Kada se lijevi i desni indeks u svojem pomicanju mimođu, onda smo gotovi. Tada samo zamijenimo pivotni element s elementom na kojem se zaustavio desni indeks (to će biti neki element manji od pivota).

Ovo izgleda relativno komplikirano, ali će na primjeru biti jasno što se događa. Na početku je stanje ovakvo:

pivot	1
$L = [\boxed{5}, \quad 3, \quad 2, \quad 7, \quad 1, \quad 9, \quad 3, \quad 7, \quad 4, \quad 8]$	d

Sad se prvo indeks l pomiče udesno sve dok ne nađe na element veći od pivota. Zatim se indeks d pomiče ulijevo sve dok ne nađe na element manji od pivota. Kad oba indeksa stanu, situacija je sljedeća:

pivot	1
$L = [\boxed{5}, \quad 3, \quad 2, \quad \boxed{7}, \quad 1, \quad 9, \quad 3, \quad 7, \quad \boxed{4}, \quad 8]$	d

Sada zamijenimo elemente unutar plavih okvira:

pivot	1
$L = [\boxed{5}, \quad 3, \quad 2, \quad 4, \quad 1, \quad 9, \quad 3, \quad 7, \quad 7, \quad 8]$	d

Nastavljamo s pomicanjem indeksa l udesno i indeksa d ulijevo. Sljedeće je zaustavljanje ovdje:

pivot	1
$L = [\boxed{5}, \quad 3, \quad 2, \quad 4, \quad 1, \quad \boxed{9}, \quad \boxed{3}, \quad 7, \quad 7, \quad 8]$	d

Nakon zamjene elemenata u plavim okvirima imamo:

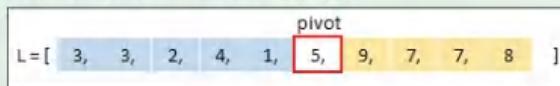
pivot	1
$L = [\boxed{5}, \quad 3, \quad 2, \quad 4, \quad 1, \quad 3, \quad 9, \quad 7, \quad 7, \quad 8]$	d

Ponovno pomičemo oba indeksa u odgovarajućim smjerovima. Međutim, nakon što se l zaustavi na elementu većem od pivota, a d na elementu manjem od pivota, vidimo da su se indeksi l i d mimoili, pa ne radimo zamjenu elemenata.

pivot	1
$L = [\boxed{5}, \quad 3, \quad 2, \quad 4, \quad 1, \quad 3, \quad 9, \quad 7, \quad 7, \quad 8]$	d

7. Složenost algoritama

Sada ne pomicemo više l i d , nego zamjenjujemo pivota elementom na indeksu d . Time smo dobili upravo željenu situaciju u kojoj su svi elementi manji od pivota lijevo od njega, a svi elementi veći od pivota desno od njega:



Kako smo i naveli gore, u ovoj ćemo situaciji rekurzivno pozvati quick_sort za plavu i za žutu polovicu liste. Cijeli taj postupak nalazi se u donjem kodu.

```
def quick_sort( L, st, en ):
    # Trivijalni problem: liste duljine 0 i 1.
    if( st+1 >= en ):
        return;

    # Pomiči indekse l i d tako da svi elementi lijevo od l budu manji od pivota,
    # a svi elementi desno od d budu veći od pivota.
    l = st+1; d = en-1; pivot = L[st];
    while( l <= d ):
        while( l < en and L[l] <= pivot ):
            l = l + 1;
        # Pomiči se udesno sve dok ne nađeš na element veći od pivota.
        while( d > st and L[d] >= pivot ):
            d = d - 1;
        # Pomiči se ulijevo sve dok ne nađeš na element manji od pivota.
        # Zamjeni elemente na kojima su se zaustavili indeksi l i d.
        if( l < d ):
            temp = L[l]; L[l] = L[d]; L[d] = temp;

    # Stavi pivot na njegov konačni indeks d u sortiranoj listi.
    if( st < d ):
        L[st] = L[d]; L[d] = pivot;

    quick_sort( L, st, d );
    # Sortiraj prvu "polovicu" niza: od indeksa st do indeksa d.
    quick_sort( L, d+1, en );
    # Sortiraj drugu "polovicu" niza: od indeksa d+1 do indeksa en.
```

složenost quick
sort algoritma

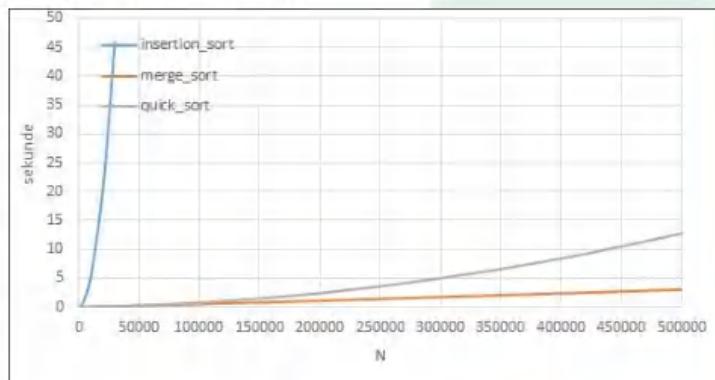
Analiza složenosti vrlo je slična kao kod merge_sorta. Međutim, bitna je razlika što ne znamo unaprijed kolika je duljina dviju lista za koje pozivamo rekurziju. Ako pretpostavimo da svaka od njih ponovno ima točno $N/2$ elemenata, dobit ćemo da je složenost quick_sort algoritma $O(N \cdot \log N)$ – provedite sami ovaj račun za vježbu, vrlo je slično kao za merge_sort. Međutim, može se dogoditi da jedna od lista ima samo jedan element, a druga $N-2$ elementa (na primjer, ako je lista L već sortirana). To je najgori mogući slučaj za ovaj algoritam i tada je složenost sortiranja $O(N^2)$. Srećom, takva „loša sreća“ u kojoj jedna lista uvijek ima jako malo elemenata, a druga jako puno vrlo se rijetko do-

7. Složenost algoritama

gađa. Kako su sva ostala svojstva quick_sorta izuzetno dobra i on ne zahtijeva dodatnu memoriju poput merge_sorta, u praksi se gotovo uvijek koristimo tim algoritmom.

Za kraj, usporedimo vremena izvođenja svih opisanih algoritama za sortiranje.

N	insertion_sort	merge_sort	quick_sort
1000	0.0735	0.0032	0.0022
2000	0.2617	0.0071	0.0050
5000	1.4894	0.0211	0.0160
10000	5.1980	0.0444	0.0351
20000	20.4863	0.0941	0.0810
30000	45.7676	0.1451	0.1332
50000		0.2498	0.2764
100000		0.5331	0.7531
200000		1.0989	2.4170
500000		3.0433	12.7652



Graf za insertion_sort jest parabola, dok su grafovi za merge_sort i quick_sort funkcije oblika $f(x) = a \cdot x \cdot \log(x)$ za dvije različite konstante a. U našoj je implementaciji quick_sort ispašao nešto sporiji.

složenost quick sort algoritma

Od problema do rješenja

U posljednjoj sekciji opisat ćemo tri različita rješenja istog zadatka. Svako će rješenje donijeti napredak u složenosti u odnosu na prethodno.

Zadatak 1: Najdulji rastući podniz

Napišite funkciju koja prima listu L u kojoj se nalazi niz prirodnih brojeva. Funkcija mora pronaći duljinu najduljeg rastućeg podniza (NRP) brojeva unutar zadane liste. Na primjer, ako je zadana lista L=[9, 3, 4, 2, 8, 5, 7], onda je najdulji podniz u kojem brojevi rastu [3, 4, 5, 7], pa funkcija mora vratiti njegovu duljinu – broj 4.

najdulji rastući podniz

Prvo rješenje. Prvo rješenje primjena je „grube sile“ (engl. *brute-force*). Rekurzivno ćemo generirati sve moguće podnizove zadane liste. Za svaki generirani podniz provjerit ćemo je li neopadajući i zapamtiti ćemo najdulji takav niz. Način generiranja svih podnizova liste L vrlo je sličan ispisivanju svih nizova binarnih znamenki iz poglavlja o rekurzijama. Sličnost između tog zadatka i donjeg koda bit će jasnija ako zamislimo da ispod svakog elementa liste napišemo ili broj 0 ili broj 1 i u podniz stavimo samo one elemente ispod kojih smo napisali broj 1. Očito, ako na sve moguće načine napišemo 0 ili 1, dobit ćemo sve moguće podnizove liste L.

metoda
"grube sile"

gdje	0	1	2	3	4	5	6
L = [9,	3,	4,	2,	8,	5,	7]
0	0	1	1	0	1	0	→ podniz = [4, 2, 5 1]

7. Složenost algoritama

```
def NRP_1( L, gdje=0, podniz=[ ] ):
    # Gdje je indeks elementa liste za koji odlučujemo hoćemo li ga dodati u listu ili ne.
    if( gdje == len(L) ):
        # Došli smo do kraja liste - provjeri je li dobiveni podniz rastući.
        rastuci = True;
        for i in range( 1, len(podniz) ):
            if( podniz[i-1] >= podniz[i] ):
                rastuci = False;
                break;

        if( rastuci ):
            return len(podniz);
        else:
            return -1;

    # 1. slučaj: nemoj dodati L[gdje] u podniz, već ga preskoči i prijeđi na L[gdje+1].
    najdulji = NRP_1( L, gdje+1, podniz );

    # 2. slučaj: dodaj L[gdje] na kraj podniza i onda prijeđi na L[gdje+1].
    najdulji = max( najdulji, NRP_1( L, gdje+1, podniz +
[L[gdje]] ) );

    return najdulji;
```

Složenost ovog algoritma samo ćemo procijeniti: ako lista L ima N elemenata, onda algoritam generira svih 2^N podnizova. Ako prepostavimo da mu za generiranje svakog podniza treba samo jedna jedina elementarna operacija, njegova složenost, i to bez provjere je li podniz rastući, bit će minimalno $O(2^N)$. Dakle, rješenje „grubom silom“ rezultiralo je algoritmom eksponencijalne složenosti. Uvjerite se da ovaj algoritam radi izuzetno sporo već za liste malih duljina, npr. za $N=25$.

Drugo rješenje. Pokušajmo sada osmisliti posve drugačiji algoritam koji neće zahtijevati generiranje svih podnizova. Za svaki element liste L postavimo pitanje: „Koliko je dug NRP od L kojem je upravo ovaj element zadnji?“. Označimo s $\text{MAX}[i]$ duljinu NRP-a koji završava elementom $L[i]$. Pogledajmo donju sliku:

zadnji	0	1	2	3	4	5	6	
$L = [$	9,	3,	4,	2,	8,	5,	7,	1]
$\text{MAX} = [$	1,	1,	2,	1,	3,	3,]

Na slici smo već izračunali $\text{MAX}[0]$, $\text{MAX}[1]$, ..., $\text{MAX}[5]$. Na primjer, $\text{MAX}[4]=3$ jer je podniz [3, 4, 8] NRP koji završava brojem 8. Želimo odrediti $\text{MAX}[6]$, odnosno, duljinu

7. Složenost algoritama

najduljeg rastućeg podniza koji završava brojem 7. Što sve može biti prethodni broj u tom podnizu? Pogledajmo sve moguće slučajeve:

- Prije broja 7 ne dolazi niti jedan broj, pa tada taj podniz izgleda ovako: [7] i ima dužinu 1.
- Prije broja 7 ne smije doći $L[0]=9$.
- Prije broja 7 smije doći $L[1]=3$, pa tada taj podniz izgleda ovako: [..., 3, 7] i ima dužinu $\text{MAX}[1]+1 = 2$.
- Prije broja 7 smije doći $L[2]=4$, pa tada taj podniz izgleda ovako: [..., 4, 7] i ima dužinu $\text{MAX}[2]+1 = 3$.
- Prije broja 7 smije doći $L[3]=2$, pa tada taj podniz izgleda ovako: [..., 2, 7] i ima dužinu $\text{MAX}[3]+1 = 2$.
- Prije broja 7 ne smije doći $L[4]=8$.
- Prije broja 7 smije doći $L[5]=5$, pa tada taj podniz izgleda ovako: [..., 4, 7] i ima dužinu $\text{MAX}[5]+1 = 4$.

Od svih tih opcija, najdulji podniz dobivamo u zadnjoj, pa zato stavljamo $\text{MAX}[6] = 4$. Sada je jasno kako ćemo odrediti NRP za cijelu listu: redom ćemo izračunavati $\text{MAX}[0]$, pa $\text{MAX}[1]$ i tako dalje, sve dok ne dođemo do zadnjeg elementa liste. NRP za cijelu listu onda je najveći od svih brojeva u listi MAX.

```
def NRP_2( L ):
    najdulji = 0; MAX = [0]*len(L);

    # Koliko je dug najdulji rast. podniz koji završava na indeksu zadnji?
    for zadnji in range( 0, len(L) ):
        # Možda se sastoji samo od jednog elementa: L[zadnji]?
        MAX[zadnji] = 1;

        # Ili možemo nastaviti bilo koji podniz koji završava na L[predzadnji],
        # ako je L[predzadnji]<L[zadnji].
        for predzadnji in range( 0, zadnji ):
            if( L[predzadnji] < L[zadnji] and
                MAX[predzadnji]+1 > MAX[zadnji] ):
                MAX[zadnji] = MAX[predzadnji] + 1;

            if( najdulji < MAX[zadnji] ):
                najdulji = MAX[zadnji];

    return najdulji;
```

Kolika je složenost ovog algoritma? Za vježbu, prebrojimo precizno elementarne operacije. Prije vanjske petlje možemo uzeti da se izvrši $N+1$ elementarna operacija (uočite, inicijalizacija liste MAX sadržava skrivenu petlju koja obilazi sve elemente!). U vanjskoj petlji, koja se izvrši N puta, imamo svaki put po jedno kopiranje broja 1, jednu usporedbu

7. Složenost algoritama

i jedno kopiranje u varijablu najdulji – dakle, ukupno $3N$ elementarnih operacija. U unutarnjoj petlji svaki put imamo po pet operacija; one se izvrše $0 + 1 + \dots + (N-1)$ puta – dakle, sveukupno $5/2 N(N-1)$ operacija. Kada sve zbrojimo, ukupan je broj operacija

$$T_{NRP_2}(N) = N+1 + 3N + 5/2 N(N-1) = 5/2 N^2 + 3/2 N + 1,$$

pa je ovaj algoritam složenosti $O(N^2)$. Vidimo da za određivanje složenosti zapravo nije nužno precizno brojati operaciju po operaciju, nego samo pažljivo odrediti *red veličine* broja operacija. Posve je svejedno nalazi li se u unutarnjoj petlji 5, 50 ili 500 elementarnih operacija, **sve dok god taj broj ne ovisi o N.**

Tim algoritmom postigli smo bitan napredak u odnosu na NRP_1, pa sada možemo računati najdulje rastuće podnizove za liste koje imaju po nekoliko tisuća elemenata. Da napravimo još jedan korak u smanjivanju složenosti, bit će ponovno potrebna posve nova ideja.

Treće rješenje. Ovo rješenje ponajprije je namijenjeno naprednjim učenicima te ilustri ra neke ideje i tehnike koje se upotrebljavaju na višim stupnjevima srednjoškolskih natjecanja u programiranju.

I ovdje ćemo uvesti pomoćno polje, INDEX, čija je definicija nešto komplikiranjia: Jedan korak algoritma sastoji se od obradivanja jednog elementa liste. U prvom koraku obradujemo element s indeksom 0, u idućem koraku onaj s indeksom 1 i tako redom.

U svakom koraku algoritma INDEX[duljina] jest indeks na kojem se nalazi najmanji element do sada obrađenog dijela liste L na kojem završava neki rastući podniz duljine duljina. Ako ne postoji podniz te duljine, onda je INDEX[duljina] = ∞ .

Pogledajmo kako to izgleda na primjeru:

	0	1	2	3	4	5	6	
$L = [$	9,	3,	4,	2,	8,	5,	7]	
				i				
	duljina	1	2	3	4	5	6	7
INDEX[duljina]	3	2	∞	∞	∞	∞	∞	∞
$L[INDEX[duljina]]$	2	4						

U ovom trenutku algoritma obradili smo elemente 9, 3, 4, 2 i na temelju njih sagradili polje INDEX. Vrijedi:

- $INDEX[1] = 3$, jer se na indeksu 3 u listi L nalazi najmanji element (to je broj 2) takav da postoji podniz duljine 1 koji završava tim elementom. To je podniz [2].
- $INDEX[2] = 2$, jer se na indeksu 2 u listi L nalazi najmanji element (to je broj 4) takav da postoji podniz duljine 2 koji završava tim elementom. To je podniz [3, 4].
- $INDEX[3] = \infty$, jer među dosad obrađenim elementima ne postoji rastući podniz duljine 3.

Sada želimo obraditi idući element, broj 8. Pogledajmo što će se promijeniti u polju INDEX.

	0	1	2	3	4	5	6	
$L = [$	9,	3,	4,	2,	8,	5,	7]	
				i				
	duljina	1	2	3	4	5	6	7
INDEX[duljina]	3	2	4	∞	∞	∞	∞	∞
$L[INDEX[duljina]]$	2	4	8					

- I dalje vrijedi $INDEX[1] = 3$, jer se na indeksu 3 nalazi najmanji element (to je broj 2) takav da postoji podniz duljine 1 koji završava tim elementom. To je podniz [2].

7. Složenost algoritama

- I dalje vrijedi $\text{INDEX}[2] = 2$, jer se na indeksu 2 nalazi najmanji element (to je broj 4) takav da postoji podniz duljine 2 koji završava tim elementom. To je podniz [3, 4].

Međutim, sada je $\text{INDEX}[3] = 4$, jer se na indeksu 4 nalazi najmanji element (to je broj 8) takav da postoji podniz duljine 3 koji završava tim elementom. To je podniz [3, 4, 8]. Kako smo dobili taj podniz? Tako da smo uzeли najdulji mogući podniz koji završava brojem manjim od 8 i na njega nadodali taj broj. Dakle, išli smo redom kroz polje INDEX i promatrati brojeve $L[\text{INDEX}[x]]$. Pronašli smo zadnji broj x za koji je $L[\text{INDEX}[x]]$ manji od broja 8. To znači da smo pronašli najdulji podniz koji završava brojem manjim od 8 (duljina tog podniza upravo je x).

U idućem koraku obrađujemo broj 5. Nakon njegova dodavanja imamo sljedeću situaciju:

0	1	2	3	4	5	6
9,	3,	4,	2,	8,	5,	7

|

duljina	1	2	3	4	5	6	7
INDEX[duljina]	3	2	5	∞	∞	∞	∞
$L[\text{INDEX}[duljina]]$	2	4	5				

Ponovno, $\text{INDEX}[1]$ i $\text{INDEX}[2]$ nisu se promijenili. Međutim, NRP duljine 3 sada može završavati brojem 5, umjesto brojem 8. Otkrili smo ga ponovno na isti način: pronade- mo najdulji podniz koji završava brojem manjim od 5 (to je podniz duljine 2) i na taj pod- niz nadodamo broj 5 – to je [3, 4, 5].

Konačno, obrađujemo broj 7. Promatrajući brojeve $L[\text{INDEX}[duljina]]$, vidimo da je NRP koji završava brojem manjim od 7 podniz duljine 3; on završava brojem 5. Zato ga proširimo brojem 7 i dobivamo podniz duljine 4 koji završava brojem 7, a broj 7 na indeksu je 6 u listi L – stavljamo $\text{INDEX}[4]=6$. Dakle, na kraju smo dobili:

0	1	2	3	4	5	6
9,	3,	4,	2,	8,	5,	7

|

duljina	1	2	3	4	5	6	7
INDEX[duljina]	3	2	5	6	∞	∞	∞
$L[\text{INDEX}[duljina]]$	2	4	5	7			

Najdulji rastući podniz koji smo pronašli ima duljinu 4.

Iako algoritam ima relativno komplikiranu ideju, sam kod vrlo je kratak:

```
def NRP_3( L ):  
    # Na početku sve elemente od INDEX postavimo na "beskonačno" (tj. na len(L)).  
    INDEX = [len(L)]*(len(L)+1); INDEX[1] = 0; najdulji = 1;  
    for i in range( 1, len(L) ):  
        # Pronadi NRP koji završava brojem manjim od  $L[i]$ ,  
        # tj. prvi indeks x tako da je  $L[\text{INDEX}[x]] \geq L[i]$ .  
        x = 1;  
        while( x <= najdulji ):  
            if( L[INDEX[x]] >= L[i] ):  
                break;  
            x = x + 1;
```

7. Složenost algoritama

```
# Dakle, L[INDEX[x-1]] < L[i].  
# Podniz koji završava na L[INDEX[x-1]] možemo proširiti brojem L[i].  
# Tako dobijemo novi NRP duljine x. On završava brojem L[i] koji je na indeksu i.  
INDEX[x] = i;  
if( x > najdulji ):  
    najdulji = x;  
  
return najdulji;
```

Međutim, ovaj algoritam i dalje može imati kvadratnu složenost! Tijelo vanjske petlje izvrši se N puta, a tijelo unutarnje petlje izvrši se onoliko puta kolika je vrijednost varijable `najdulji`, odnosno, duljina NRP-a u svakom od koraka algoritma. Ako je lista L sama rastuća, vrijedit će `najdulji=i`, pa će ukupna složenost i dalje biti $O(N^2)$.

Da riješimo taj problem, potrebno je uočiti samo još jedan mali detalj: vrijedi

$$L[INDEX[1]] \leq L[INDEX[2]] \leq L[INDEX[3]] \leq \dots$$

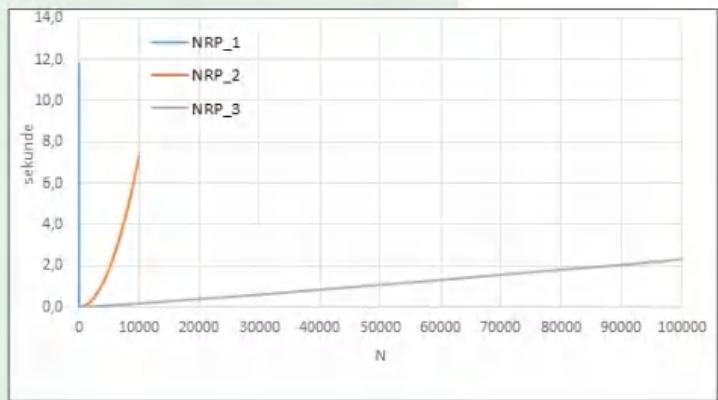
Primjerice, ako pronađemo podniz duljine 3 koji završava na broj 100, onda smo automatski pronašli i podnizove duljina 2 i 1 koji završavaju na broj 100. U listi možda postoje i manji brojevi od 100 na koje završavaju rastući podnizovi duljina 2 i 1, pa zato vrijede gornje nejednakosti. Zbog te činjenice možemo iskoristiti binarno pretraživanje umjesto unutarnje while petlje!

```
def NRP_3( L ):  
    # Na početku sve elemente od INDEX postavimo na "beskonačno" (tj. na len(L)).  
    INDEX = [-1]*(len(L)+1); INDEX[1] = 0; najdulji = 1;  
    for i in range( 1, len(L) ):  
        # Binarno pretraživanje u polju INDEX:  
        # tražimo najveći indeks x tako da je L[INDEX[x]] < L[i].  
        d = 1; g = najdulji;  
  
        while( d <= g ):  
            s = (d+g) // 2;  
            if( L[INDEX[s]] < L[i] ):  
                d = s+1;  
            else:  
                g = s-1;  
  
    # Kad se petlja zaustavi, upravo je d-1 indeks s gornjim svojstvom.  
    INDEX[d] = i;  
    if( d > najdulji ):  
        najdulji = d;  
  
return najdulji;
```

Sada se tijelo unutarnje petlje izvrši najviše $\log_2(\text{najdulji})$ puta. Kako je najdulji $\leq N$, ukupna složenost cijelog algoritma jest $O(N \log N)$.

Prikažimo na kraju tablično i grafom razliku u vremenima između algoritama koje smo opisali.

N	NRP_1	NRP_2	NRP_3
8	0.0003	0.0001	0.0001
20	1.4649	0.0004	0.0002
23	11.8227	0.0006	0.0002
1000		0.0723	0.0014
2000		0.2932	0.0029
5000		1.8305	0.0083
10000		7.3305	0.0179
50000			0.1091
500000			1.2978
1000000			2.6730



Vremena za NRP_3 za potrebe prikaza na grafu pomnožili smo s 10.

Zadaci za vježbu

1. a) Odredite složenost funkcije a u ovisnosti o duljini N liste X .

```
def a( X ):
    suma = 0;
    for i in range(0, len(X)):
        for j in range(0, len(X)):
            suma = suma + X[i]*X[j];
    return suma;
```

b) Odredite složenost funkcije b u ovisnosti o veličini broja N :

```
def b( N ):
    suma = 0;
    for i in range( 1, N ):
        suma = suma + i*(N-i+1);

    return suma;
```

7. Složenost algoritama

c) Odredite složenost funkcije c ako liste X i Y obje imaju po N elemenata:

```
def c( X, Y ):  
    suma = 0;  
    for i in range( 0, len(X) ):  
        suma = suma + X[i];  
  
    for i in range( 0, len(Y) ):  
        suma = suma + Y[i];  
  
    return suma;
```

d) Odredite složenost funkcije d u ovisnosti o veličini broja N :

```
def d( N ):  
    suma = 0;  
    for i in range( 1, N ):  
        for j in range( 1, i ):  
            for k in range( 1, j ):  
                suma = suma + i*j*k;  
  
    return suma;
```

2. a) Odredite složenost rekurzivne funkcije e u ovisnosti o veličini broja N :

```
def e(N):  
    if( N == 1 ):  
        return 1;  
    else:  
        return N*e(N-1);
```

b) Odredite složenost rekurzivne funkcije f u ovisnosti o veličini broja N :

```
def f(N):  
    if( N <= 1 ):  
        return 1;  
    else:  
        return N + f(N // 2);
```

c) Odredite složenost rekurzivne funkcije g ako je N broj elemenata liste L , a funkciju pozivamo s $g(L)$:

```
def g( L, i=0 ):  
    if( i == len(L) ):  
        return 0;  
    else:  
        return max( L[i]+g(L, i+1), L[i]-g(L, i+1) );
```

7. Složenost algoritama

- d) Odredite složenost rekurzivne funkcije h ako je N broj elemenata liste L , a funkciju pozivamo s $h(L)$:

```
def h( L, i=0 ):
    if( i == len(L) ):
        return 0;
    else:
        return max( L[i]+g(L, i+1), 23 );
```

3. Osnovna je ideja algoritma NRP_2 rekurzivna. Zapišite tu ideju i pripadnu funkciju u Pythonu korištenjem logike 1-2-3. Koja je složenost algoritma bez korištenja tehnike memoizacije, a koja uz korištenje te tehnike?
4. Napišite funkciju koja prima dvije liste prirodnih brojeva X i Y koje obje imaju po N elemenata. Funkcija treba odrediti duljinu najduljeg zajedničkog podniza tih dviju lista. Na primjer, ako je $X=[2, 6, 1, 9, 4, 8, 5, 4, 9]$, $Y=[6, 7, 9, 1, 2, 8, 3, 5, 1]$, onda najdulji zajednički podniz od X i Y ima duljinu 4: to je podniz [6, 9, 8, 5]. Trebate napisati dva rješenja različitih složenosti:
 - a) Napišite funkciju tako da generira sve moguće podnizove lista X i Y i uspoređuje jesu li jednaki. Uvjerite se da ta funkcija ima eksponencijalnu složenost, odnosno barem $O(2^N)$.
 - b) Napišite funkciju tako da izračunava vrijednosti u kvadratnoj tablici MAX na ovaj način: $MAX[a][b]$ je duljina najduljeg zajedničkog podniza lista $X[0:a]$ i $Y[0:b]$. Tada je $MAX[len(X)][len(Y)]$ broj koji se traži u zadatku. Izračunavanje tablice organizirajte redak po redak – tako će vam u svakom trenutku već biti izračunate sve vrijednosti potrebne za određivanje $MAX[a][b]$. Odredite složenost napisane funkcije i uvjerite se da je ona polinomijalna.

8. Zapis

Većina do sada upoznatih Pythonovih elemenata dava nam je veliku slobodu i omogućava visok stupanj kreativnosti. Ipak, boljim poznavaočima drugih programskih jezika poput Pascala i C-a sigurno je zapela za oko činjenica da u Pythonu ne postoji tip podataka zapis. Njega u Pascalu definiramo ključnom riječi *record*, a u C-u ključnom riječi *struct*. Činjenica da nema tog tipa nimalo ne smanjuje mogućnosti Pythona. Sve što mogu Pascal i C s tim posebnim tipom podataka može i Python ako se upotrebljavaju postojeće podatkovne zbirke lista i rječnik.

8.1. Zapis – osnovna ideja

Zapis

Što je ustvari zapis ili struktura? Često moramo raditi s podacima koji se sastoje od komponenti različitog tipa. Pojedine od tih komponenti mogu imati svoje komponente. Na primjer, želimo na pregleđan način zapisati osobne podatke učenika jednog razreda. Prvo trebamo zapisati podatke o jednom učeniku. Učenik može biti opisan s više komponenti počevši od imena, prezimena, visine do, recimo, općeg uspjeha. Dalje, datum rođenja može opet biti opisan s pomoću dana, mjeseca i godine. Taj problem u Pythonu sigurno možemo rješiti koristeći se listama i/ili rječnikom. Probajmo to rješiti na jedan drugačiji način.

8.2. Zapis u Pythonu

Sjećate se kada smo u drugom razredu spominjali klase? Prisjetimo se.

Python je objektno orijentirani programski jezik. Ideja takva pristupa pisanju programa jest definiranje složenih tipova podataka (u tome ih kontekstu zovemo klase) na način da se u jednoj cjelini obuhvate svi podaci koji opisuju taj složeni tip i sve operacije (metode) koje nad tim složenim tipom možemo izvršavati. Pojedine varijable takva složenog tipa još zovemo objekti.

Čitajući taj podsjetnik kao da smo čitali odgovor na razmišljanje iz prošlog poglavlja. Zapis u Pythonu realizirat ćemo koristeći se svojstvima klasa. Pri tome nećemo objašnjavati teoriju koja stoji iza toga, već samo upotrijebiti svojstva klase koja su potrebna za rješavanje zadatka sa zapisom.

Ideja zapisa jest kreirati jednu opću klasu koja će u sebi imati sve podatke koji opisuju naš složeni tip. Nakon toga ćemo, koristeći se tom klasom, kreirati jedan konkretni objekt s konkretnim podacima i dalje raditi s njim.

Opći je oblik deklaracije zapisa:

```
class ime_zapisa:  
    komponenta_1 = inicijalna_vrijednost  
    komponenta_2 = inicijalna_vrijednost
```

Opisanu ideju razradio je kolega Mihael Bobičanec koji nam je dozvolio da je predstavimo u ovom udžbeniku.

8. Zapis

komponenta_3 = inicijalna_vrijednost

...

komponenta_N = inicijalna_vrijednost

Primjer 1.

Osmisli i definiraj zapis *Ucenik* koji će omogućiti zapisivanje podataka i rad s podacima koji opisuju učenika koji je zadan svojim imenom, prezimenom, razredom i prosjekom.

Rješenje:

Zapis/klasa zvat će se *Ucenik* i u sebi će imati četiri komponente: ime, prezime, razred i prosjek. Ona će opisivati općenitog učenika koji je opisan zadanim komponentama. Ime, prezime i razred stringovi su i njih čemo inicijalizirati na prazan string. Prosjek je realni broj koji čemo inicijalizirati na nulu.

Klasa/objekt

```
class Ucenik:  
    ime = ''  
    prezime = ''  
    razred = ''  
    prosjek = 0
```

Kako možemo kreirati jednog konkretnog učenika (*objekt ucenik iz klase Ucenik*) i pridružiti mu vrijednosti koje opisuju baš njega? Jednostavno! Konkretnе vrijednosti pridružujemo komponentama konkretnog učenika navođenjem naziva i komponente.

```
ucenik = Ucenik()  
ucenik.ime = input('Ime: ')  
ucenik.prezime = input('Prezime: ')  
ucenik.razred = input('Razred: ')  
ucenik.prosjek = float(input('Prosjek: '))
```

Ako komponentama konkretnog učenika ne pridružimo nove vrijednosti, one će imati vrijednosti na koje su inicijalizirane komponente u definiciji zapisa (klase). Ispis vrijednosti komponenti isto je jednostavan

```
print(ucenik.ime, ucenik.prezime, ucenik.razred, ucenik.prosjek)
```

U početku smo rekli kako želimo zapisati podatke za sve učenike nekog razreda. U situaciji kada imamo jednog učenika, lako možemo proširiti priču na N učenika. Za to čemo se koristiti listom u koju čemo zapisati N objekata iz zapisa *Ucenik*.

```
N = int(input('Broj učenika u razredu: '))  
razred = []  
for i in range(N):  
    ucenik = Ucenik()  
    ucenik.ime = input('Ime: ')  
    ucenik.prezime = input('Prezime: ')
```

(nastavak na idućoj stranici)

8. Zapis

```
ucenik.razred = input('Razred: ')
ucenik.prosjek = float(input('Prosjek: '))
razred = razred + [ucenik]
```

Kako u ovom poglavlju nećemo ulaziti u detaljna pojašnjenja klasa i objekata, postoji mogućnost da će tijekom rada sa zapisima dolaziti u situacije koje nećete moći objasniti. Opišimo jednu takvu situaciju.

Primjer 2.

Zapis (klasa) *Ucenik* ima sljedeće komponente: ime, broj izostanaka i listu svih ocjena. Kreiraj dva konkretna učenika te za svakog od njih omogući upis konkretnih vrijednosti za prvi mjesec nastave (rujan).

Rješenje:

Prvo ćemo definirati zapis (klasu) *Ucenik* s komponentama iz teksta zadatka: string *ime*, prirodan broj *izostanci* te lista *ocjene*.

```
class Ucenik:
    ime = ''
    izostanci = 0
    ocjene = []
```

Kreirajmo dva konkretna učenika, *x* i *y* te u komponentu *ime* učitajmo vrijednosti s ulaza.

```
x = Ucenik()
x.ime = input('Ime učenika je: ')
y = Ucenik()
y.ime = input('Ime učenika je: ')
```

Ostale komponente nismo posebno inicijalizirali ili u njih nešto učitavali s ulaza te one imaju inicijalne vrijednosti definirane u zapisu *Ucenik*.

```
print(x.ime, x.izostanci, x.ocjene)
print(y.ime, y.izostanci, y.ocjene)
>>>
Ime učenika je: Spužva
Ime učenika je: Patrik
Spužva 0 []
Patrik 0 []
```

Sada ćemo osvježiti podatke u komponentama *izostanci* i *ocjene*.

```
x.izostanci += 1      # Spužva je u rujnu izostao 1 sat
y.izostanci += 12     # Patrik je u rujnu izostao 12 sati
x.ocjene.append(5)    # Spužva je u rujnu dobio jednu peticu
y.ocjene.append(3)    # Patrik je u rujnu dobio jednu trojku
```

Ispišimo trenutno stanje u varijablama (*objektima*) *x* i *y*.

```
print(x.ime, x.izostanci, x.ocjene)
print(y.ime, y.izostanci, y.ocjene)
```

(nastavak na idućoj stranici)

8. Zapis

```
>>>  
Spužva 1 [5, 3]  
Patrik 12 [5, 3]
```

Broj izostanaka je očekivan ali ispis ocjena ne odgovara onom što smo očekivali. Što se je dogodilo?

Znamo da je u Pythonu varijabla brojčanog tipa nepromjenljiva i da se svakom promjenom njene vrijednosti mijenja i memorijска lokacija na kojoj je zapisana. S druge strane, lista je promjenjivi tip podataka i prilikom promjene vrijednosti ne mijenja se memorijска lokacija na kojoj je zapisana.

Kako nakon kreiranja konkretnih varijabli (objekata) *x* i *y* nismo lokalno inicijalizirali listu *x.ocjene* i *y.ocjene* promjenom njihovih vrijednosti došlo je i do promjene inicijalnih vrijednosti u zapisu (klasi) *Ucenik*. Da bi izbjegli ovakve situacije, nužno je lokalno inicijalizirati komponentu *ocjene*.

```
x = Ucenik()  
x.ime = input('Ime učenika je: ')  
x.ocjene = []  
  
y = Ucenik()  
y.ime = input('Ime učenika je: ')  
y.ocjene = []  
  
x.izostanci += 1  
y.izostanci += 2  
x.ocjene.append(5)  
y.ocjene.append(3)  
  
print(x.ime, x.izostanci, x.ocjene)  
print(y.ime, y.izostanci, y.ocjene)  
>>>  
Ime učenika je: Spužva  
Ime učenika je: Patrik  
Spužva 1 [5]  
Patrik 2 [3]
```

Podsjetite se:

```
>>> a = 5  
>>> s = 'Python'  
>>> L = [1,6]  
>>> id(a), id(s),  
id(L)  
(1917501520,  
35395808, 44499912)  
>>> a = a + 1  
>>> s = s + ' 3.4'  
>>> L.append(6)  
>>> id(a), id(s),  
id(L)  
(1917501536,  
47424400, 44499912)
```

8. Zapis

Od problema do rješenja

Kroz rješavanje dvaju konkretnih problema pokazat ćemo kako u praksi možemo upotrijebiti ideju zapisa, klase i objekta.

Problem 1.

Napiši program koji upisuje prirodni broj N i nakon toga podatke za N učenika oblika (ime, prezime, razred, prosjek). Učenike zatim treba upisati u razred. Program treba ispisati sve podatke za one učenike koji imaju prosjek veći ili jednak prosjeku koji će se dodatno zadati. Učenici trebaju biti ispisani sortirani po prosjeku od manjeg prema većem.

Rješenje:

Prvo ćemo definirati zapis (*klasu*) *Ucenik* s komponentama iz teksta zadatka.

```
class Ucenik:  
    ime = ''  
    prezime = ''  
    razred = ''  
    prosjek = 0
```

Zatim ćemo N puta kreirati konkretnog učenika (*objekt*) *ucenik* i u svakog od njih s ulaza ćemo učitati konkretnе vrijednosti. Nakon učitavanja cijeli ćemo objekt dodati u listu razred.

```
N = int(input('Broj učenika: '))  
razred = []  
for i in range(N):  
    ucenik = Ucenik()  
    ucenik.ime = input('Ime: ')  
    ucenik.prezime = input('Prezime: ')  
    ucenik.razred = input('Razred :')  
    ucenik.prosjek = float(input('Prosjek: '))  
    razred = razred + [ucenik]
```

Sada ćemo po prosjeku sortirati sve učenike iz razreda.

```
for i in range(N - 1):  
    for j in range(i + 1, N):  
        if razred[i].prosjek > razred[j].prosjek:  
            pom = razred[i]  
            razred[i] = razred[j]  
            razred[j] = pom
```

8. Zapis

Proći ćemo po svim članovima liste *razred* i ispisati podatke o onim učenicima koji imaju prosjek veći od zadanog.

```
koliko = float(input('Tražimo sve veće od prosjeka: '))
print('ime      prezime      prosjek')
print('-----')
for i in range(N):
    if razred[i].prosjek > koliko:
        print('{:10s}{:12s}{:6.2f}'.format(uc[i].ime,
uc[i].prezime, uc[i].prosjek))
```

Problem 2.

Napiši program koji za zapise oblika (ime, prezime, razred, prosjek) kreira izbornik oblika:

- 1) upis
- 2) ispis po abecedi
- 3) ispis po prosjeku
- 4) traženje pojedinca
- 5) brisanje pojedinca

i na temelju korisnikova izbora izvršava zadane funkcije.

Rješenje:

```
class Ucenik:
    ime = ''
    prez = ''
    raz = ''
    pros = 0

uc = []

def upis(uc):
    N = int(input('Broj učenika: '))
    for i in range(N):
        print('\nNovi učenik:')
        ucenik = Ucenik()
        ucenik.ime = input('Ime: ')
        ucenik.prez = input('Prezime: ')
        ucenik.raz = input('Razred: ')
        ucenik.pros = float(input('Prosjek: '))
        uc = uc + [ucenik]
    return uc
```

8. Zapis

```
def ispis(uc):
    print('ime      prezime      razred  projek')
    print('-----')
    for i in range(len(uc)):
        print('{:10s} {:12s} {:6s} {:.2f}'.format(uc[i].ime, uc[i].prez, uc[i].raz, uc[i].projek))
    return

def sort_abc(uc):
    for i in range(len(uc) - 1):
        for j in range(i + 1, len(uc)):
            if uc[i].prez > uc[j].prez or uc[i].prez == uc[j].prez and uc[i].ime > uc[j].ime:
                pom = uc[i]
                uc[i] = uc[j]
                uc[j] = pom

def sort_projek(uc):
    for i in range(len(uc) - 1):
        for j in range(i + 1, len(uc)):
            if uc[i].projek < uc[j].projek:
                pom = uc[i]
                uc[i] = uc[j]
                uc[j] = pom

def trazi_brisi(uc, izbor):
    nadjen = False
    ime = input('Ime: ')
    prezime = input('Prezime: ')
    for i in range(len(uc)):
        if uc[i].prez == prezime and uc[i].ime == ime:
            if izbor == 4:
                print('{:10s} {:12s} {:5s} {:.2f}'.format(uc[i].ime, uc[i].prez, uc[i].raz, uc[i].projek))
            else:
                uc.pop(i)
                break
            nadjen = True
    return nadjen

while 1:
    print('\n\n1) upis\n2) ispis po abecedi\n3) ispis po projeku\n4) traženje pojedinca\n5) brisanje pojedinca\n')

```

8. Zapis

```
izbor = int(input('izbor:'))
if izbor == 1:
    uc = upis(uc)
elif izbor == 2:
    sort_abc(uc)
    ispis(uc)
elif izbor == 3:
    sort_pros(uc)
    ispis(uc)
elif izbor == 4:
    if not trazi_brisi(uc,izbor):
        print('osoba nije na popisu')
elif izbor == 5:
    if not trazi_brisi(uc,izbor):
        print('osoba nije na popisu')
else: break
```

9. Multimedijski projekt

9.1. Od “Camera obscura” do Instagrama

Ljudi su nekad u prošlosti počeli tražiti načine kako portretirati poznate i nepoznate te na taj način njihov lik trajno ostaviti drugima u nasljeđe. Vrijeme je pokazalo da su slikarstvo i kiparstvo samo neki od načina koji su prilično postojani u prenošenju ljudskog lika.

Danas je velika vjerojatnost da ćete nekog svog pretka prije vidjeti na staroj crno-bijeloj fotografiji nego kao umjetnički portret. Ta jedna fotografija čuvala se i prenosila kao obiteljsko blago. Kako će neki vaš daleki nasljednik upoznati vaš lik možemo samo nagađati i strahovati jer smo svjedoci da učenik mobitelom snimi više digitalnih fotografija tijekom velikog odmora, nego što su (sveukupno) snimila braća Lumière.

Sadašnjost je već takva da znanstvenici uspijevaju rekonstruirati lica čovjekovih predača koristeći se fosilnim ostacima; Photoshop metode skrivaju istinu; Deepfakes mijenja stvarnost, a stvarni život zamjenjuje virtualni svijet u kojem se život prikazuje drukčijim nego jest.

Sve su to pitanja za sat Filozofije ili Etike koji slijedi nakon sata Informatike. U ovom trenutku nužno je sjetiti se i zapamtiti da sve što se jednom objavi na internetu, ostaje na internetu. Jednog će dana do toga moći doći i vaši roditelji, budući poslodavci i u kočnicu, vaša djeca.

Objava fotografija na Instagramu, objava videa na YouTubeu ili na TikTok aplikaciji dio su današnje svakodnevice. Ostavimo postrani sve etičke i moralne dvojbe vezane uz objavu takvih materijala i okrenimo se akademskim primjerima kojima možemo ispuniti nastavu informatike.

Multimedija je zajednički naziv za uratke u kojima kombiniramo video, sliku i zvuk u cjevlinu. U nastavku ćemo predstaviti jednu ideju za projekt u kojem ćemo uz međupredmetnu suradnju, čitanje, glumu, video, zvuk i fotografiju uči u svijet multimedije.

9.2. Projektni zadatak

Cilj: Osmisliti i kreirati video uradak koji će na zanimljiv i sažet način predstaviti neki od klasika svjetske književnosti.

Ideja: Kreativni tim sastavljen od pet do sedam učenika jednog razrednog odjela osmislit će scenarij, snimiti nekoliko video i audio materijala, pripremiti foto-album te od svega kreirati multimedijski uradak.

Realizacija: Odabrati jedan klasik svjetske književnosti. Npr., *Zločin i kazna*, Fjodora Mihajloviča Dostojevskog. Na nastavi Hrvatskog jezika treba osmislati scenarij koji će djelo sažeti u deset scena koje ga najbolje opisuju. Svaka će scena biti prikazana video zapisom u trajanju između dvije i četiri minute.. Scene možemo oživiti na razne načine. Navedimo neke.

1. Gluma – scenu se može odglumiti. Podijeliti uloge, osmislati dijalog te mobitelom snimiti video materijal koji će se naknadno obradivati;

9. Multimedijski projekt

2. Animacija – foto album montirati u video zapis;
3. Naracija – unaprijed pripremljen tekst možemo snimiti te ga dodati na video pozadinu koju smo za tu priliku posebno snimili;
4. Stop-animacija – scenu možemo snimiti koristeći tehniku stop-animacije;
5. Radio drama – jednu od scena možemo realizirati u obliku radio drame.

Ovo su neki od kreativnih načina kojima možemo realizirati naš projekt. Naravno, video, audio i foto materijale treba obraditi i okupiti u smislenu cjelinu.

Premijera: Premijeru uratka možete imati na nastavnom satu, a zatim ga objaviti na razrednom YouTube kanalu koji će biti kreiran za tu priliku.

Promocija: U sklopu promocije uratka možemo kreirati plakat koji će prikazivati lažni profil glavnog junaka klasika na nekoj društvenoj mreži. U ovom slučaju Rodiona Romanovića Raskoljnikova.

U nastavku ćemo opisati neke alate za obradu slike, videa i zvuka.

9.3. Obrada slike

GIMP (GNU Image Manipulation Program) je aplikacija otvorenog koda za obradu fotografija. Od svoga nastanka 1996. godine do danas razvio se u moćan alat koji stoji uz rame s komercijalnim alatima poput Adobe Photoshopa. Alat možete besplatno preuzeti sa stranice www.gimp.org/downloads i instalirati na računalo.

Nakon instalacije programa i njegova pokretanja, upoznajmo se s radnim okružjem.



Potraži na internetu sažetak Hamleta prikazan LEGO kockicama.

9. Multimedijski projekt

GIMP pruža mnoštvo mogućnosti. Pokažimo neke od tih mogućnosti na konkretnom primjeru.

Putovati svijetom prekrasno je. Upoznavanje ljudi, različitih kultura i krajeva obogaćuje nam duh i pogled na život. Međutim, ljudi nisu uvijek u mogućnosti putovati onoliko koliko bi željeli. No ne trebamo se obeshrabriti, ponekad je dovoljna samo mašta uz malo tehnologije kako bismo otputovali na sva mesta na kojima još nismo bili.

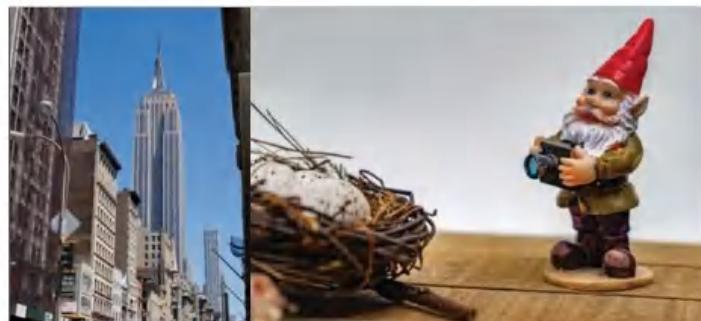
U poznatom filmu *Amélie* iz 2001. godine (*Le Fabuleux Destin d'Amélie Poulain*) glavna junakinja patuljka iz očeva vrta povjerava prijateljici. Ona radi kao domaćica zrakoplova i zbog prirode posla, zrakoplovom putuje po svijetu. Ideja je da njena prijateljica oču šalje fotografije patuljka iz različitih dijelova svijeta kako bi ga razveselila i izvukla iz depresije. Naravno, *Amélie* je mogla i mnogo lakše ostvariti svoj naum koristeći alate za obradu slike.

Vodite računa o autorskim pravima slika koje pronađete na internetu i nikad ni na koji način ne mijenjajte slike koje nisu označene za ponovnu upotrebu s promjenama.

Da biste doznali o kakvom je patuljku riječ, na YouTube-u pogledajte isječak iz filma: www.youtube.com/watch?v=8ekm_Dj0Uzl.

Dakle, najprije trebamo pronaći odgovarajuće fotografije vrtnog patuljka i Empire State Buildinga u New Yorku.

Nakon kratkog pretraživanja interneta odabrali smo sljedeće fotografije:



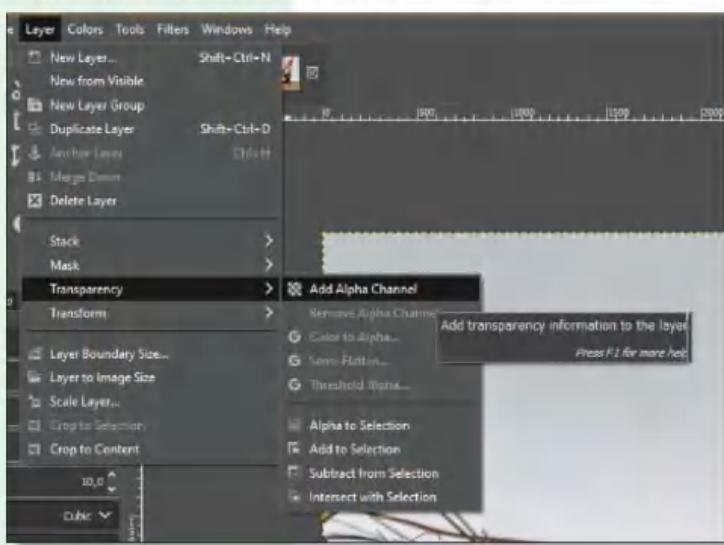
Dok obradujete ove slike, preporučamo da u pozadini slušate soundtrack filma Amélie.

Izrežimo vrtnog patuljka s druge fotografije. Otvorimo foto-

Slika 2. ESB i vrtni patuljak

grafiju u GIMP-u. Primijetite da se otvaranjem fotografije automatski kreirao novi sloj (Layer) u odjeljku slojeva.

Budući da namjeravamo izrezati vrtnog patuljka te ga umetnuti u fotografiju Empire State Buildinga, bit će nam potrebna prozirna podloga slike vrtnog patuljka. Stoga ćemo u izborniku Layer odabrati Transparency pa Add Alpha Channel.



Slika 3. Layer – Transparency - Add Alpha Channel

9. Multimedijski projekt

Kako izrezati vrtnog patuljka iz ove fotografije? Možemo odabratи dva alata: **Free select tool** i **Scissors select tool**. Potonji nam olakšava izrezivanje jer ima ugrađen algoritam koji pomaže u preciznijem prućenju crte razdvajanja između objekta koji izrezujemo i pozadine.



Slika 5. Izrezivanje dijela slike

Potom pratimo crtu razdvajanja vrtnog patuljka i pozadine. U mogućnostima odabranog alata označite mogućnost **Feather edges** i postavite brojčanu vrijednost. Na primjer – vrijednost 15. Time ćemo izbjegći neprirodno oštре rubove na izrezanom dijelu slike.



Slika 4. Free select tool i Scissors select tool

Kada smo spojili sve točkice oko vrtnog patuljka vrijeme je da uklonimo nepotrebni višak s fotografije. Pritisnite **Enter** ili kliknite unutar označenog prostora kako bi se izdvojeni dio označio. U izborniku **Select** odaberite mogućnost **Invert Selection**. Time smo označili sav prostor oko vrtnog patuljka.

Pritisnite **Delete** na tipkovnici ili odaberite iz izbornika **Edit > Clear**. Kako bi vjerojatno oduševljeno rekla *Amélie*: Voilà, pozadina iza vrtnog patuljka je nestala!

Vježba 1.

Pokušaj izrezati sliku koristeći alat **Free Select Tool**. U čemu se razlikuju ova dva alata? U kojim slučajevima je bolje koristiti jedan, a u kojim drugi alat za izrezivanje?

Preostaje nam još prilagoditi veličinu slike. Pronadite unutar alatne trake **Crop Tool** □ i izrežite patuljka iz slike. Trebali biste dobiti rezultat kao na slici:



Slika 6. Slika prilagođene veličine

Fotografiju trebamo spremiti u .png formatu jer će se tako očuvati prozirnost pozadine koja nam je potrebna. Kako bismo to napravili u izborniku **File** birajmo **Export As** i upišimo ime slike.

Sada u programu GIMP otvorimo sliku zgrade Empire State Buildinga.

Kako bismo ubacili izrezanog patuljka unutar slike, otvorimo ga kao novi sloj. Dakle, iz izbornika **File** odaberimo **Open as layer** i pronađimo sliku patuljka. Alternativno sliku mo-

Ako odaberemo mogućnost **Save As**, uradak će biti pohranjen u .xcf formatu. To je format koji pamti što smo radili na slici. Na taj način možemo pohraniti sve dotada napravljeno (i idući put nastaviti gdje smo stali).



9. Multimedijski projekt



Slika 7. Vrtni patuljak u novom sloju

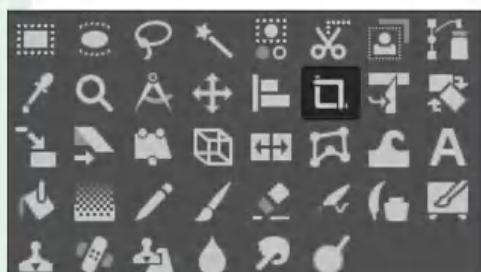
žemo otvoriti kao novi sloj ako je povučemo iz *Eksplorera za datoteke* i ispustimo (*drag and drop*) u već otvorenu sliku.

Ako je potrebno, veličinu ubačene slike možemo prilagoditi koristeći alat *Crop* iz alatne trake. Rezultat bi trebao biti nalik ovome:

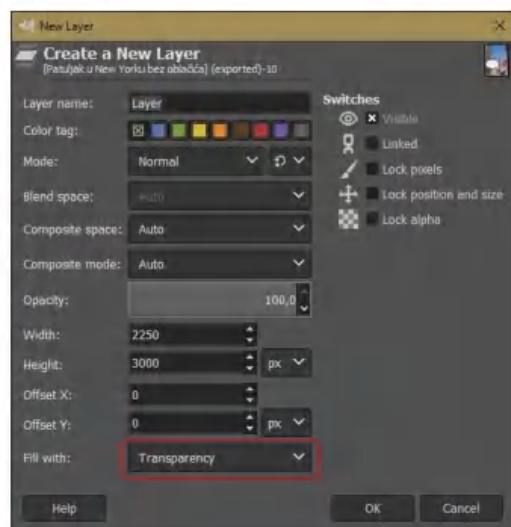
Mogli bismo postaviti i jedan oblačić u kojem bi se upisali riječi vrtnog patuljka. GIMP nije program koji je namijenjen crtanju, no jednostavne stvari poput dodavanja malog oblačića možemo izvesti.

Prilikom obrade slika dobra je praksa svaki novi element koji postavljamo u sliku, postaviti u novi sloj. U odjeljku sa slojevima pritiskom na ikonu stvorimo novi sloj. Otvorit će nam se dijalog u kojem možemo urediti detalje novog sloja.

Tu mu možemo dodijeliti ime, urediti njegovu veličinu i sl. Nama je u ovom slučaju najvažnije da novi sloj bude proziran. Stoga, postavimo mogućnost **Fill with na Transparency**. Što god budemo radili na novom sloju, možemo lako izmijeniti ili čak ukloniti bez da našavamo prethodni rad na pozadini. Oblačić ćemo stvoriti kombinacijom *Ellipse Select Toola* i *Paths Toola* koje nalazimo u alatnoj traci.



Slika 9. Ellipse Select Tool i Paths Tool



Slika 8. Novi sloj



Slika 10. Završna fotografija

Vježba 2.

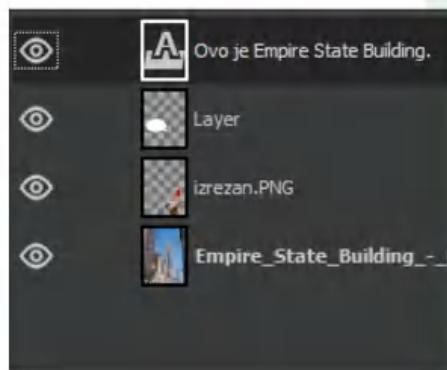
Pronadi na internetu digitalni sadržaj pomoću kojeg ćeš saznati kako se pravi oblačić za tekst. Uradi tekstualni oblačić kao na slici.

Primijetite da su u odjeljku sa slojevima vidljivi svi prethodno napravljeni slojevi. Oni se pritiskom na ikonicu oka mogu učiniti nevidljivima.

9. Multimedijski projekt

Slojevi su poslagani jedan na drugi. Promjenom njihova redoslijeda, pomoću *drag and drop* tehnike, npr. kada bi se donji sloj u kojem je slika Empire State Buildinga, podigao na vrh, više ne bi bili vidljivi niti vrtni patuljak niti njegov oblačić s tekstom. Zašto je tako?

Ovo su samo neke od mogućnosti. Mnoštvo drugih čeka otkrivanje i kreativno korištenje.



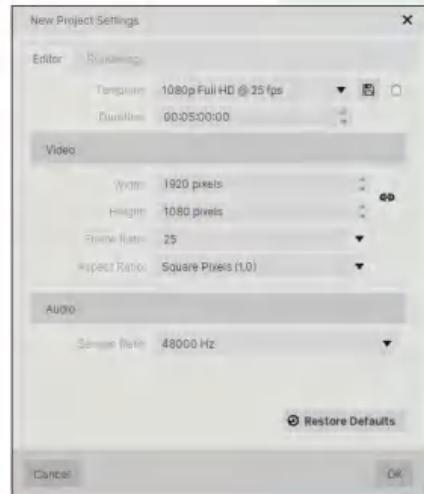
Slika 11. Uređivanje vidljivosti slojeva

9.4. Obrada videoa

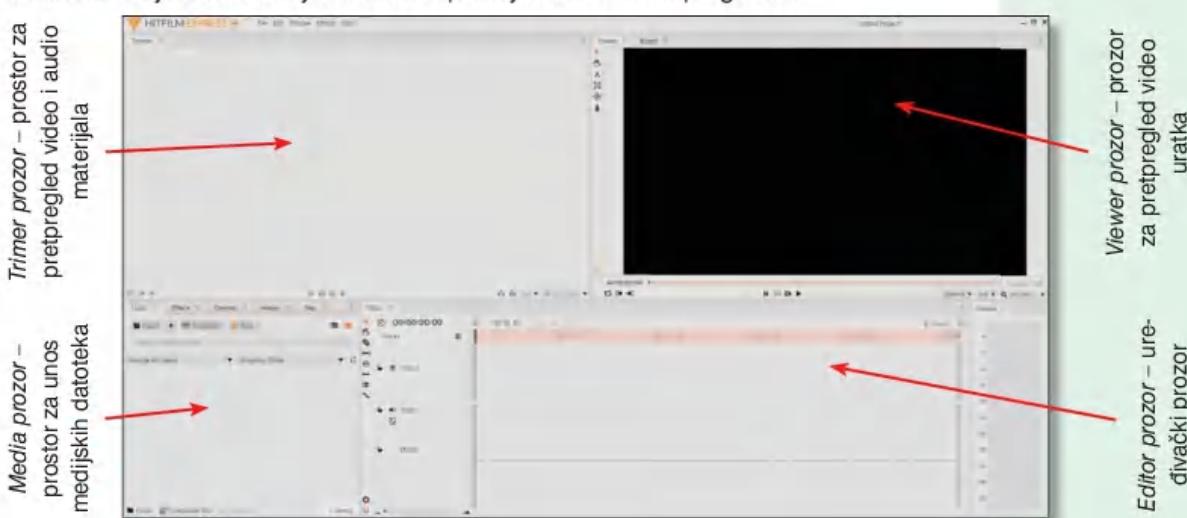
HitFilm Express je besplatni softver za izradu i uređivanje video uradaka. Posjetite stranicu fxhome.com/hitfilm-express, a nakon obvezne besplatne registracije na vaš će mail stići povznica pomoću koje ćete moći skinuti instalaciju. Ovdje ćemo predstaviti samo osnovne mogućnosti programa, a na mrežnim stranicama *HitFilm Expressa* naći ćete mnogo video uputa o tome kako koristiti ovaj softver. Naravno, i YouTube je jedan od izvora podataka koji možete koristiti.

Krenimo s izradom prvog videa. Pronadite ikonicu *New* na prozoru ili odaberite *File > New*. Otvorit će vam se prozor poput ovog.

U njemu možemo postaviti početna svojstva videa u našem projektu poput rezolucije slike i kvalitete zvuka. Podrazumijevane vrijednosti postavljene su za video Full HD rezolucije i trajanja 5 minuta. Duljinu videa uvijek možemo promijeniti u samom programu.



Slika 12.
Novi projekt



Slika 13. Radno okružje *HitFilm Expressa*

9. Multimedijski projekt

Jedan od načina obrade videa i mogućnosti samog programa je kreiranje videa na zadanu temu. Ideje koje će vam pasti na pamet tijekom tog procesa vjerojatno će biti izvedive, a jedino ograničenje bit će vrijeme koje želite uložiti u istraživanje mogućnosti.

Za korištenje pjesme *Welcome To The Jungle* nemamo autorska prava, stoga taj video uradak nećemo javno distribuirati.

Trimmer

U ovom smo udžbeniku za primjer uzeli kratak video šetnje po Times Squareu u New Yorku koji smo sami snimili.

 Videozapis **3g9TimesSquare.mp4** možete preuzeti s mrežne stranice uz udžbenik.

Iskoristit ćemo i sliku našeg vrtnog patuljka koju smo obradili u poglavlju o GIMP-u. Kao glazbu za video koristit ćemo besmrtnu pjesmu Guns N' Rosesa, *Welcome To The Jungle*.

Sve medijske datoteke (video, audio ili slike) koje ćemo koristiti u izradi videa unosimo u program na prozoru s označkom *Media*. Možemo ih unijeti pomoću gumba *Import* ili ih možemo povući i ispustiti unutar *Media* prozora (*drag and drop*).

Primijetimo da bilo koji uneseni materijal možemo pregledati u *Trimmer* prozoru. Možemo ga čak skratiti ili iz njega izdvojiti samo neki željeni detalj koji ćemo ubaciti u završni video.



Slika 14. Prozor Trimmer

Izreži početak i kraj
video isječka.

Ubaci isječak u
uređivački prozor.

Ako ništa nije potrebno uređivati, snimku iz *Media* prozora samo odvučemo na uređivački prozor.

Primijetite da se na *Viewer* prozoru pojavio video spreman za pretpregled. Isto tako, primijetite da se na *Editor* prozoru pojavio video i audio kanal naše snimke. Jačinu zvuka u audio kanalu možemo pojačavati i smanjivati, a možemo i potpuno ukloniti zvuk.

9. Multimedijski projekt



Slika 15. Uređivanje jačine zvučnog zapisa

Vrijeme je da ubacimo glazbenu podlogu u naš video. Prije svega moramo stvoriti novi audio kanal koji će "ugostiti" glazbenu podlogu.

Desnim klikom na postojeći *Audio 1* kanal dobivamo izbornik iz kojeg biramo **Insert track**. Sada u novostvoreni *Audio 2* prostor iz *Media* odjeljka dovucimo medijsku datoteku glazbene podloge. Ako je glazbena podloga preduga, otidite na kraj *Audio 2* kanala dok se ne pojavi znak zelene vitičaste zagrade.



Slika 16. Skraćivanje duljine zvučnog zapisa

Pomoću miša uhvatimo rub i skratimo trajanje glazbene podloge na duljinu trajanja videa. Svaki se video ili audio kanal može preimenovati. Ako radimo video s brojnim zvučnim efektima, korisno je nazvati svaki kanal posebnim imenom.

Vježba 3.

Na *Viewer* prozoru pokreni video. Poigraj se jačinom zvuka originalnog videa i glazbene podloge.

Vrijeme je da uključimo vrtnog patuljka u obilazak Times Squarea. Stvorimo novi *Video 2* kanal i u njega dovucimo sliku vrtnog patuljka iz *Media* prozora.

Ako je ubaćena slika prevelika ili premala za video, u *Viewer* prozoru mišem lako možemo prilagoditi veličinu i položaj slike u snimci. Pritom držite tipku *Shift* na tipkovnici kako biste očuvali omjer dimenzija slike. Ubačenu sliku u video možemo rotirati i zrcaliti. Zar nije izvrsno što smo u poglavlju GIMP napravili pozadnu vrtnog patuljka prozirnom?

Podrazumijevana duljina video isječka kreiranog od slike je 5 sekundi. Povlačeći kraj isječka možemo produžiti ili skratiti njegovo trajanje. Povlačeći cijeli isječak lijevo ili desno po kanalu određujemo trenutak u kojem će se patuljak pojaviti u videu.



Slika 17.
Prilagođavanje
veličine slike

9. Multimedijski projekt



Slika 18. Animacija gitare 1. dio

Vježba 4.

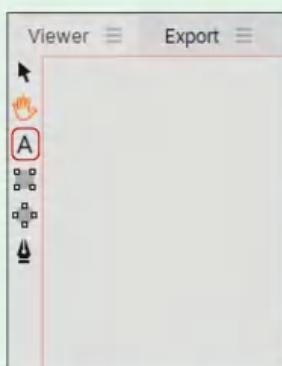
Pronađi na internetu sliku električne gitare jer mislimo da je vrtnom patuljku već dosta fotografirana.

Pokušajmo dodati sliku električne gitare vrtnom patuljku u ruke. Kreirajmo novi Video 3 kanal i unesimo u njega sliku gitare. Kliknimo desni klik na video isječak gitare i odaberimo mogućnost *Speed/Duration*. Neka trajanje isječka bude 1 sekunda.

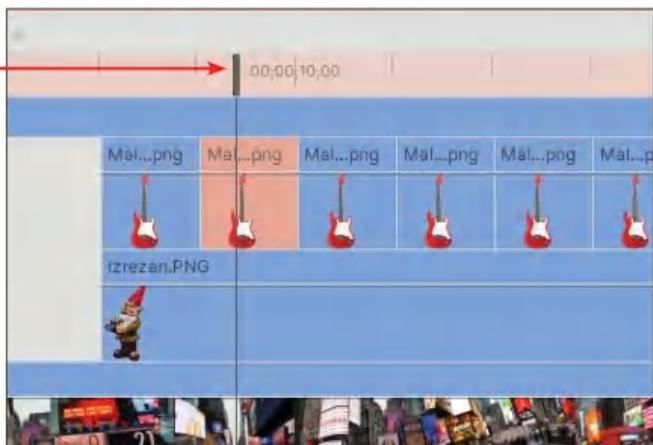
Kopirajte isječak gitare i zalijepite ga onoliko puta koliko traje video isječak vrtnog patuljka. Spojite isječke gitare da se nastavljaju jedan na drugoga.

pokazivač reprodukcije

Postavimo pokazivač reprodukcije na svaki drugi isječak gitare te u *Viewer prozoru* zarotirajmo gitaru tako da se dobije efekt sviranja.



Slika 20. Dodavanje teksta

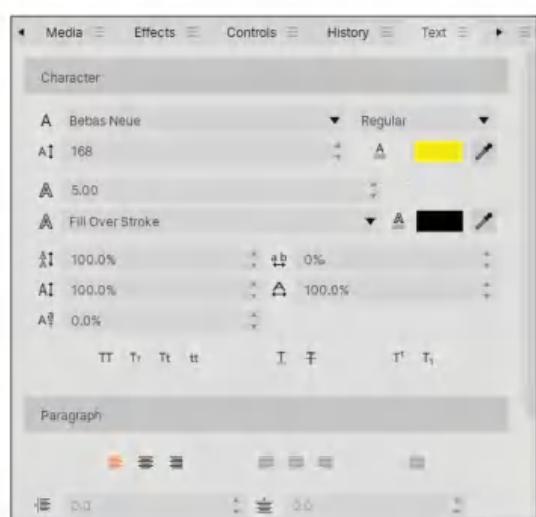


Slika 19. Animacija gitare 2.dio

HitFilm Express pruža jednostavnu mogućnost pisanja teksta po video uratku. Postavimo pokazivač reprodukcije na vrijeme na kojem želimo da nam se pojavi tekst na ekranu te na *Viewer prozoru* odaberimo ikonicu sa slovom A.

Kliknimo unutar videa i napišimo WELCOME TO THE JUNGLE. Na prozoru za uređivanje pojавio se novi video kanal, a trajanje video isječka s tekstom je opet 5 sekundi. Naravno, to možemo promijeniti ručno povlačeći rubove isječka ili postavljajući konkretnu vrijednost u *Speed/Duration* mogućnosti video isječka.

Veličinu i vrstu slova možemo urediti unutar *Text* prozora koji se nalazi поред *Media* prozora. Dobili smo umet-



Slika 21. Prozor za uređivanje teksta

9. Multimedijski projekt

nuti tekst unutar videa. Za bolje uočavanje slova na svjetloj i tamnoj podlozi postavite i *Fill Over Stroke*. Kako bi opet rekla *Amélie*: *Voilà*, video je spreman za pregled!



Slično kao u programu *GIMP*, ako odaberemo *File > Save As*, program će spremiti cijeli projekt u specijaliziranu datoteku ekstenzije *.hfp*. Ovu datoteku otvara isključivo *HitFilm Express* i na taj način možemo spremiti naš posao za kasniji nastavak rada.

Slika 22. Završni video uradak

Za kraj, spremimo video u *.mp4* video formatu. Da bismo to uradili odaberemo *File > Export* te napišimo ime videa i lokaciju na koju će se uradak na računalo pohraniti.

 Videozapis [3g9zavrsni_video.mp4](#) možete preuzeti s mrežne stranice uz udžbenik.

Vježba 5.

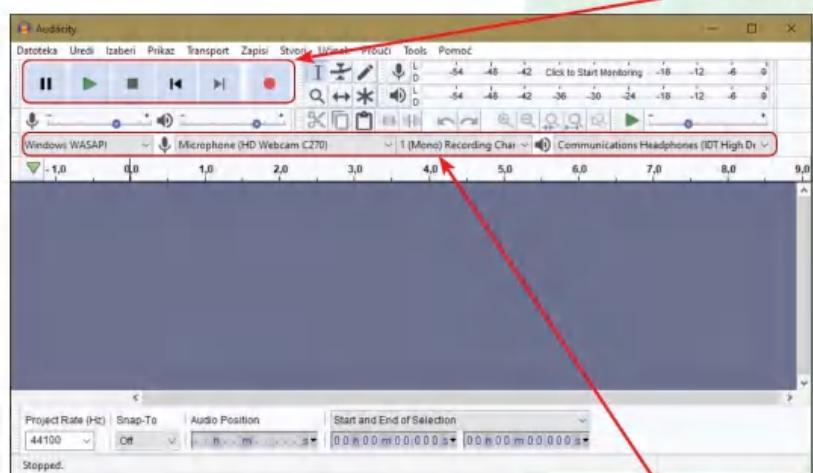
HitFilm Express dolazi s mnoštvom efekata. Označi na uređivačkom prostoru jedan video isječak te isprobaj nekoliko efekata koji se nalaze u izborniku Effects.

Ovo je samo dio mogućnosti koje tek treba otkriti i primijeniti. Preporuka je da budete znatiteljni i uporni te da se uvijek sjetite kako postoje i profesionalni redatelji kojima ni milijuni dolara nisu pomogli da snime ili obrade stvari kako treba.

9.5. Obrada zvuka

Pored pozadinske glazbe u video uradcima, često nam je potrebna i glasovna naracija. Nažalost snimanje glasovne naracije ne može se izravno napraviti u programu *HitFilm Express*. U tu svrhu na raspolaganju imamo alate za snimanje i obradu zvuka.

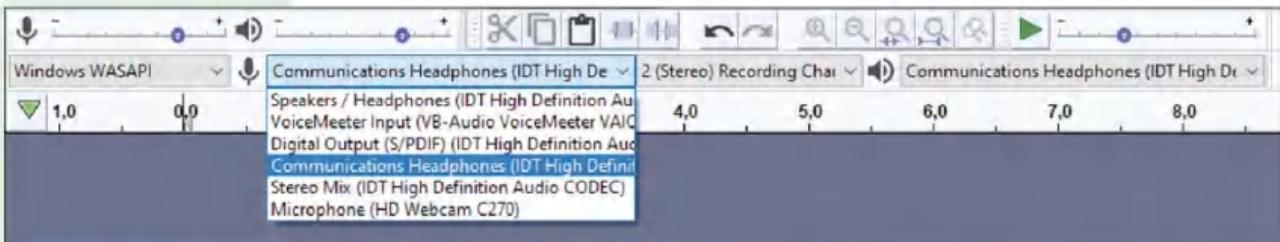
Audacity je besplatan softver otvorenog koda koji možete preuzeti na stranici www.audacityteam.org.



Slika 23. Prozor programa Audacity

kontrole izvora i izlaza zvuka

9. Multimedijski projekt



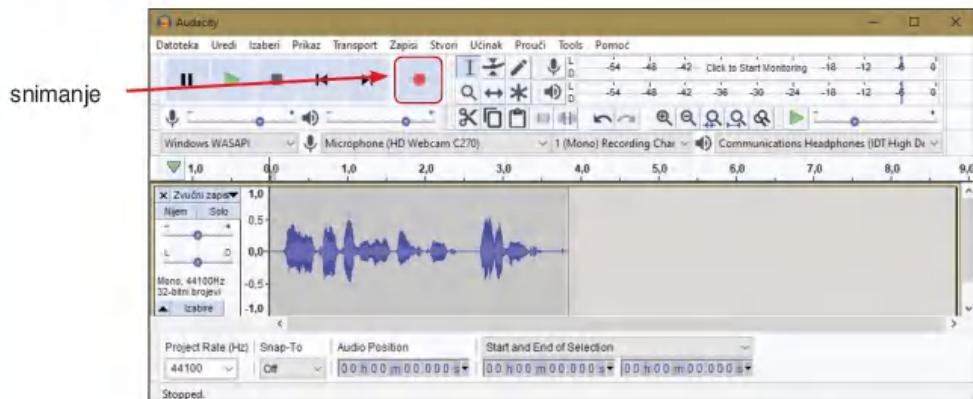
Slika 24. Izvor zvuka

Pomoću Audacityja možemo snimati zvuk s raznih izvora koji su dostupni na računalu. Izvor odakle snimamo zvuk možemo postaviti na traci s kontrolama izvora i izlaza zvuka.

Vrsta izvora za snimanje zvuka ovisi o tome kako je pojedino računalo hardverski opremljeno. Ako želimo snimati s mikrofona, onda koristimo mikrofon u padajućem izborniku. Ako želimo snimati zvuk s računala onda odaberimo *Speakers / Headphones* ili *Stereo Mix*.

Ponovno se vraćamo vrtnom patuljku na putu oko svijeta. U prijašnjim poglavljima uređili smo sliku vrtnog patuljka i postavili ga u video šetnje po Times Squareu. Snimimo neku prigodnu naraciju za šetnju vrtnog patuljka po New Yorku.

Za izvor zvuka odaberemo *Mikrofon* i pokrenemo snimanje.



Slika 25. Snimanje naracije

Nakon što smo snimili govor preslušamo snimljeni materijal. Ako smo zadovoljni, možemo se poigrati snimljenim zapisom.

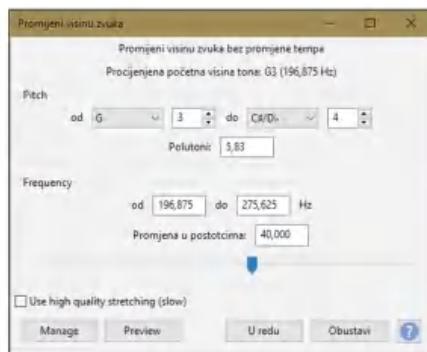
Izmijenimo visinu glasa na snimljenom materijalu kako bi više odgovarala glasu vrtnog patuljka. Označimo cijelu snimku mišem ili pritisnimo *Ctrl+A* na tipkovnici. U izborniku *Učinak*, odaberimo **Promijeni visinu zvuka**. Dobit ćemo prozor s mogućnostima uređivanja kao na slici.

9. Multimedijski projekt

Postavimo promjenu u postotcima na 40,000 % i pritisnemo gumb *U redu*. Preslušajmo snimku. Ako smo zadovoljni postignutim učinkom, pritiskom na *Datoteka > Export* izvezemo snimku u mp3 format.

Vježba 6.

Postavite snimljeni i obrađeni audio zapis u video šetnje po Times Squareu.



Slika 26. Promjena visine zvučnog zapisa

Od problema do rješenja

Na početku poglavlja pisali smo o kreiranju video uratka koji će na zanimljiv i sažet način predstaviti neki klasik svjetske književnosti. Kao ideju spomenuli smo klasik Zločin i kazna, a ovdje ćemo sada opisati konkretnu vizualizaciju klasika Stranac, Alberta Camusa.

Podijelimo djelo u deset scena. Za svaku scenu raspišimo njen opći opis te prijedlog realizacije. Tijekom snimanja scena bit će potrebno još detaljno raspisati korake realizacije koji uključuju detaljni scenarij, definiranje knjige snimanja, snimanje i obradu materijala te još puno toga.

1. scena

Danas ili jučer umrla mi je majka. Zapravo je svejedno. I najobičnija ništarija i najveći junak na kraju će izdahnuti i postati pepeo. Morao sam obavijestiti svojeg poslodavca, no umjesto da mi je izrazio sućut, on je bio nervozan što će dva dana izostati s posla. Prije odlaska na put najeo sam se u najdražem restoranu kod Celesta. Poslijepodnevna me vrućina u autobusu ulijuljala. Miješala se ta sparina s bučnim autobusom i užegлом cestom, pa sam zaspao.

Gluma. Mersault se vozi u autobusu naslonjen na staklo i drjema. Dok promiču kadrovi vidljivi kroz prozor, u pozadini se čuje njegov glas koji prepričava zadnje događaje. Scenu iz restorana treba posebno odglumiti i ubaciti u snimak vožnje kao sjećanje.

Stranac je klasik svjetske književnosti koji se nalazi na popisu književnih ispitnih djela za školski esej na Državnoj maturi.

2. scena: Maria i Mersault

Sprovod me poprilično izmorio. Dok sam se ujutro brijao dosjetio sam se da bih se mogao opustiti u moru na lučkom kupalištu. Na plaži je bilo mnogo mladih, a onda sam susreo Mariju Cardonu, daktilografkinju koja mi se oduvijek svidala. Brčkali smo se u moru, gledao sam u veliki plavetni svod i pomalo drijemao. Marija je s osmijehom pristala otići u kino na komediju s Fernandelom. Doduše, uznemirilo ju je kad je shvatila da mi je umrla majka. No već je do navečer potpisnula tu misao i zdušno se smijala filmu.

9. Multimedijski projekt

Animirani slijed. Niz fotografija kojima dočaravamo radnju scene složit ćemo u animirani slijed. Scenu ćemo opisati iz Marijine perspektive čija će naracija u pozadini pratiti animaciju. Glazba u podlozi videa treba dočaravati promjene u psihičkom stanju junaka tijekom dana.

3. scena: Raymond i Mersault

Na hodniku sam svoje zgrade susreo susjeda Raymonda. Taj malen, plečat tip; skladištar prividno dobrih manira, bio je uvijek savršeno dotjeran. Govorilo se da se bavi nekim sumnjivim poslovima. To me nije odviše zabrinjavalo. Čak ni činjenica da ima zavoj oko ruke i da sigurno upada u različite neprilike. Pozvao me k sebi na krvavicu i vino... Savršen je to izgovor da ne moram kuhati! Tada mi je otkrio da je cijelu priču zakomplicirala jedna žena. Bila je to njegova ljubavnica za koju je tvrdio da ga vara i troši njegov novac. Odlučio sam mu pomoći da joj se osveti.

Radio drama. Kroz razgovor dva lika trebamo opisati i dočarati njihov susret. Dodatnim zvučnim efektima kao što su hod, otvaranje vrata, točenje vina, zvuk prženja krvavice, odjek glasa u hodniku, smijeh žene obogatit ćemo dramsku radnju.

4. scena: Salamano i Mersault

Jedan je od mojih susjeda Salamano, starac koji se vuče vodeći na uzici starog psa. Ne možete pogriješiti nalikuju jedan drugome. Prepeličar ima neku gadnu kožnu bolest. Kad sam ga susreo, bio je potpuno izbezumljen. Mrmljao si je nešto u bradu, a pogledom je pretraživao svaki zakutak ulice. Izgovarao je neke vulgarizme kojima je i inače oslovljavao svojeg psa. Tada mi je rekao da ga je izgubio. Potanko mi je objasnio da su bili na Vojnom vježalištu, a on je, utopljen u gomili, želio pogledati kralja izvrđavanja. Nakon toga njegov je pas isčeznuo. Imali su takav odnos; Salamano ga je vječito tukao, ali bio je navikao na njega. Na svašta se čovjek navikne.

Gluma. Mersault gleda s prozora i prati susjeda Salamana kako šeta. Prisjeća se njihovog prvog susreta, odnosa Salamana i psa, zamišlja i vizualizira trenutak u kojem Salaman gubi psa. Alternativno možemo snimiti scenu iz perspektive psa koji priповijeda svoja iskustva.

5. scena: Dolazak na plažu

Jednog smo dana Maria i ja pristali otići s Raymondom na plažu. Prije nego što smo se ukrcali u autobus Raymond je u ulici vidio Arape, naslonjene na trafiku. Suptilno mi je namignuo kako bi mi dao znak da se tamо nalazi i brat njegove ljubavnice s kojim je bio u svadbi. Mariji nije ništa bilo jasno, no nakon što sam joj pojasnio situaciju kao da je nasmjela da bi bilo dobro da se ne zadržavamo i da se požurimo. Da bismo došli do mора, morali smo proći puteljkom koji se uzdizao, a zatim spuštao do tog kopnenog plavetnila. Maria se zabavljala na neobičan način: svojom je platnenom torbom zamahivala i kidala latice cvijeća. Prolazili smo kroz gusto poslagane urbane kuće sa zelenim i bijelim ogradama. Neke su se ograde skrivale iza prekrasnih tamarisa, a neke su stajale na goleti.

9. Multimedijski projekt

Uz pomoć Googleove usluge *Google Street View* pronaći ćemo u Alžиру lokaciju koja najbolje odgovara ovoj sceni. Kretanjem po mapi i snimanjem tog kretanja vizualizirat ćemo mjesto radnje. U pozadini kroz razgovor likova opisujemo radnju.

6. scena: Druženje kod Massona

Na druženje nas je zapravo pozvao Raymondov prijatelj Masson; zanimljiv, visok tip s punašnom ženom Parižankom. Drvena kolibica na kraju obale jednim je dijelom bila smještena uz liticu, a njezini prednji stupovi bili su uronjeni u more. Svidjela mi se pa sam to i rekao Massonu. Zadovoljno se smješkao i otkrio nam da u njoj provodi sve vikende i praznike. Ljubazno nam je ponudio svježe ulovljene ribe, a zatim smo se Masson, Maria i ja uputili na kupanje.

Stop animacija. Hint – LEGO kockice.

 Iduće 3 scene preuzmite s mrežne stranice uz udžbenik: [3g9stranac.pdf](#).

10. scena: Ispovjednik

Tišinu je mojih mjeseci provedenih u ćeliji razbio isповjednik. Bio je zabrinut jer odbijam njegove posjete. Ispovjednik me istovremeno živcirao i tišio. Morao sam mu objasniti da ne vjerujem u Boga i da mi njegov dolazak ništa ne predstavlja. No priznao sam mu da se bojam, uvjeren da je to sasvim normalno. Optimistično mi je ukazao na to da bi mi upravo u ovakvim trenucima mogla pomoći vjera. No ja nisam želio ničiju pomoć. Posebno sam planuo kad mi je rekao da će se moliti za mene. Tada sam ga grubo uhvatio za ovratnik svećeničke halje i urlao na njega. Želio sam da zna ono najvažnije, ono što gori u meni: ja sam siguran da sam to što jesam i prihvaćam svoju sudbinu. Mogao sam biti netko drugi, živjeti nečiji tudi život. No zašto bih ja to učinio? Ta naposljetku ništa nije važno jer naš život nema nikakvog smisla.

Objave na društvenim mrežama. Scenu ćemo prikazati kroz objave koje Mersault objavljuje na svom Twitter profilu iščekujući kraj.

 Prikaz jednog lažnog Mersaultovog profila na jednoj od društvenih mreža možete preuzeti s mrežne stranice uz udžbenik: [3g9mersault.pdf](#).

Za kraj, nakon što snimimo i obradimo video, možemo se posvetiti promociji i premijeri. Premijeru možemo imati na razrednom YouTube kanalu otvorenom posebno za tu prigodu, a promociju možemo odraditi uz pomoć filmskog plakata koji ćemo staviti na oglasnu ploču škole.

Tko je Arapin kojega je ubio Meursault, glavni junak Camusova romana? To pitanje okosnica je višestruko nagrađivanoga romana „Meursault, protuistraga“ Kamela Daouda.



10. Baze podataka

Alphabet, krovna kompanija Googlea, početkom 2020. godine približila se vrijednosti od bilijun dolara. Ono što Alphabet čini jedinstvenim je činjenica da su to uspjeli ne plaćajući svoju temeljnu uslugu. Njihovu tražilicu Google i servis elektroničke pošte Gmail koristimo bez naknade.

Veliki dio njihove vrijednosti dolazi iz marketinškog potencijala. Google raspolaže velikim brojem podataka o svakom od nas koje može ciljano koristiti za optimizaciju reklamnih kampanja. Ovo nije samo specifičnost Googlea već i ostalih kompanija koje prikuđaju ili imaju pristup velikim količinama podataka o svojim korisnicima.

Količina podataka koja se generira svakim danom raste. Pitanje je kako te podatke oblikovati i upotrijebiti što efikasnije i korisnije. Od šume podataka koja se nudi, nije svejedno kome ćemo i kako ponuditi podatke. Zato je nužno dobro i smisleno organizirati prikupljene podatke.



Oracle je jedna od najpoznatijih svjetskih IT kompanija koja svoj uspjeh duguje radu s bazama podataka. Njihova baza podataka koristi se u mnogim profesionalnim okruženjima.

Zašto koristiti baze podataka, a ne recimo Excel? Upotreboom Excela za pohranu podataka samo jedna osoba u istom trenutku može koristiti te podatke. Excel poznaje zapis podataka samo u retke i stupce, dok baza omogućuje višedimenzionalnu pohranu. Pronaći u Excelovoj tablici podatke koji zadovoljavaju neke zajedničke preuvjetne znati komplikirano, dok je baza dizajnirana upravo za takve svrhe.

U današnje vrijeme najraširenije su **relacijske baze podataka**. Primjerice, većina društvenih mreža koristi relacijsku bazu za pohranu podataka poput popisa korisnika, popisa poznanstava, popisa objavljenih poruka i sl. Ipak, relacijske baze podataka nisu prikladne za pohranu baš svih vrsta podataka. Npr., kada postoji potreba za upitom oblika "Jesu li osoba A i B ikako povezane jedna s drugom?" tada relacijske baze podataka ne mogu dati odgovor na to pitanje.

Za rješenje takvog problema prikladno je koristiti **graf baze podataka**. To su baze podataka koje podatke spremaju kao čvorove i veze između njih. Ovakvu bazu lako možemo vizualizirati ako zamislimo da su u njoj direktno pohranjeni svi gradovi na svijetu (čvorovi) i sve ceste koje se nalaze između njih (veze). Kada se svi podaci o ljudima i prijateljstvima među njima spreme u graf bazu podataka, lako možemo odgovoriti na postavljeno pitanje zbog karakteristika baze i unaprijed implementiranih algoritama kao što su BFS, DFS i Dijkstra.

Svaki put kada je za određeni set podataka prikladno pitati jesu li dvije značajke u podacima ikako povezane, valja razmotriti upotrebu graf baze podataka za rješavanje problema.

10. Baze podataka

10.1. Osnove baze podataka

Baza podataka organizirana je i uređena cjelina međusobno povezanih podataka spremnih bez nepotrebne redundancije (zalihosti, viška informacija) u vanjskoj memoriji računala, a u svrhu računalne obrade. Današnje baze uz standardne tekstualne i numeričke podatke pohranjuju i multimedijiske podatke (slika, zvuk, video i sl.).

Organiziranje podrazumijeva da su podaci pripremljeni tako da se mogu jednostavno koristiti, tj. pregledavati, pretraživati, sortirati, uspoređivati, mijenjati, nadopunjavati, bri-sati i dr. Sva dodavanja, promjene, brisanje i čitanje podataka obavljuju se posredstvom posebnog softvera – **sustava za upravljanje bazom podataka** (*Data Base Management System* – DBMS). DBMS je poslužitelj (server) baze podataka. On oblikuje fizički prikaz baze u skladu s traženom logičkom struktururom, te u ime klijentata obavlja sve operacije podataka. Danas postoji nekoliko široko zastupljenih DBMS-a:

- komercijalni: MS Access, Oracle, MS SQL, DB2
- besplatni: MySql, PostgreSQL, SQLite itd.

DBMS - Data Base
Management System

Upotreboom baza podataka nastoje se ostvariti sljedeći ciljevi:

- **Fizička nezavisnost podataka** – razdvaja se logička definicija baze od njezine stvarne fizičke građe, odnosno promjena fizičke građe ne zahtijeva promjene u postojećim aplikacijama.
- **Logička nezavisnost podataka** – razdvaja se globalna logička definicija cijele baze podataka od lokalne logičke definicije za jednu aplikaciju, odnosno promjena globalne logičke definicije (npr. dodavanje novih zapisa ili veza) ne zahtijeva promjenu u postojećim aplikacijama.
- **Fleksibilnost pristupa podacima** – u mrežnim i hijerarhijskim bazama, načini pristupa podacima bili su unaprijed definirani. Danas se podrazumijeva da korisnik može slobodno pretraživati podatke i uspostavljati veze među podacima. Relacijske baze udovoljavaju tom zahtjevu.
- **Istovremeni pristup podacima** – baza mora omogućiti većem broj korisnika istovremeno korištenje istih podataka, pri čemu ne smije dolaziti do međusobnog ometanja i svaki korisnik treba imati dojam da je baza samo njegova.
- **Čuvanje integriteta** – automatsko očuvanje, korektnost i konzistencija podataka u situaciji kad postaje greške u aplikacijama, te konfliktne istovremene aktivnosti korisnika.
- **Mogućnost oporavka nakon kvara** – osiguravanje pouzdane zaštite baze za slučaj kvara hardvera ili pogreške u radu sistemskog softvera.
- **Zaštita od neovlaštenog korištenja** – mora postojati mogućnost da se korisnici mogu ograničiti prava korištenja baze, tj. da se svakom korisniku reguliraju ovlašte-nja što smije, a što ne smije raditi.
- **Zadovoljavajuća brzina pristupa** – operacije s podacima moraju se odvijati dovoljno brzo, u skladu s potrebama određene aplikacije.
- **Mogućnost prilagođavanja i kontrole** – velika baza zahtijeva stalni nadzor: pra-

10. Baze podataka

ćenje performansi, rutinsko pohranjivanje rezervnih kopija podataka, reguliranje ovlasti korisnika i slično.

Podaci su logički organizirani po nekom modelu. Model čini osnovu za osmišljavanje, definiranje i implementiranje baze podataka. Današnji DBMS-i podržavaju četiri osnovna modela:

- **Relacijski model** zasnovan je na matematičkom pojmu relacije. Podaci i veze među podacima prikazuju se preko dvodimenzionalnih struktura. Većina suvremenih baza podataka služi se ovim modelom.
- **Mrežni model** može se predložiti usmjerenim grafom u kojem su čvorovi podaci, a lukovi među čvorovima definiraju veze među podacima.
- **Hijerarhijski model** čine hijerarhijski organizirani podaci. Zastupljenost ovih baza nije velika, ali imaju svoje prednosti kao što je brzo spremanje i dohvaćanje podataka.
- **Objektni model** inspiriran je objektno-orientiranim programskim jezicima te on prihvata semantiku objekata podržanu u objektno-orientiranim programskim jezicima.

10.2. Relacijske baze podataka

Osnovne principe i strukture relacijskog modela podataka iznio je matematičar E.F. Codd 1971. godine u knjizi *A Relation Model of Data for Large Shared Data Banks*.

Osnovni element relacijskog modela jest **relacija**. Codd je relaciju zamišljaо kao tablicu koja se sastoji od stupaca – vrijednosti atributa i redaka – n-torki.

	stupac – atribut				
redak – slog					

Slika 1. Vizualizacija relacije

Primarni ključ je atribut ili skup atributa čija vrijednost jednoznačno određuje pojavljivanje svake n-torke tj. svakog redaka relacije. Primarni je ključ polje ili skup polja s jedinstvenim vrijednostima u tablici. Kandidati za ključ su svi atributi koji u nekoj relaciji imaju sve vrijednosti jedinstvene. Među kandidatima za ključ odabire se primarni ključ. Pojam primarnog ključa iznimno je važan jer je pomoću njega moguće jednoznačno identificirati i pristupiti svakoj n-torci u relaciji.

Relacijske baze podataka se sastoje od relacija, a među relacijama ne definiraju se никакve veze u fizičkom smislu, već su relacije povezane preko stranih – vanjskih ključeva.

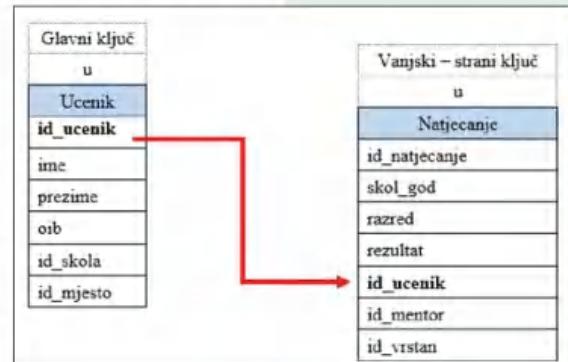
10. Baze podataka

Strani, vanjski ključ u jednoj relaciji je atribut koji je primarni ključ u drugoj relaciji. Ova veza nije fizička, već logička, tj. nema nikakve fizičke implementacije ovih veza. To je jedna od osnovnih razlika između relacijskog i ostalih modela podataka. Za razliku od primarnog ključa, strani ključ ne mora biti jedinstven.

Pravilo referencijskog integriteta je jedno od osnovnih pravila zaštite integriteta podataka u bazi i glasi:

Strani, vanjski ključ u povezanoj tablici mora odgovarati primarnom ključu osnovne tablice, te ukoliko se ne koristi kaskadni update ili kaskadni delete vrijedi:

- ne može se mijenjati vrijednost primarnog ključa ako postoji povezani slog u drugoj tablici;
- ne može se izbrisati slog u tablici primarnog ključa ako postoji povezani slog u drugoj tablici;
- ne može se unijeti vrijednost vanjskog ključa prije nego se unese ista vrijednost kao primarni ključ u povezanoj tablici



Relacije kojima su međusobno povezane tablice u relacijskoj bazi podataka mogu biti:

- 1:1 (svakom slogu iz prve tablice odgovara točno jedan slog iz druge tablice)
- 1:N (svakom slogu iz prve tablice odgovara više slogova iz druge tablice)
- M:N (svakom slogu prve tablice odgovara više slogova iz druge tablice, ali vrijedi i obratno. Ova se veza u pravilu realizira pomoću treće tablice vezama 1:N)

Komunikacija korisnika, odnosno aplikacijskog programa i DBMS-a odvija se pomoću SQL jezika upita.

Slika 2. Veza glavnog i vanjskog ključa.

10.3. Baze podataka i Python

SQLite3 se može integrirati u Python koristeći **SQLite3 modul** koji dolazi s Pythonom.



Osnovne naredbe u radu s bazama pokazat ćemo na primjeru baze *MojaBaza* koja će sadržavati dvije tablice **tablica: program** (id, naziv) i **tablicu razred** (id, odjel, program_id).

Funkcija	Objašnjenje
connect()	Kreira objekt Connection koji predstavlja bazu podataka. U primjeru <code>veza=sqlite3.connect ('MojaBaza.db')</code> odredili smo da se podatci pohranjuju u bazu pod nazivom <i>MojaBaza.db</i> .
cursor()	Stvara objekt kursor za interakciju s bazom podataka.
execute()	Omogućuje izvođenje SQL naredbi.
commit()	Zapisuje podatke u bazu podataka.
close()	Zatvara, prekida vezu s bazom podataka. Treba paziti da su sve izmjene prethodno spremljene u bazu s funkcijom <code>commit()</code> .
fetchall()	Dohvaća retke rezultata upita i sprema ih u listu

Tablica 1. Osnovne SQLite funkcije

10. Baze podataka

Kreiranje baze podataka

Ako se pokušamo povezati s datotekom baze podataka SQLite koja ne postoji, ona će se automatski kreirati.

Connection,
connect()

Kako bismo kreirali bazu podataka, prvo moramo kreirati objekt *Connection* (koji predstavlja bazu podataka) pomoću funkcije **connect ()** modula SQLite3 te zatim definirati kurzor koji će nam služiti za interakciju s bazom podataka.

Sljedeće naredbe stvaraju novu bazu podataka MojaBaza.db.

```
import sqlite3  
veza = sqlite3.connect('MojaBaza.db')  
kursor = veza.cursor()  
veza.close()
```

Tip podataka u Python-u	Tip podataka u SQLite
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

Ako želimo da baza bude kreirana u radnoj memoriji trebamo koristiti ključnu riječ :memory: umjesto imena datoteke.

Prije nego krenemo u kreiranje tablica, upoznat ćemo se s tipovima podataka koje SQLite podržava: NULL, INTEGER, REAL, TEXT, BLOB.

Konverzija tipova podataka koje podržava Python u tip podatka koji podržava SQLite prikazana je tablicom:

Tablica 2. Konverzija tipova

Naredba	Objašnjenje
CREATE TABLE ime_tablice ()	Kreiranje tablice.
DROP TABLE ime_tablice	Brisanje tablice.
INSERT INTO ime_tablice (atributi) VALUES (vrijednosti)	Umetanje podataka u tablicu. Omogućuje upis jednog ili više redova u tablicu s pripadnim vrijednostima.
UPDATE ime_tablice SET atribut1 = nova_vrijednost1, atribut2 = nova_vrijednost2 WHERE uvjet	Ažuriranje postojećih redaka u tablici.
DELETE FROM ime_tablice WHERE uvjet	Uklanjanje redaka iz tablice.
SELECT	Prikaz podataka iz jedne ili više tablica.
SELECT popis_atributa FROM ime_tablice	Prikaz navedenih atributa iz tablice.
SELECT * FROM ime_tablice	Prikaz svih atributa iz navedene tablice.
SELECT DISTINCT popis_atributa FROM ime_tablice	Prikaz jedinstvenih (bez ponavljanja) redaka u rezultatu pretraživanja.

Tablica 3. SQLite naredbe (nastavak na sledećoj stranici)

10. Baze podataka

Naredba	Objašnjenje
<pre>SELECT popis_atributa FROM ime_tablice ORDER BY atribut1 ASC, atribut2 DESC</pre>	Prikaz sortiranih atributa zadane tablice. Sortirati se može po jednom ili više atributa. Kad je navedeno više atributa sortira se redom kako su navedeni. Ako je naveden parametar ASC poredak će biti od uzlazan (od manjeg prema većem), a ako je parametar DESC sortirati će silazno, odnosno od većeg prema manjem.
<pre>SELECT popis_atributa FROM ime_tablice WHERE uvjet</pre>	Prikaz atributa iz tablice koji zadovoljavaju zadani uvjet (jednostavan ili složen). Pri formiranju uvjeta mogu se koristiti relacijski operatori (<, <=, =, >, >=, <> ili !=) i logički operatori (ALL, AND, ANY, BETWEEN, EXISTS, IN, LIKE, NOT, OR).

Kreiranje tablica

Za stvaranje tablice koristimo naredbu CREATE TABLE u kurzoru s funkcijom za izvršavanje (execute).

CREATE TABLE

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = veza.cursor()

kursor.execute("""CREATE TABLE IF NOT EXISTS program (
    id integer PRIMARY KEY,
    naziv text NOT NULL)""")

kursor.execute("""CREATE TABLE IF NOT EXISTS razred (
    id integer PRIMARY KEY,
    odjel text NOT NULL,
    program_id integer not null,
    FOREIGN KEY (program_id) REFERENCES program (id))""")

veza.close()
```

Važno je primijetiti da su SQL naredbe u naredbi kursor.execute() stringovi, te ih je moguće definirati posebno.

Unos podataka u tablicu

Unos podataka vršimo naredbom INSERT INTO, te zatim zapisivanjem podataka u bazu funkcijom commit.

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = veza.cursor()

kursor.execute("""INSERT INTO program
VALUES (1, "Opća gimnazija")
""")
```

10. Baze podataka

```
VALUES (,"Prirodoslovno-matematička gimnazija")
"""
veza.commit()
veza.close()
```

Također je moguće zapisati više podataka odjednom definiranjem liste u Pythonu i korištenjem funkcije `executemany` umjesto `execute`. Za formatiranje stringa koristimo znak `?`.

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = veza.cursor()

razredi=[(1,"1.a",2),(2,"1.b",2),(3,"1.c",1),(4,"1.d",1),
         (5,"2.a",2),(6,"2.b",2),(7,"2.c",1),(8,"2.d",1)]
kursor.executemany("INSERT INTO razred VALUES (?,?,?)", razredi)
veza.commit()
veza.close()
```

SQLite Viewer

Sadržaj pojedinih tablica baze moguće je pogledati pomoću online SQLite preglednika [SQLite Viewer](https://inloop.github.io/sqlite-viewer/) (inloop.github.io/sqlite-viewer/).

Dovoljno je odvući bazu s diska u za to predviđeni prostor, te u padajućem izborniku odabrati tablicu čiji sadržaj želimo vidjeti.

The screenshot shows the SQLite Viewer interface. At the top, there's a logo and the text "SQLite Viewer" followed by "view sqlite file online". Below that is a blue header bar with the text "Drop file here to load content or click on this box to open file dialog.". Underneath is a table titled "razred" with 8 rows of data. The columns are labeled "id", "name", and "program_id". The data is as follows:

id	name	program_id
1	1.a	2
2	1.b	2
3	1.c	1
4	1.d	1
5	2.a	2
6	2.b	2
7	2.c	1
8	2.d	1

At the bottom right of the viewer window, there's a "Execute" button. The footer of the window contains the text "© 2010 - inloop Novi Sad, inloop.eu" and "SQLite viewer 1.0".

Slika 3. SQLite preglednik

Promjena i brisanje podataka

Promjena i brisanje podataka unutar tablice mogu se jednostavno napraviti. Naredbama `UPDATE` i `DELETE` mijenjamo, odnosno brišemo podatke. Naredba `WHERE` određuje koje podatke mijenjamo ili brišemo, a naredbom `SET` radimo promjenu.

10. Baze podataka

U sljedećem ćemo primjeru preimenovati razred 2.d u 2.e.

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = veza.cursor()
sql = """
UPDATE razred
SET odjel = "2.e"
WHERE odjel = "2.d"
"""

kursor.execute(sql)
veza.commit()
veza.close()
```

Sljedeći primjer pokazuje kako možemo obrisati određeni redak tablice.

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = veza.cursor()
sql = """
DELETE FROM razred
WHERE odjel = "2.e"
"""

kursor.execute(sql)
veza.commit()
veza.close()
```

U zadnjim dvama primjerima, prvo smo definirali string s naredbama te ih onda proveli.

Naredba WHERE

Unutar naredbi SELECT, UPDATE i DELETE možemo upotrijebiti naredbu WHERE za određivanje uvjeta pretraživanja redaka. Npr.

```
SELECT
    popis_atributa
FROM
    ime_tablice
WHERE
    uvjet
```

Uvjet pretraživanja u naredbi WHERE ima sintaksu oblika `izraz 1 uvjetni_operator izraz 2`, gdje uvjetni operator provjerava odnos između `izraz 1` i `izraz 2`. Uvjetni operatori mogu biti relacijski i logički.

10. Baze podataka

Primjeri:

WHERE

```
atribut1 = 5  
atribut2 IN (1, 2, 3)  
atribut3 LIKE 'An%'  
atribut4 BETWEEN 1 AND 15
```

Uvjetni operator	Objašnjenje
=	jednako
!= ili < >	različito
<	manje
<=	manje ili jednako
>	veće
>=	veće ili jednako
ALL	ISTINA (1) ako su svi izrazi ISTINA (1)
AND	ISTINA (1) ako su oba izraza ISTINA (1), odnosno LAŽ(0) ako je jedan izraz LAŽ (0)
ANY	ISTINA (1) ako je barem jedan izraz ISTINA (1)
BETWEEN	ISTINA (1) ako je vrijednost unutar navedenog raspona
EXISTS	ISTINA (1) ako dodatni upit nije prazan
IN	ISTINA (1) ako se vrijednost nalazi u listi vrijednosti
LIKE	ISTINA (1) ako vrijednost odgovara obrascu
NOT	suprotna vrijednost od navedenog
OR	ISTINA (1) ako je bilo koji izraz ISTINA (1)

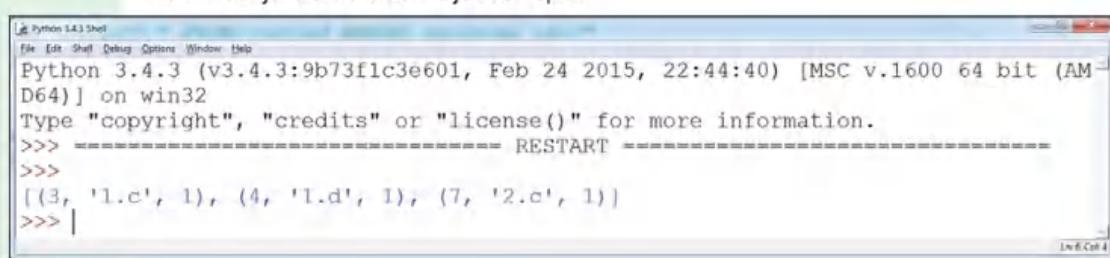
Tablica 4.
WHERE
operatori

Osnovni SQL upiti

Ako želimo prikazati sve razrede Opće gimnazije možemo to učiniti sljedećim upitom:

```
import sqlite3  
veza = sqlite3.connect('MojaBaza.db')  
kurzor = veza.cursor()  
sql = "SELECT * FROM razred WHERE program_id=?"  
kurzor.execute(sql, [(1)])  
print (kurzor.fetchall())  
veza.close()
```

Po izvršenju bismo dobili sljedeći ispis.



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM  
D64)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> ===== RESTART =====  
>>>  
[(3, 'l.c', 1), (4, 'l.d', 1), (7, '2.c', 1)]  
>>> |
```

Slika 4. Prikaz
razreda

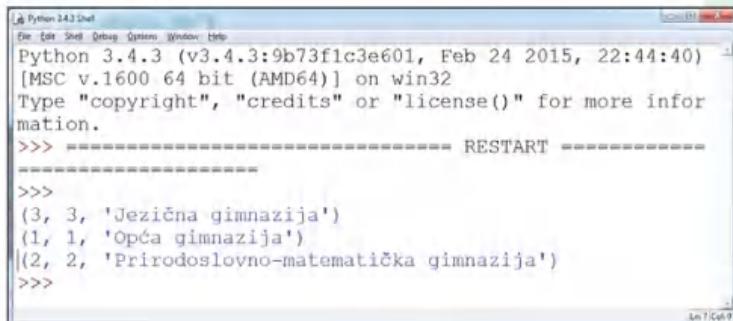
10. Baze podataka

U ovom primjeru upotrebom naredbe SELECT iz tablice *razred* odabrali smo **sve** attribute odnosno stupce (simbol *) gdje je program_id=1. Nakon toga funkcijom *fetchall* izdvajamo sve rezultate i ispisujemo ih.

Osim naredbe *fetchall* postoji i naredba *fetchone* koja dohvata samo jedan rezultat upita. Ako želimo ispisati cijelu tablicu *program* abecedno sortiranu po nazivu iskoristit ćemo sljedeći programski kod.

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = vezा.cursor()
sql = "SELECT rowid, * FROM program ORDER BY naziv"
for i in kursor.execute(sql):
    print (i)
vezа.close()
```

On će rezultirati ovakvim ispisom:



The screenshot shows the Python 3.4.3 Shell window. It displays the command line input and output. The command `print (i)` is run, followed by `vezа.close()`. The output shows three rows from the 'program' table, ordered by 'naziv': (3, 3, 'Jezična gimnazija'), (1, 1, 'Opća gimnazija'), and (2, 2, 'Prirodoslovno-matematička gimnazija').

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> -----
>>> (3, 3, 'Jezična gimnazija')
(1, 1, 'Opća gimnazija')
(2, 2, 'Prirodoslovno-matematička gimnazija')
>>>
```

fetchall

fetchone

Slika 5. Abecedno sortirana tablica program

Naredba ORDER BY

Kod SQLite modula redoslijed pohranjivanja podataka u tablice nije strogo definiran tako da redovi u tablici mogu, ali i ne moraju biti u onom redoslijedu u kojem su umetani. Stoga prilikom upotrebe naredbe SELECT redoslijed redaka u skupu rezultata nije određen. Ukoliko želimo da nam rezultat bude sortiran, silazno ili uzlazno upotrijebit ćemo ORDER BY naredbu. Npr.

```
SELECT
    popis_atributa
FROM
    ime_tablice
ORDER BY
    atribut1 ASC,
    atribut2 DESC
```

Naredba ORDER BY dolazi nakon ključne riječi FROM i omogućuje sortiranje rezultata na temelju jednog ili više atributa u uzlaznom ili silaznom redoslijedu. Atribut po kojem želimo sortirati retke rezultata navodimo nakon ORDER BY. Ako nakon atributa navedemo ASC, rezultat će biti uzlazno sortiran (od manjeg prema većem), a silazno (od većeg prema manjem) ako navedemo DESC. Ukoliko ne definiramo način sortiranja, po-drazumijeva se da je to ASC.

Ukoliko želimo sortirati skup rezultata po više atributa (stupaca), tada ih trebamo odvojiti zarezom (,), a rezultati će se sortirati redom po atributima kako su navedeni. Skup rezultata možemo sortirati i uz pomoću atributa koji se ne pojavljuje na popisu za prikaz u naredbi SELECT.

10. Baze podataka

Primjer: (vezan uz bazu MojaBaza.db)

```
sql = """
SELECT
odjel from razred
ORDER BY
program_id
"""

kurzor.execute(sql)
print (kurzor.fetchall())
```

Prilikom popunjavanja tablice postoji mogućnost da nam neke vrijednosti nisu trenutno poznate. U tom se slučaju kao oznaka da podatak nedostaje koristi vrijednost NULL. NULL ne možemo usporediti s nekom drugom vrijednošću. NULL se ne može usporediti ni sa sobom, odnosno usporedba NULL=NULL daje rezultat laž.

Kad sortiranja, smatra se da je NULL manji od bilo koje druge vrijednosti, tj. pojavit će se na početku skupa rezultata ako koristimo ASC ili na kraju skupa rezultata kada koristimo DESC.

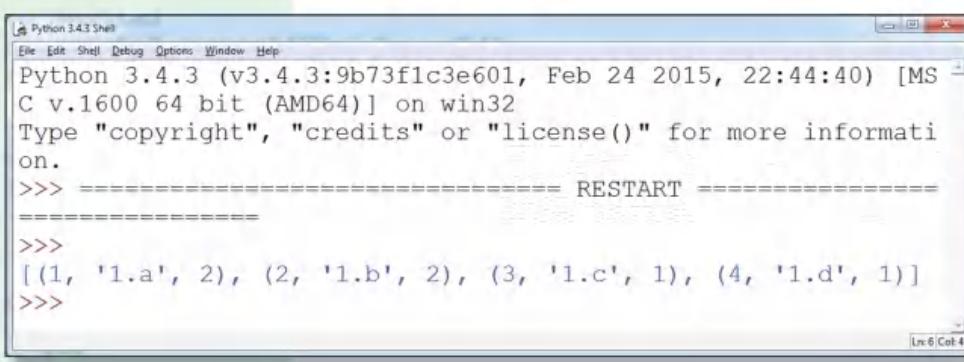
Ako trebamo naći podatke kojima ne znamo puni naziv ili su slični, koristit ćemo upit LIKE. Koristimo se znakom % koji predstavlja dio upita koji nije kriterij po kojem pretražujemo, tj. dio koji je različit u upitim koje ćemo dobiti kao povrat. U našem primjeru to znači pronaći podatke koji su vezani uz prvi (1.) razred. Osim toga, upitima %o% ili %d, naći ćemo podatke koji u sebi sadrže slovo o, odnosno završavaju slovom d.

```
import sqlite3
veza = sqlite3.connect('MojaBaza.db')
kursor = veza.cursor()

sql = """
SELECT * FROM razred
WHERE odjel LIKE "1.%"
"""

kursor.execute(sql)
print (kursor.fetchall())

veza.close()
```



The screenshot shows the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the following text:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
[(1, '1.a', 2), (2, '1.b', 2), (3, '1.c', 1), (4, '1.d', 1)]
>>>
```

Nakon izvršenja gornjih naredbi ispis će izgledati kao na slici:

Slika 6. Uporaba upita LIKE

10. Baze podataka

Prijedlog aktivnosti

Ovo je bio kratak pregled mogućnosti obrade podataka koje pružaju Python i SQLite. O ostalim detaljima možete se upoznati radeći na projektnom zadatku koji može povezivati nekoliko predmeta.

Rasporedite se u skupine po četvero. Zadatak svake grupe je izraditi bazu podataka prema zadanoj temi ili je sami predložite.

Ideje za neke teme su:

- napraviti bazu podataka o karakterističnim biljkama zavičaja
- napraviti bazu podataka o znamenitim bivšim učenicima vaše škole
- napraviti bazu podataka o znamenitim sugrađanima
- napraviti bazu podataka o olimpijcima iz područja znanosti (Matematika, Fizika, Kemija, Informatika, Geografija)

10.4. Baze koje život znače

Kao što smo napisali u uvodu, dobar dizajn baze koji omogućuje brz i pouzdan pristup podacima pretpostavka je na kojoj možemo graditi uspjeh. Sustavi poput Facebooka svakodnevno prikupljaju ogromne količine podataka, a korisnici od kojih ti podaci dolaze očekuju da će im svaka njihova objava i fotografija biti dostupne istog trena. Porastom vremena pristupa podacima raste i frustracija korisnika koji se onda prirodno okreće boljim i učinkovitijim rješenjima. Za takve projekte dobra baza temelj je poslovнog uspjeha.

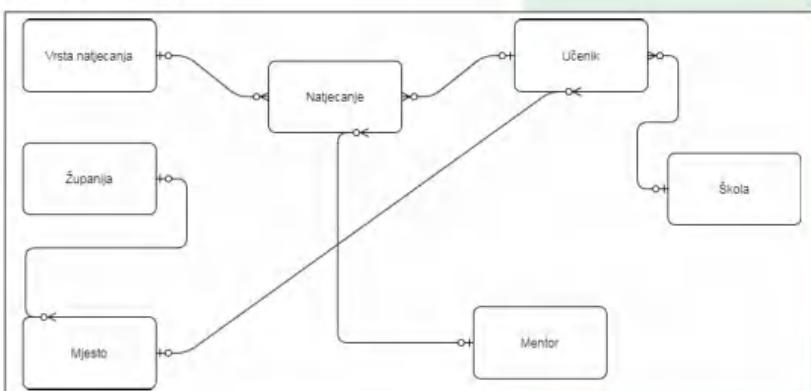
Kvalitetan dizajn baze podataka ključan je preduvjet za ostvarenje cilja zbog kojeg razvijamo bazu. Naknadna dodavanja tablica i relacija napravit će više problema nego koristi. Počašćimo na jednom primjeru kako možemo dizajnirati bazu. Taj primjer bit će akademske naravi jer baze s kojima rade veliki sustavi izvan su našeg trenutnog obzora promatranja.

Problem

Modelirajmo bazu podataka koja će nam omogućiti prikupljanje i analizu podataka o učenicima i njihovim nastupima na natjecanjima iz informatike.

Opis rješenja

Prvi korak u modeliranju baze je definiranje svih tablica i veza među tablicama. Bolje je uložiti više vremena u osmišljavanje baze nego poslije raditi nadogradnje i korekcije. Baza podataka bi trebala sadržavati tablice: Učenik, Vrsta natjecanja, Županija, Mjesto, Škola, Mentor, Natjecanje. Veze među tablicama možemo prikazati sljedećim dijagramom.



Za svaku tablicu trebamo definirati podatke koje će sadržavati. Krenimo redom.

Slika 7. Koncept veza među tablicama

10. Baze podataka

Tablica **Učenik** – sadrži podatke koji se odnose na učenika i koji ne ovise o natjecanjima na kojima će sudjelovati.

Tablica Učenik						
ID_ucenik	ime	prezime	OIB	datumR	ID_mjesto	ID_skola

Tablica **Mjesto** – sadrži relevantne i nepromjenjive podatke o mjestu iz kojeg dolazi učenik.

Tablica Mjesto			
ID_mjesto	Mjesto	PBroj	ID_zupanija

Tablica **Mentor** – sadrži podatke o nastavniku koji na natjecanjima sudjeluje kao mentor.

Tablica Mentor				
ID_mentor	ime	prezime	OIB	ID_skola

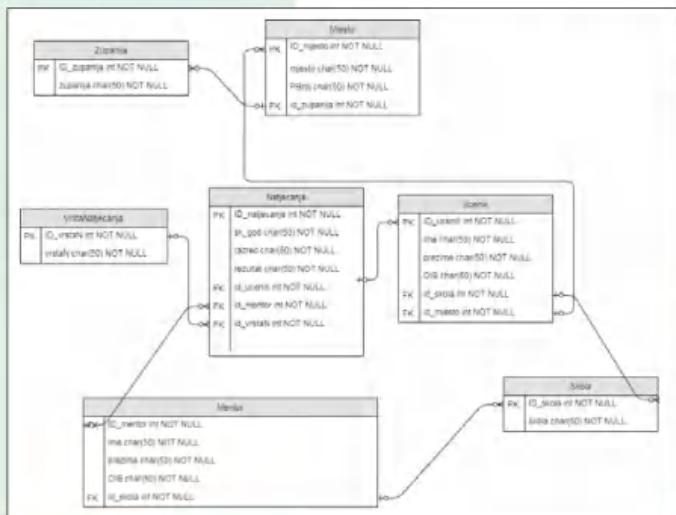
Tablica **Natjecanje** – centralna tablica baze u kojoj se vode podaci o svim natjecanjima. Ona povezuje sve ostale tablice i većina atributa (polja) ove tablice predstavljaju vanjske ključeve ostalih tablica.

Tablica Natjecanje						
ID_natjecanje	škol_god	razred	ID_ucenik	ID_vrstaN	ID_mentor	rezultat

Preostale tri tablice opisuju županiju, školu i vrstu natjecanja.

Tablica Županija		Tablica Škola		Tablica Vrsta natjecanje	
ID_zupanija	zupanija	ID_skola	skola	ID_vrstaN	vrstaN

Za izradu grafa korišten je alat **draw.io** na stranici www.draw.io.



Osmislimo relacije među tablicama. Povezane tablice sadrže iste vrijednosti. U jednoj tablici u obliku primarnog ključa a u drugoj tablici u obliku vanjskog – stranog ključa.

Veze među tablicama prikazane su na slici 8.

Nakon što smo koncipirali relacije, sada je trenutak da i fizički kreiramo bazu.

Programski kod za kreiranje baze možete preuzeti iz datoteke: **3g1Okreiranje.py**.

Pokažimo na nekoliko problema što i kako možemo raditi s bazom.

Problem 1.

Upiši u tablice baze podatka učenike škole koji su sudjelovali na natjecanjima. Napiši upit kojim ćeš doći do traženog popisa.

10. Baze podataka

Tablica Učenik

id_ucenik	ime	prezime	OIB	datumR	id_skola	id_mjesto
1	Erik	Erić	12387578901	13.3.2002.	1	1
2	Enio	Prvić	08763253574	27.02.2004.	1	1
3	Matan	Metiković	70000035418	18.11.2003.	5	3
4	Koko	Limač	55411152546	11.11.2004.	5	3
5	Borko	Borić	39898998871	12.9.2003.	2	2
6	Miro	Mirić	00033257981	07.08.2002.	2	2

Tablica Škola

id_skola	skola
1	Gimnazija Požega
2	Gimnazija Pula
3	Tehnička škola Rudera Boškovića
4	Gimnazija Lucijana Vranjanina Zagreb
5	XV. gimnazija Zagreb
6	Prva gimnazija Varaždin
7	Gimnazija Beli Manastir

Problem 2.

Napiši upit koji će iz baze ispisati sve natjecatelje kategorije *Algoritmi*.

Problem 3.

Proširi gornji upit tako da uz učenike budu prikazani i njihovi mentori.

Tablica Mentor

id_mentor	ime	prezime	OIB	id_skola
1	Ado	Trenč	12345678901	5
2	Miro	Skarfić	01598763574	2
3	Pino	Blekšmit	78960235418	1
4	Karol	Kovačić	55487963852	6
5	Bero	Beg	33251498871	3
6	Pirol	Mun	31000257981	5

Problem 4.

Pronađi za koliko različitih natjecanja su trenutno upisani podaci u bazi.

Problem 5.

Pronađi školu s najviše osvojenih prvih mjesta.

 Rješenja problema možete preuzeti s mrežne stranice uz udžbenik: [3g10rjesenja.py](#).

11. Izrađivanje i objavljivanje mrežnih stranica

Svakodnevno pregledavamo sadržaj na internetu. Svaka slika, videozapis ili tekstualni sadržaj koji vidimo nalazi se na mrežnoj stranici. Pregledavanje mrežne stranice jednostavan je postupak, ali što ako ne želimo samo pregledavati sadržaj koji je netko drugi izradio već i mi sami želimo stvarati svoje mrežne stranice na internetu? Upoznajmo najprije važne pojmove, pa krenimo u izradu svoje mrežne stranice.

Mrežna stranica

Poveznica

URL adresa

Mrežni preglednik

Mrežnu stranicu nazivamo još i web-stranica. Mrežne stranice nisu zasebni dijelovi na internetu, jednako tako kao što i stranice u knjizi koju čitaš za lektiru nisu zasebni listovi, već čine jednu veću cjelinu. Mrežne su stranice hipertekstualni dokumenti međusobno povezani **poveznicama** (linkovima) na lokaciju koja objedinjuje više međusobno povezanih mrežnih stranica u mrežno mjesto (engl. site).



Slika 2. Poveznice vode s jedne stranice na drugu stranicu (putanja označena crvenim strelicama)

Primjer mrežnog mjeseta mrežne su stranice tvoje škole. Do mrežne stranice doći ćemo tako da u mrežni preglednik upišemo njezinu **URL adresu** (npr. adresu Portala za škole: www.sko-le.hr). **Mrežni preglednik** je program

kojim pregledavamo mrežne stranice (npr. Google Chrome, Microsoft Edge, Mozilla Firefox ili Opera).



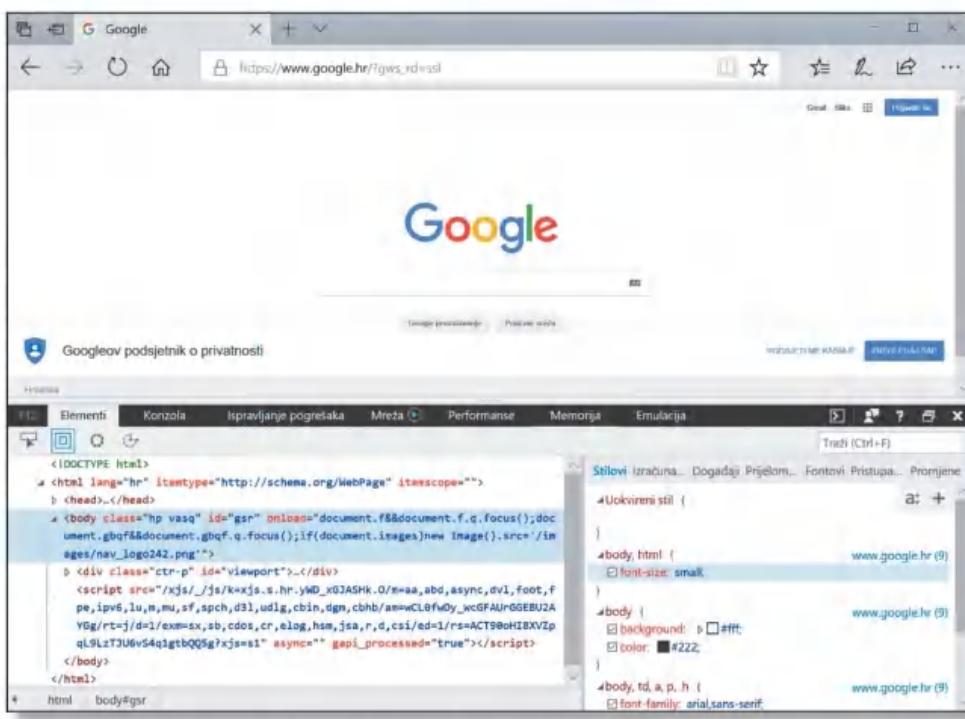
Slika 1. Ikone mrežnih preglednika: Google Chrome, Microsoft Edge, Mozilla Firefox i Opera

Vježba 1.

1. U mrežnom pregledniku otvori početnu stranicu Google-ove tražilice (www.google.hr).
2. Na tipkovnici pritisni funkciju tipku F12 za pregled izvornog kôda stranice.

Prozor mrežnog preglednika podijelio se na dva dijela – u jednom dijelu vidimo stranicu onako kako ona zaista izgleda, a u drugom vidimo njezin kôd. Klikom na strelice unutar kôda koje su usmjerene udesno otvorit će se dodatni kôd. Što primjećuješ? Koliko je kôda potrebno da bi se prikazala jedna stranica?

11. Izradivanje i objavljivanje mrežnih stranica



Slika 3. Mrežna stranica i HTML kôd početne stranice Google

Postoje dva načina na koja možemo izraditi mrežnu stranicu:

1. pisanjem HTML označa i teksta u tekstualnom uređivaču i
 2. oblikovanjem elemenata stranice pomoću vizualnog HTML uređivača.
- Za početak upoznat ćemo HTML jezik i njegove osnovne mogućnosti.

HTML jezik i vizualni
HTML uređivači

11.1. HTML jezik

Već smo spomenuli da je HTML jezik jednostavan za upotrebu i lako se pamti. U ovom poglavlju objasnit ćemo osnovne mogućnosti HTML-a i pokazati kako iskodirati jednostavan HTML dokument.

Kao što ste vidjeli u prethodnoj vježbi, u pozadini svake mrežne stranice nalazi se njezin kôd. Taj kôd kojim je opisana mrežna stranica naziva se **HTML jezik**.

Da bismo se koristili HTML jezikom, ne trebamo instalirati nikakav posebni program, već nam obični tekstualni uređivač (npr. Blok za pisanje) može poslužiti za kreiranje HTML dokumenata.

HTML nije programski jezik. On ne omogućava izvršavanje računskih zadataka kao što je primjerice zbrajanje dva broja. Njegova je uloga da opiše mrežnom pregledniku na koji način treba prikazati mrežnu stranicu tj. hipertekstualni dokument. Zato se i zove prezentacijski jezik.

Postoje i online uređivači HTML kôda, poput ovog na mrežnoj adresi html-online.com/editor/.

11. Izrađivanje i objavljivanje mrežnih stranica

Tag – HTML oznaka

HTML oznake

HTML jezik sastoji se od osnovnih građevnih elemenata koje nazivamo **HTML tagovi** ili **oznake**. Većina oznaka dolazi u paru: prvi element predstavlja početnu, a drugi element završnu oznaku, tj. uključuju i isključuju određeno oblikovanje. Postoji i mali broj oznaka koje su samo-zatvarajuće, tj. nemaju svoju završnu oznaku.

Primjer HTML oznake za naslov stranice:

```
<title>Početna stranica</title>
```

```
<html>
  <head>
    < title > </ title >
  </head>

  <body>
    ...
  </body>
</html>
```

Slika 4. Struktura HTML dokumenta

Struktura HTML-a

Svaki HTML dokument započinje oznakom **<html>** koja ima svoju završnu oznaku **</html>**. Te oznake govore mrežnom pregledniku gdje počinje, a gdje završava mrežna stranica i sve što se nalazi unutar tih dviju oznaka bit će prikazano na mrežnoj stranici.

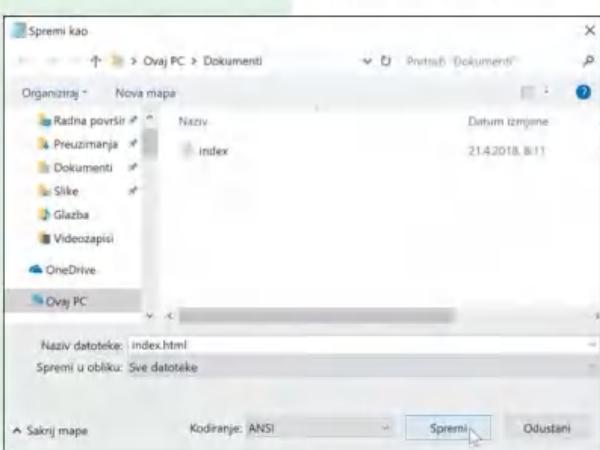
Sljedeća je važna oznaka **<head> </head>** koja predstavlja zaglavlj mrežne stranice u kojoj su definirana stilska obilježja stranice (npr. jezik, način kodiranja i slično). Unutar zaglavlja nalazi se i oznaka **<title> </title>** unutar koje pišemo naslov mrežne stranice, a koji se prikazuje u naslovnoj traci mrežnog preglednika.

Oznaka **<body> </body>** je oznaka unutar koje smještamo sadržaj mrežne stranice – tekst, slike, videozapise, tablice... Unutar te oznake pišu se sve ostale oznake nužne za uređivanje teksta, umetanje poveznica ili umetanje multimedijiskog sadržaja.

Kako izraditi HTML dokument?

Pokrenite program *Blok za pisanje*. Pronaći ćete ga tako da otvorite Start te u prozor za pretragu upišete ime programa.

Istraži na internetu koje su preostale oznake – tagovi HTML jezika i kako se njima oblikuju elementi mrežnih stranica.



Vježba 2.

1. U Blok za pisanje prepisi sljedeći HTML kôd.

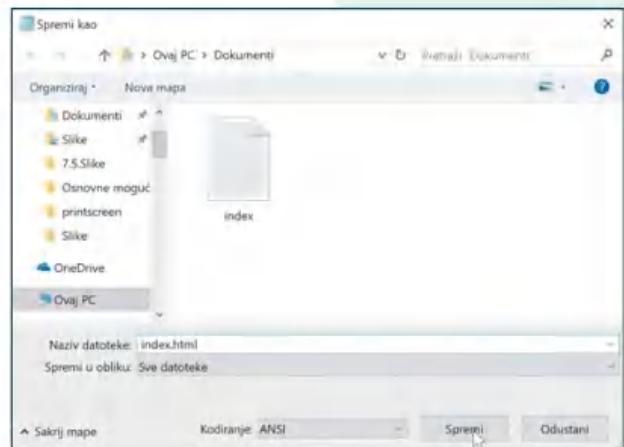
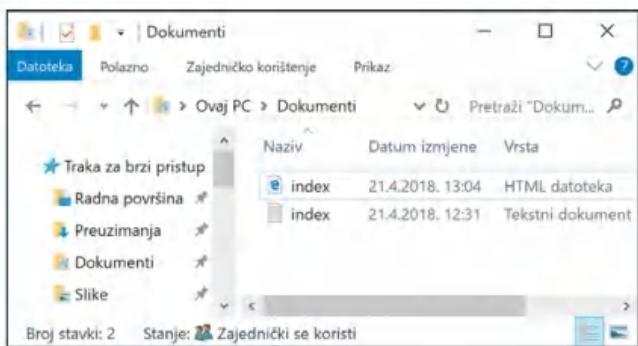
```
<html>
  <head>
    <title>Početna stranica</title>
  </head>
  <body>
    Moja mrežna stranica napisana u HTML jeziku.
  </body>
</html>
```

Slika 5. Spremanje datoteke u tekstualnom obliku

11. Izrađivanje i objavljivanje mrežnih stranica

Kada mrežne stranice izrađujemo pomoću HTML jezika, dokument koji spremimo pod imenom *index.html* bit će prva stranica koju će mrežni preglednik otvoriti, tzv. *home page*.

2. Nakon odabira opcije *Spremi kao*, u polje *Naziv datoteke* upiši *index.html* (ili *.htm*), klikni na *OK* ili pritisni *Enter*.
3. Pregledaj svoju mrežnu stranicu tako da je otvořiš dvostrukim klikom na njezinu ikonu (imat će ikonu mrežnog preglednika) u mapi u kojoj je spremljena.



Slika 6. Spremanje datoteke u obliku HTML-a

Slika 7. Ikone tekstualnog i HTML dokumenta

11.2. Osnovne mogućnosti HTML jezika

Osim osnovnih oznaka kojima HTML jezik opisuje mrežnu stranicu, a koje ste upoznali u prethodnoj lekciji, postoji i mnoštvo drugih oznaka kojima umećemo slike, poveznice, mijenjamo boju pozadine ili ugrađujemo multimediju. Mrežna stranica bit će zanimljivija ako ima elemente multimedije.

Nadogradit ćemo svoju mrežnu stranicu koju smo prethodno izradili pa otvoriti tekstualni dokument *index.txt*.

Pozadinska boja stranice

Stranica će biti zanimljivija i živopisnija ako sadržava element poput boje. Možemo zadati boju teksta i boju pozadine.

Vježba 3.

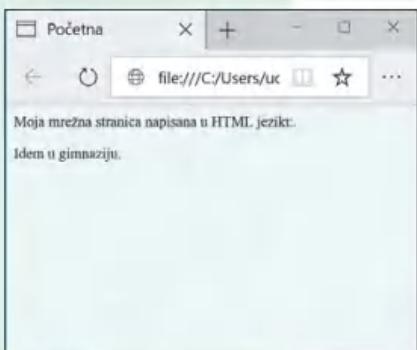
1. Svojoj mrežnoj stranici dodat ćemo pozadinsku boju. Boja se dodaje tako da se unutar početne oznake `<body>` upiše `bgcolor=""`, a unutar navodnika upisuje se ime boje na engleskom jeziku. Umjesto *azure*, možeš upisati neku drugu boju (npr. *orange*, *green*, *blue*, *purple*, *pink*, *yellow*).

```
<body bgcolor="azure">
```

Oznakom **background** koja dolazi unutar oznake `<body>` možeš dodati sliku na pozadinu stranice. Kôd će izgledati ovako `<body background="pozadina.jpg">`. Važno je odabrati sliku veće razlučivosti (primjerice razlučivosti zaslona) poštujući autorska prava i spremiti je pod nekim prepoznatljivim imenom u istu mapu gdje je spremljena stranica te znati koji je tip slikovne datoteke (JPG, PNG, GIF...). Detaljno objašnjenje o načinu umetanja slike na pozadinu mrežne stranice pronađi u videozapisu **3g11pozadina.mp4**.

11. Izrađivanje i objavljivanje mrežnih stranica

2. Spremi i tekstualni i HTML dokument *index*.
3. Otvori HTML dokument i pregledaj svoju stranicu u mrežnom pregledniku.



Slika 8. Pozadinska boja stranice

Ako postaviš pozadinsku boju u crnu, vodi računa i o boji slova – u osnovnoj su značajki slova crne boje i moramo koristiti oznaku za promjenu boje slova ako ih želimo vidjeti na crnoj pozadini.

Ako želiš da tvoja mrežna stranica ima pozadinsku boju u nekoj nijansi, tada se umjesto naziva boje upisuje njezin kôd. Primjerice, `<body bgcolor="#80FF00">` obojat će pozadinu u zelenastu boju limete. Pokušaj na

svoju stranicu umetnuti kôd neke boje po izboru, a kôdove možeš pronaći na stranici html-color-codes.info.



Slika 9. HTML grafikon boja

Oznake `<p>` i `
`

Kako nam sav tekst kojeg upišemo ne bi bio nepregledan, koristit ćemo se oznakama za strukturiranje teksta. Oznaka `<p>` govori mrežnom pregledniku da sve što će biti napisano nakon nje postaje novi odlomak. Ako želimo samo tekst smjestiti u novi red (istovjetno pritisku tipke *Enter* na tipkovnici), tada ćemo upotrijebiti oznaku `
`. Oznaka `
` samozatvarajuća je – nema svoju završnu oznaku.

Vježba 4.

1. Dodaj u svoju tekstualnu datoteku *index.txt* elemente kôda iz donjeg primjera.

```
<html>
<head>
<title>Početna stranica</title>
</head>
<body bgcolor="azure">
Moja mrežna stranica napisana u HTML jeziku.<br>
<p>Idem u gimnaziju.</p>
</body>
</html>
```

2. Spremi tekstualnu i HTML datoteku.

3. Pregledaj HTML dokument u mrežnom pregledniku. Koja je razlika između oznake `<p>` i `
`?

HTML kôd	Prikaz u pregledniku
Moja mrežna stranica napisana u HTML jeziku. <p>Idem u gimnaziju.</p>	Moja mrežna stranica napisana u HTML jeziku. Idem u gimnaziju.

11. Izrađivanje i objavljivanje mrežnih stranica

Za bolje razumijevanje kako mrežni preglednik interpretira sadržaj i zašto koristimo oznake `
` i `<p>`, usporedi sljedeća dva primjera:

HTML kôd	Prikaz u pregledniku
Prvi redak. Drugi redak. Kraj.	Prvi Redak. Drugi redak. Kraj.

HTML kôd	Prikaz u pregledniku
Prvi redak. Drugi redak. <p>Kraj.</p>	Prvi Redak. Drugi redak. Kraj.

Umetanje slike u HTML dokument

Da bi naša stranica bila zanimljivija, dodat ćemo sliku. Samozatvarajuća oznaka za umetanje slike u HTML dokument jest ``, odnosno punog oblika: ``.

Ako na mrežnu stranicu ne postavљаш slike čiji si autor, vodi računa o autorskim pravima kad pretražuješ i preuzimaš slike s interneta.

`` - oznaka za umetanje slike

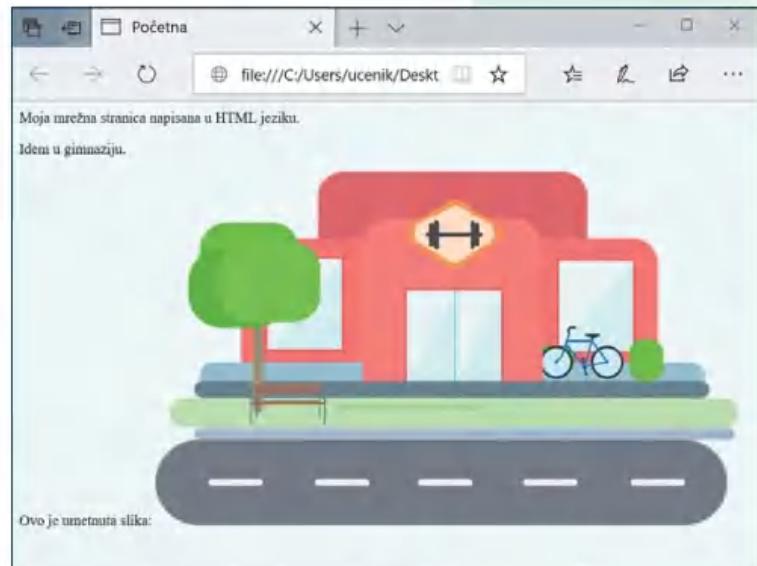
Vježba 5.

Prije nego što započnemo vježbu, preuzmi na svoje računalo datoteku 3g11skola.png i spremi ju u istu mapu gdje se nalazi index.html datoteka.

1. Otvori `index.txt` datoteku. Nadogradji svoj HTML dokument tako da dopišeš dio kôda označen crvenom bojom.

```
<body bgcolor="azure">
Moja mrežna stranica napisana u
HTML jeziku.<br>
<p>Idem u gimnaziju.</p>
Ovo je umetnuta slika:
<br>
</body>
```

2. Spremi tekstualnu i HTML datoteku.
3. Pregledaj HTML dokument u mrežnom pregledniku.



Kod umetanja slika potrebno je voditi računa o veličini slike. Ako je slika prevelika, nećemo je moći čitavu vidjeti u mrežnom pregledniku. Isto tako, treba paziti kojeg je tipa (PNG, JPG, GIF) kako bismo ispravno napisali njezin nastavak.

Slika 10. Slika umetnuta u mrežnu stranicu

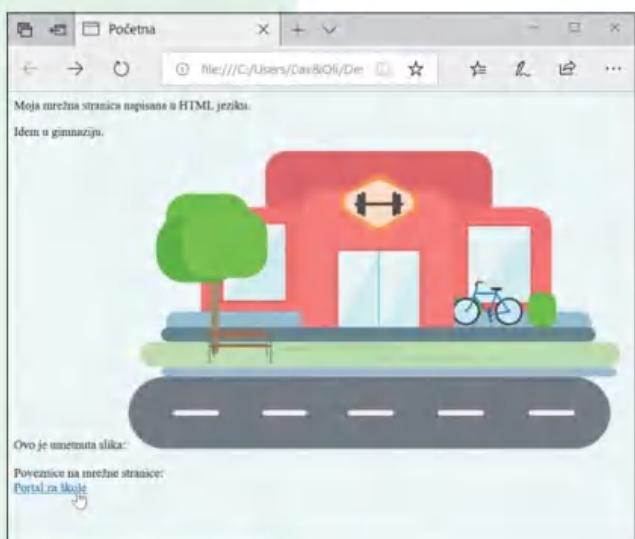
11. Izrađivanje i objavljivanje mrežnih stranica

poveznica (hiperlink,
link, hiperveza)

Oznaka za umetanje poveznice

Već smo spominjali da mrežno mjesto čini više mrežnih stranica, HTML dokumenata, koji imaju zajedničku temu. Oni su međusobno povezani putem poveznica. Na sljedećem primjeru vidjet ćemo kako se u HTML dokument pomoću oznake dodaje poveznica na neku mrežnu stranicu na internetu.

Vježba 6.



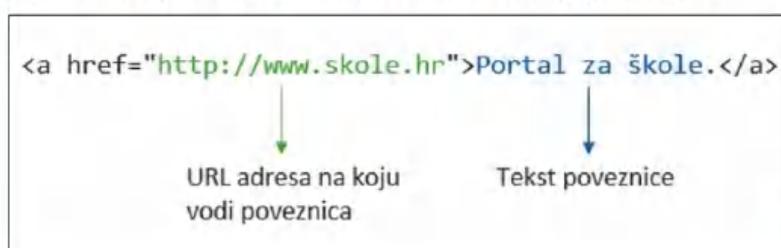
Slika 11. Poveznica na mrežnoj stranici

1. Otvori tekstualni dokument *index.txt* pa ga nadogradi tako da upišeš crveno označeni dio kôda unutar `<body>` oznake.

```
<body bgcolor="azure">  
Moja mrežna stranica napisana u HTML  
jeziku.<br>  
<p>Idem u gimnaziju.</p>  
Ovo je umetnuta slika:  
<br>  
Poveznice na mrežne stranice:<br>  
<a href="http://www.skole.hr/">Portal za  
škole.</a><br>  
</body>
```

2. Spremi tekstualni i HTML dokument i pregledaj svoju mrežnu stranicu. Provjeri vodi li te netom stvorena poveznica na *Portal za škole*.

Dijelovi od kojih se sastoji oznaka `<a>` za umetanje poveznice:



Slika 12. Dijelovi oznake `<a>`

Tvoju početnu stranicu povezali smo s mrežnom stranicom smještenom na drugom računalu u mreži, na domeni skole.hr.

Pogledajmo sada kako je povezati sa stranicom smještenom lokalno, na tvojem računalu.

Vježba 7.

1. Izradi novi dokument u *Blok u pisanje* i kreiraj osnovnu strukturu HTML stranice.
2. Zadaj naslov stranice *Fizika* i izaberi pozadinsku boju. Unutar oznake `<body>` napiši rečenicu: *Fizika je moj omiljeni predmet*.
3. U novi redak umetni sliku 3g11fizika.png.

11. Izrađivanje i objavljivanje mrežnih stranica

4. Dokument nazovi *fizika* i spremi u tekstualnom i HTML obliku u istu mapu gdje se nalazi i dokument *index*.
5. Otvori datoteku *index.txt* i u njoj dodaj oznaku za poveznicu na *fizika.html*. Budući da se dokument nalazi na lokalnom računalu, kod upisivanja URL adrese u oznaku za poveznicu ne trebaš upisati *http://* ili *https://* niti *www*, već samo ime datoteke: *fizika.html*. Tekst poveznice neka bude *Moja fizika*.
6. Spremi preinake u dokumentu *index* u tekstualnom i u HTML obliku. Otvori *index.html* u pregledniku i isprobaj poveznicu.

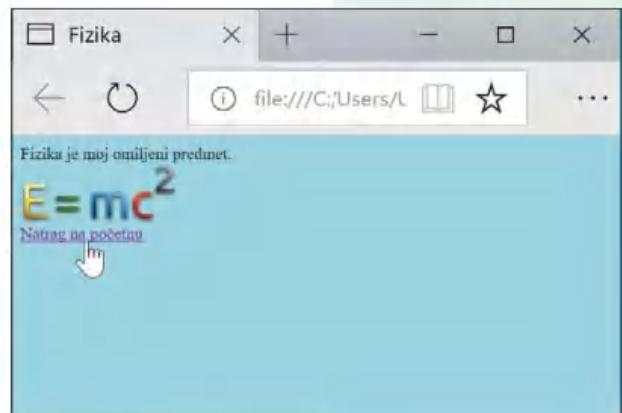
Preporučujemo da sve HTML dokumente spremiš unutar iste mape prije nego što ćeš ih pozivati, jer u protivnom je potrebno pisati putanje do dokumenta koji želiš povezati poveznicom – npr. *file:///C:/Users/Učenik/Dokumenti/HTML/Moja stranica/psi.html*

Vježba 8.

1. U dokumentu *fizika.txt* dodaj poveznicu koja će te vratiti natrag na početnu (*index*) stranicu. Tekst poveznice neka bude *Natrag na početnu*.
2. Spremi tekstualni i HTML dokument i pregledaj svoju mrežnu stranicu. Trebala bi izgledati kao na slici 14.

Oznake za ukošavanje, podebljavanje i bojanje teksta

Kako sav tekst napisan unutar oznake `<body>` ne bi bio monoton i jednoličan, postoje oznake kojima se napisani tekst može istaknuti kako bi se lakše razlikovao i estetski bio privlačniji. U tu svrhu naučit ćemo upotrebljavati tri nove oznake: `` za podebljavanje teksta, `<u>` za podcrtavanje teksta, `<i>` za ukošavanje teksta i `` za bojanje teksta u zadalu boju.



Slika 13. Dokument *fizika.html* otvoren u pregledniku

HTML kôd	Prikaz u pregledniku
<code>Ovo je podebljan tekst.</code>	Ovo je podebljan tekst.
<code><u>Ovo je podcrtan tekst.</u></code>	<u>Ovo je podcrtan tekst.</u>
<code><i>Ovo je ukošen tekst.</i></code>	<i>Ovo je ukošen tekst.</i>
<code>Ovo je crveno obojan tekst.</code>	Ovo je crveno obojan tekst.

Vježba 9.

1. U Bloku za pisanje stvari novu datoteku *kemija.txt* koristeći se oznakama za strukturiranje HTML dokumenta.
2. Stranicu naslovi *Kemija*.

11. Izrađivanje i objavljivanje mrežnih stranica

The screenshot shows an online HTML editor interface. On the left, the HTML code is displayed:

```
<!DOCTYPE html>
<html>
<body>
<p><b>Ovo je podebljan tekst</b></p>
<p><i>Ovo je ukošen tekst</i></p>
<p>Ovo je<sub>subscript</sub> i<br/><sup>superscript</sup></p>
</body>
</html>
```

On the right, the rendered HTML is shown:

Ovo je podebljan tekst
Ovo je ukošen tekst
Ovo je subscript i superscript

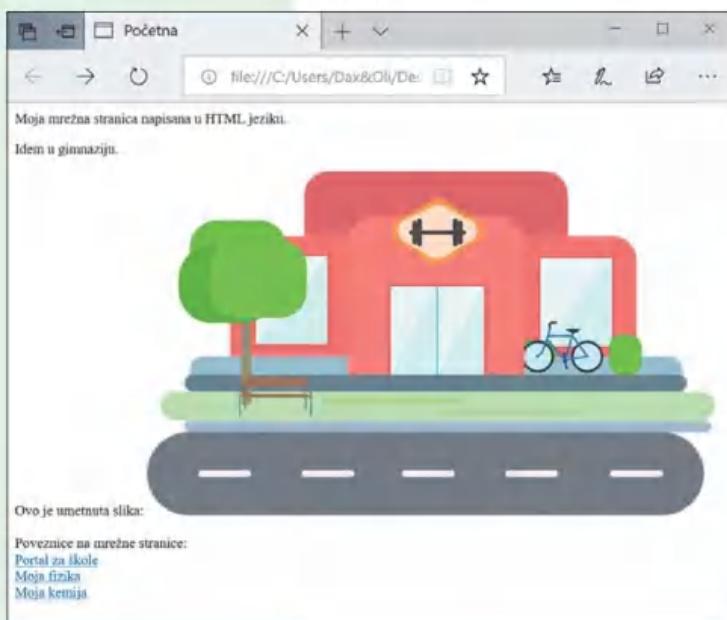
Slika 14. Online editor za isprobavanje HTML oznaka sa interaktivnim sučeljem

Vježba 10.

1. Otvori dokument *kemija.txt*.
2. Ispod napisanih rečenica o kemiji u novi odlomak umetni sliku  **3g11kemija.jpg** (spremi je u mapu s preostalim dokumentima iz dosadašnjeg rada).

3. Unutar oznake `<body>` upiši rečenicu: *Kemija je moj drugi omiljeni predmet*.
4. Ispod te rečenice u novi redak napiši tri rečenice o kemiji, svaku u novom retku, i oblikuj ih oznakama za ukošeno, podebljano, podcrтано и објено (boju odaberisam).
5. Spremi tekstuallnu i HTML datoteku i otvori mrežnu stranicu u pregledniku.

Istraži na internetu koje još oznake za oblikovanje teksta postoje i pokušaj putem mrežne stranice https://www.w3schools.com/html/html_formatting.asp isprobati neke od njih pomoću interaktivnog sučelja.



3. Ispod slike u novi redak dodaj poveznicu koja će voditi na početnu stranicu (*index.html*).
4. Stranici *kemija* dodaj pozadinsku boju po izboru.
5. Spremi tekstuallni i HTML dokument i pregledaj ga u mrežnom pregledniku.
6. Na početnoj stranici dodaj poveznicu na dokument *kemija.html* s tekstrom poveznice: *Moja kemija*.
7. Spremi početnu stranicu u tekstuallnom i HTML obliku.
8. Otvori *index.html* u mrežnom pregledniku i provjeri rade li ispravno poveznice među svim dokumentima.

Slika 15. Prikaz dokumenta *index.html* nakon doda- vanja svih poveznica

Ugradnja multimedijskog sadržaja u HTML dokument

Ugradnja multimedijskog sadržaja u HTML dokument omogućava da mrežne stranice budu zanimljive i interaktivne. Prednost je i u tome što ne moramo zauzimati dodatni prostor na mrežnom poslužitelju postavljanjem multimedije, nego HTML oznakom na svoju stranicu „ugradimo“ taj element.

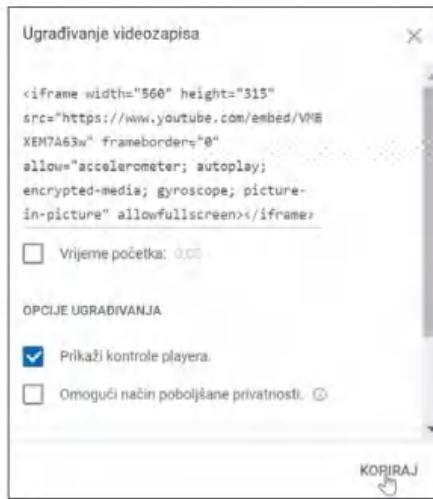
embed - ugradnja sadržaja dostupnog na internetu u vlastitu mrežnu stranicu

11. Izrađivanje i objavljivanje mrežnih stranica

Opciju ugradnje multimedijskog sadržaja isprobat ćemo na primjeru videozapisa sa servisa YouTube.

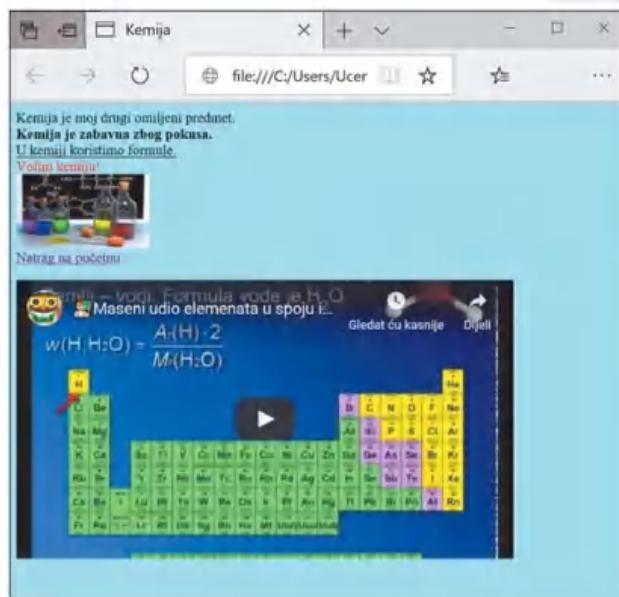
Vježba 11.

1. Otvori dokument *kemija.txt* i ispod povezničce dodaj oznaku za novi odlomak.
2. Pokreni mrežni preglednik, otvori stranicu www.youtube.com, pretraži videozapise s ključnim pojmom *kemija*, odaberis jedan videozapis i pokreni ga.
3. Nakon pokretanja videozapisa, u izborniku ispod odaberis opciju Podijeli te nakon toga Ugradi.
4. Kopiraj kôd koji je prikazan u okviru (slika 16.).



HTML oznaka za ugradnju multimedijskog sadržaja je `<iframe> </iframe>`.

Slika 16. Kôd za ugradnju sadržaja



5. Kopirani kod zaliđe (Ctrl + V) unutar oznake `<body>` (nakon prethodno dodane oznake za odlomak `<p>`).
6. Spremi tekstualnu i HTML datoteku *kemija.html*. Otvori *kemija.html* u mrežnom pregledniku i pregledaj umetnuti sadržaj.

Slika 17. Ugrađeni videozapis

Vježba 12.

1. U tekstualnom dokumentu *kemija.txt* pronađi kôd kojim je ugrađen videozapis i promjeni visinu i širinu prozora za prikaz (`širina = width`; `visina = height`). Visina i širina neka budu proizvoljni brojevi.
2. Spremi tekstualni i HTML dokument *kemija.html*.
3. Otvori mrežnu stranicu u pregledniku.
Što zaključuješ o ugradnji sadržaja i promjeni veličine prozora za prikaz?

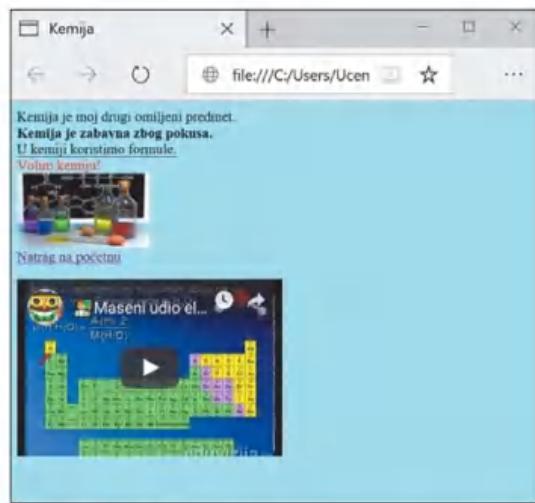
11. Izrađivanje i objavljivanje mrežnih stranica

Rješenje

Kako bismo riješili ovaj zadatak, bilo je potrebno unutar kôda za ugradnju promijeniti brojke pored elemenata *width* i *height*. Tako bi primjerice *width="300"* *height="200"* prikazalo prozor veličine 300 x 200 pikseala. Tako vrlo lako mijenjamo veličinu prozora za prikaz.

Osim videozapisa, možemo ugraditi bilo koji drugi sadržaj s neke druge mrežne stranice. Tu nam mogućnost pojednostavljuje opcija *embed* (ugraditi) pa kreirani kôd kopiramo i umetnemo u svoj dokument.

servisi za generiranje
embed kôda



Slika 18. Izmijenjena visina i širina prozora za prikaz ugrađenog sadržaja

A što ako smo pronašli sadržaj za kojeg opcija *embed* ne postoji? Na internetu postoje servisi za generiranje *embed* kôda. Neki od njih su komercijalni, dok su neki besplatni,

ali je uvjet da je vaš mrežni sadržaj u koji želite ugraditi kôd objavljen i da ima svoju URL adresu.

A screenshot of the siegemedia Embed Code Generator website. The top navigation bar includes links for About, Services, Work, Blog, and Contact, along with social media icons. The main section is titled 'Embed Code Generator' and has tabs for 'Embed Image', 'Embed Video', and 'FAQ'. Under the 'Settings' tab, there are fields for Site Name ('YourDomain.com'), Post URL ('https://domain.com/post-url/'), Image URL ('https://domain.com/image.jpg'), Image Alt ('Infographic Name'), Width of Image ('540px'), and Height of Image ('Leave empty to keep proportion'). To the right, there's a 'Use This Code' section with the generated HTML code for embedding the image. Below this is a 'Results Preview' section with a preview of the code and a 'Share this Image On Your Site' button.

Slika 19. Online servis za kreiranje *embed* kôda

Kada upotrebjavamo opciju *embed* za objavu na vlastitoj mrežnoj stranici, uvijek se prikazuje izvor ugrađenog sadržaja. Za ugrađenu sliku će pisati rjezina URL adresa i odvest će nas na mrežno mjesto gdje je možemo pregledati u izvornom obliku. Ali čak i tada moramo poštovati autorska prava, pa je najbolje upotrebljavati sadržaje za koje smo sigurni da njihovim korištenjem nećemo prekršiti ničije pravo.

Provjerite svoje znanje o HTML jeziku uz kviz 3g11kviz1.htm.

11. Izrađivanje i objavljivanje mrežnih stranica

11.3. Objavljivanje mrežnih stranica i online grafički uređivači

Bez obzira izrađujemo li mrežne stranice u HTML jeziku ili vizualnom (grafičkom) HTML uređivaču, neizostavan je korak objaviti mrežne stranice i podijeliti ih u virtualnom svijetu.

Mrežne stranice koje smo izradili lokalno na svom računalu upotrebom HTML jezika i tekstualnog uređivača ili nekog programa za vizualno uređivanje mrežnih stranica (**WYSIWYG** uređivač) možemo objaviti tako da zakupimo domenu i odberešmo neku od tvrtki koje nude **web hosting**.

Vježba 13.

Potraži na internetu barem tri tvrtke koje nude **web hosting**. Usporeди njihove ponude.

Mrežne stranice smještaju se na računalo **poslužitelj**, koji iznajmljujemo od tvrtke za udomljavanje mrežnih stranica. Domena je adresa mrežne stranice na internetu i nemoguće je da dvije različite mrežne stranice imaju istu domenu. Primjer domene jest www.skole.hr. Postoje razne mrežne stranice koje nam omogućavaju provjeru *dostupnosti domene*, kao npr. www.regica.net.

Cijene udomljavanja mrežnih stranica i iznajmljivanja domene ovise o potrebljima koje imamo: količina prostora na disku poslužitelja, prometu podataka koji ostvarujemo, potrebi za dodatnim uslugama i sl.

Vježba 14.

Posjeti stranicu www.regica.net i provjeri dostupnost neke svoje zamišljene domene.

Mrežne stranice koje smo izradili lokalno na svom računalu možemo prenijeti na računalo poslužitelj programom za prijenos podataka ili možemo upotrijebiti **CMS** – sustav za upravljanje sadržajem koji se instalira na poslužitelju. Na taj način mrežne stranice uređujemo izravno na poslužitelju, što uvelike olakšava objavu i uređivanje mrežnih stranica.

Na internetu postoji velik broj besplatnih alata za izradu i objavljivanje mrežnih stranica kao što su to **WordPress** (www.wordpress.com), **WIX** (www.wix.com) ili **Weebly** (www.weebly.com) na kojima izrađujete korisnički račun, ili **Google Sites** (sites.google.com) na koji imate pristup ako imate korisnički račun za sve Googleove usluge.

Detaljan opis kako upotrebljavati **Google Sites**, **Weebly** i **WordPress** pronaći ćete u mapi Vježba:

3g11Googlesites.mp4, 3g11Weebly.mp4 i 3g11WordPress.mp4.



web hosting – udomljavanje mrežnih stranica

WYSIWYG (*What You See Is What You Get*) su vizualni (grafički) HTML uređivači što znači da ne trebate poznavati HTML jezik da biste izradili svoju mrežnu stranicu. Tehnikom *drag & drop* („dovuci i ispusti“) i pisanjem te oblikovanjem teksta uređujete elemente stranice i odmah je vidljiv krajnji rezultat – izgled stranice u pregledniku.

CMS

Neki su od poznatijih CMS alata **Joomla!**, **Drupal** i **MediaWiki**. Listu svih CMS alata potraži na en.wikipedia.org/wiki/List_of_content_management_systems



11. Izrađivanje i objavljivanje mrežnih stranica

11.4. Priprema multimedijskih sadržaja za prikazivanje na mreži

Multimedijski sadržaj na mrežnim stranicama privlači pozornost i na zanimljiji način prikazuje željeni sadržaj. Za sliku kažemo da govori više od tisuću riječi, a zamislite samo koliko riječi govori videozapis koji sadrži zvuk. Multimedijski sadržaji obogaćuju mrežne stranice, no moramo ih dobro pripremiti kako ne bismo postigli suprotan učinak od onog koji bismo željeli – da otjeraju posjetitelje umjesto da ih privuku.

Formati slike za objavu na mrežnim stranicama

Formati slike za objavu na mrežnim stranicama su **GIF, JPEG i PNG**.

Rezolucija (razlučivost) slike broj je okomitih i vodoravnih piksela koji čine sliku, npr. 800x600. Pravilo koje vrijedi jest: što je veći broj piksela, kvaliteta je slike bolja.

Ti formati omogućuju nam brže postavljanje slika na mrežni poslužitelj jer zauzimaju malo memorijskog prostora, pa samim time i brže putuju kroz mrežu do odredišta, tj. korisnik koji ih pregledava na svom računalu brže će ih preuzeti. GIF format podržava i animaciju.

Osim formata slike, važna je i njezina rezolucija.

Alat koji možete koristiti za obradu slike jest **Paint 3D**, koji je dio operacijskog sustava Windows 10. Također, s interneta možete preuzeti besplatne alate za obradu slike (smanjivanje i obrezivanje slike, podešavanje boje, kontrasta i oštirine, dodavanje efekata). To su primjerice:

- **Paint.NET** (www.getpaint.net/download.html),
- **Gimp** (www.gimp.org/downloads) i
- **PhotoScape** (www.photoscape.org/ps/main/download.php).

Jednako tako možete se koristiti i besplatnim mrežnim uređivačima slika koji nude profesionalne mogućnosti za uređivanje. Neki su od takvih alata **Pixlr** (pixlr.com) i **LunaPic** (www194.lunapic.com/editor).



Slika 20. Primjer slike u JPEG formatu male razlučivosti



Slika 21. Uvećani dio slike

11. Izrađivanje i objavljivanje mrežnih stranica

Formati zvuka za objavu na mrežnim stranicama

Ako na svoju mrežnu stranicu želite postaviti zvučni (audio) zapis, onda će format **MP3** zasigurno biti najbolji izbor. To je jedan od najpopularnijih formata za zvučne zapise na mreži. Postoje različite varijante MP3 zapisa, od onih s velikim gubitcima do onih u kojima je gubitak kvalitete gotovo nečujan. U zvučnom zapisu važan je *bitrate*.



bitrate – broj bitova koji se prenosi u vremenskoj jedinici

Uz MP3, popularni su zvučni formati **WAV** i **Ogg**. Besplatan program za snimanje zvuka jest **Audacity** (www.audacityteam.org/download), a ako želite jedan zvučni format pretvoriti u neki drugi, onda možete upotrijebiti i besplatne mrežne pretvarače (engl. *converter*) (online-audio-converter.com).

Vježba 15.

- Preuzmi na svoje računalo datoteku **3g11grmljavina.wav**.
- U mrežnom pregledniku otvori stranicu online-audio-converter.com i učitaj datoteku **3g11grmljavina.wav**.
- Nakon učitavanja, odaberi format datoteke za konverziju mp3, kvaliteta standardna (128 kbps) i pritisni *Convert*.
- Preuzmi konvertiranu datoteku na svoje računalo, preslušaj obje datoteke i usporedi veličinu datoteke prije i nakon konverzije. Što zaključuješ?

Formati videozapisa za objavu na mrežnim stranicama

Najpopularniji oblik multimedijskog sadržaja zasigurno je video. On sadržava elemente drugih medija (sliku, tekst, animaciju i zvuk) te nam na taj način prenosi poruku na više različitih razina. Najpopularniji videoformat za objavu na mreži jest **MP4** jer ga podržavaju svi mrežni preglednici, pametni telefoni i tableti, kao i pametni TV uređaji (engl. *smart TV*). Također, podržava ga i standard **HTML5**.



Standard **HTML5** aktualna je inačica HTML jezika za izradu mrežnih stranica koja objedinjuje tri različita tipa kôda: **HTML** – osigurava strukturu mrežne stranice; **CSS** – zadužen za izgled mrežne stranice i **JavaScript** – daje mrežnoj stranici funkcionalnost.



S obzirom na to da se danas videosadržaj može besplatno objaviti na besplatnim servisima, kao što su **YouTube** (www.youtube.com) i **Vimeo** (www.vimeo.com), malobrojni su oni koji videozapise postavljaju izravno na svoj mrežni poslužitelj jer tako nepotrebno zauzimaju svoj mrežni prostor. Najčešće se videozapisi postavljaju (engl. *upload*) na jedan od takvih servisa i zatim se ugrađuju (engl. *embed*) na vlastitu mrežnu stranicu.

Kako postaviti videozapis na YouTube i Vimeo, pogledajte u videouputama **3g11youtube.mp4** i **3g11vimeo.mp4**.



Slika 22. Servisi za objavu videozapisa – YouTube i Vimeo

12. Kriptografija i Tkinter

12.1. Teorija prije problema

Kriptografija

Kriptologija (grč. *kryptos* = skriven; *logos* = riječ, govor) je znanost koja se bavi si-gurnošću (skrivanjem) podataka. Sastoјi se od kriptografije i kriptoanalize. **Kriptografija** (grč. *grafein* = pišem) se bavi konkretnim metodama skrivanja podataka, odnosno algoritmima i protokolima za šifriranje (kodiranje, enkriptiranje) i dešifriranje (dekodiranje, dekriptiranje) podataka. **Kriptoanaliza** (grč. *analysis* = raščlanjivanje) ima suprotan cilj: traženje propusta u kriptografskim algoritmima i protokolima s ciljem saznavanja kriptografski zaštićenih informacija ili poboljšanja samih kriptografskih algoritama.

Kriptografski algoritam

Osnovni pojam u kriptografiji jest **kriptografski algoritam** ili šifra (engl. cipher). To je skup kriptografskih funkcija koje s ključem k pretvaraju p (izvorni tekst) u c (šifrirani tekst) i obrnuto. Pritom tekst znači niz znakova neke abecede, npr. latiničnih slova, arapskih znamenaka ili bitova. Kriptografski protokol standardizirani je postupak koji opisuje način upotrebe kriptografskih algoritama u svrhu uspostave sigurne komunikacije, što uključuje tajnost, integritet i autentifikaciju.

Prva i osnovna namjena kriptografije jest **čuvanje tajnosti** podataka. Kriptografska funkcija mora imati takva svojstva da je bez valjanog ključa „nemoguće“ iz poznatog šifriranog teksta dobiti izvorni tekst. Pritom pod nemoguće najčešće podrazumijevamo finansijski ili vremenski neisplativo, a ne teoretski nemoguće. Ako se ključem koristi višekratno, što je najčešće slučaj, treba osigurati i dodatne uvjete, na primjer da je iz više parova izvornog i šifriranog teksta također nemoguće dobiti ključ.

Druga zadaća kriptografije jest **osiguranje integriteta** podataka. Integritet podataka znači osiguranje od slučajnih i namjernih promjena šifriranog teksta. Tajnost nije dovoljna ako postoji mogućnost neopažene izmjene šifriranog teksta.



Claude Elwood Shannon američki je matematičar i kriptograf koji se smatra ocem teorije informacija. Članak *Matematička teorija komunikacija*, kojim je utemeljio teoriju prenošenja informacija, objavio je 1948. godine.

Treća je namjena **autentifikacija**, odnosno utvrđivanje identiteta sudionika u komunikaciji. Integritet daje samo potvrdu da podaci nisu promijenjeni između enkripcije i dekripcije. Autentifikacija pak dokazuje da izvorni tekst potječe od osobe koja tvrdi da ga je poslala. Integritet i autentifikacija povezani su i s neporecivosti, odnosno nemogućnosti da osoba zanijeka slanje određenih podataka i tako zaobiđe poštivanje ugovora (npr. nijekanjem bankovne transakcije).

Kriptografija se povijesno upotrebljavala primarno u vojne i obavještajne sruhe. Međutim, s razvojem tehnologije, osobito elektroničkih računala, poprima sve širu primjenu u svakodnevnom životu. Primjerice, omogućava tajnost telefonskih razgovora koji se prenose nesigurnim linijama. HTTPS, SSL/TLS, SSH i slični protokoli omogućavaju autentifikaciju na internetu i tajnu komunikaciju s drugim računalima, što štiti od prijevara i krađe osobnih podataka.

Digitalni potpisi i certifikati u kombinaciji s navedenim omogućuju prijenos novca (internetsko bankarstvo, e-kupovina) preko interneta poput ostalih vrijednih podataka. Osim

12. Kriptografija i Tkinter

na internetu primjenjuje se za identifikaciju pomoću magnetnih i čip kartica na bankomatima, elektroničkih brava, biometrijskih zapisa (otisci prstiju, izgled šarenice) u osobnim kartama i putovnicama, pri elektroničkom glasovanju itd. Naposljetku omogućuje ograničenje korištenja pojedinim uslugama kao što je Pay TV i ostale tehnologije protiv neovlaštenog kopiranja odnosno piratstva.

Intuitivno se može zaključiti da sigurnost kriptografskog sustava ovisi o složenom dizajnu koji se čuva u tajnosti, poput tajnih protokola i mehaničkih kriptografskih strojeva poznatih iz povijesti. Međutim, u praksi takav dizajn treba izbjegavati. To je načelo u kriptografiji poznato kao **Kerckhoffsova doktrina** („Kriptosustav mora ostati siguran iako je sve o njemu, osim ključa, javno poznato.“) ili **Shannonova maksima** („Neprijatelj uvijek poznaje sustav!“). U praksi je ta doktrina potvrđena više puta uspješnim zlonamjernim napadima na tajne „neprobojne“ sustave. Najbolju sigurnost obično nude dugo godina korišteni algoritmi koji su već prošli mnogobrojne bezuspješne kriptoanalitičke napade te implementacije otvorenog koda koje omogućavaju mnogo većoj populaciji kriptoanalitičara da pronađu propuste u implementaciji ne-go komercijalni softver zatvorenog koda.

U kriptografiji se upotrebljavaju standardna imena preuzeta iz Primjenjene kriptografije Brucea Schneiera. Alice je osoba ili računalo koje šalje šifrirani tekst Bobu. Eve (od engl. eavesdropper) je napadač koji samo pasivno prisluškuje komunikaciju, dok je Mallory (engl. malicious) zlonamjerni napadač koji pokušava promijeniti ili zamijeniti sadržaj teksta, autentificirati se kao Alice ili Bob.

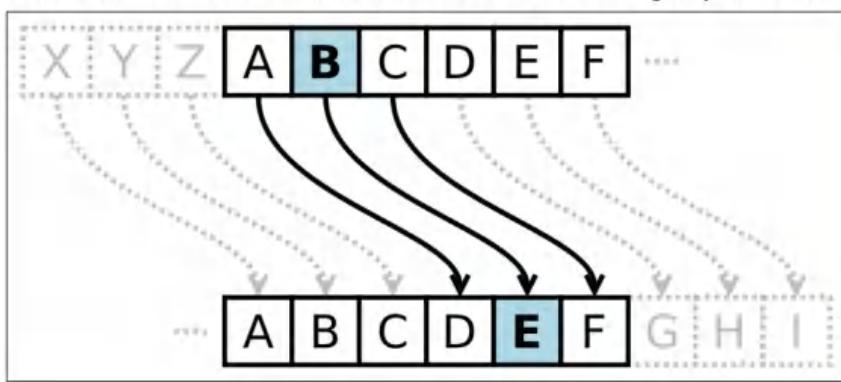
12.2. Supstitucijske šifre

Povijest kriptografije seže u antička vremena, u doba starih Egipćana. Do pojava široko dostupnih elektroničkih računala koristila se uglavnom u vojne i obavještajne svrhe, što je i danas vrlo značajno područje primjene. U antici algoritmi su ograničeni na jednostavne supstitucije (zamjene) i transpozicije znakova teksta, ručnim metodama. Pritom se potvrda autentičnosti obavljala pečatima.

Poznati rimski vojskovođa i državnik Gaj Julije Cezar je u 1. st. pr. Krista koristio jednostavni supstitucijski algoritam, po njemu prozvan Cezarovim algoritmom. Tijekom šifriranja svako slovo abecede zamjenjujemo slovom koje je za k znakova udaljeno od njega, gdje je k ključ/pomak između jedan i broja slova u abecedi. Zamjena je ciklička, tj. zadnjih k slova s kraja abecede se zamjenjuje s prvih k slova. Cezar je koristio ključ $k=3$. U današnjoj engleskoj abecedi to bi značilo zamjenu A s D, B s E, ..., W sa Z, X s A, Y s B, Z sa C. Rimski car August je koristio verziju s $k=1$.

Algoritam Atbash, pronađen u Bibliji, otvoreni tekst šifrica zamjenom prvog slova abecede zadnjim, drugog predzadnjim itd. Dešifriranje vrši na isti način. Prepostavlja se da je bio korišten u cenzuri vjerskih tekstova.

Cezarova šifra



Danas se za miješanje memorije i neozbiljnju enkripciju koristi ROT13, verzija s ključem $k=13$, koja ima dodatno svojstvo da je šifriranje identično dešifriranju.

Slika 1.
Cezarova
šifra

12. Kriptografija i Tkinter

U otvorenim tekstovima koje ćemo šifrirati korist ćemo samo znakove engleske abecede.

Primjer 1.

Napišimo funkcije u Pythonu koje će šifrirati otvoreni tekst i dešifrirati šifrate koristeći Cezarovu šifru.

```
def cezar_sifriraj(tekst, pomak):
    sifrat = ""
    for slovo in tekst:
        sifrat += chr(ord('a') + (ord(slovo) - ord('a') +
pomak) % 26)
    return sifrat
def cezar_desifriraj(sifrat, pomak):
    tekst = ""
    for slovo in sifrat:
        tekst += chr(ord('a') + (ord(slovo) - ord('a') -
pomak) % 26)
    return tekst
>>> cezar_sifriraj('PYTHON', 3)
'SBWKRQ'
>>> cezar_desifriraj('SBWKRQ', 3)
'PYTHON'
>>>
```

12.3. Vigenèreova šifra

Kod supstitucijske šifre svakom slovu otvorenog teksta odgovara jedinstveno slovo šifrata. Recimo, slovo „A“ se kod Cezarove šifre ($k=3$) svaki put preslika u slovo D.

Kod Vigenèreove šifre svako se slovo otvorenog teksta može preslikati u jedno od m mogućih slova, gdje je m duljina ključa. Algoritam šifriranja definiramo na sljedeći način:

1. zapišimo otvoreni tekst;
2. ispod otvorenog teksta, počevši od prvog do zadnjeg slova u njemu, zapisujemo redom slova iz ključa ponavljajući postupak sve dok ne popunimo sva mesta;
3. svako slovo otvorenog teksta šifriramo slovom iz ključa koji je zapisan ispod njega;
4. slovo otvorenog teksta pomicemo za k_i mesta, gdje je k_i pozicija u abecedi odgovarajućeg slova iz ključa. Zamjena je ciklička, nakon slova Z dolazi opet slovo A.

Primjer 2.

Šifrirajmo otvoreni tekst „ALGORITAM“ uz pomoć ključa „PYTHON“. Zapišimo otvoreni tekst i ključ kako je opisano u algoritmu.

A	L	G	O	R	I	T	A	M
P	Y	T	H	O	N	P	Y	T

12. Kriptografija i Tkinter

Prvo slovo otvorenog teksta „A“ pomaknut ćemo za 15 mesta što je redni broj slova „P“ u abecedi i dobiti, naravno, slovo P. Slovo „L“ pomaknut ćemo za 24 mesta („Y“) i dobiti slovo „J“. Ponavljanjem ovih koraka dobit ćemo šifrat oblika „PJZVFVIYF“.

A	L	G	O	R	I	T	A	M
P	Y	T	H	O	N	P	Y	T
P	J	Z	V	F	V	I	Y	F

Uočite da se jednom slovo „A“ šifriralo u slovo „P“, a drugi put u slovo „Y“.

Veliku pomoć u šifriranju ovim načinom možemo dobiti ako koristimo tzv. **Vigenèreov kvadrat** iz kojeg se lako vizualno možemo odrediti šifrat slova.

Vigenèreov kvadrat

 Vigenèreov kvadrat možete preuzeti s mrežne stranice uz udžbenik: [3g12Vigenere.pdf](#).

Napišimo funkcije u Pythonu za šifriranje i dešifriranje Vigenèreovom šifrom.

```
from itertools import cycle
def vigenere_sifriraj(tekst, kljuc):
    sifrat = ""
    for slovo, k in zip(tekst, cycle(kljuc)):
        sifrat += chr(ord('A')+(ord(slovo)-ord('A')+ord(k)-
ord('A'))%26)
    return sifrat
def vigenere_desifriraj(sifrat, kljuc):
    tekst = ""
    for slovo, k in zip(sifrat, cycle(kljuc)):
        tekst += chr(ord('A')+(ord(slovo)-ord('A')-
ord(k)+ord('A'))%26)
    return tekst
>>> vigenere_sifriraj('ALGORITAM', 'PYTHON')
'PJZVFVIYF'
>>> vigenere_desifriraj('PJZVFVIYF', 'PYTHON')
'ALGORITAM'
>>>
```

12.4. Playfairova šifra

Nakon što je postalo jasno da se Vigenèeova šifra ipak može razbiti metodama za određivanje duljine otvorenog teksta i ključa, počelo se prelaziti na nove načine šifriranja. Playfairova šifra ušla je u uporabu zbog svoje jednostavnosti. Definirao ju je britanski znanstvenik Charles Wheatstone 1854. godine, a ime je dobila po njegovom prijatelju barunu Playfairu od St. Andrewsa koji ju je popularizirao.

12. Kriptografija i Tkinter

Šifriranje se temelji na tablici s pet redaka i pet stupaca u koju prvo redom upisujemo slova iz ključa (bez ponavljanja), a zatim redom preostala slova iz abecede. Slova I i J zapisujemo na isto mjesto. Npr., odgovarajuća tablica za ključ „PYTHON“ bit će:

P	Y	T	H	O
N	A	B	C	D
E	F	G	IJ	K
L	M	Q	R	S
U	V	W	X	Z

Sada podijelimo otvoreni tekst koji šifriramo na blokove od po dva slova. Niti jedan blok ne smije se sastojati od dva jednaka slova, a duljina teksta mora biti parna. I jedno i drugo postižemo umetanjem slova X gdje je to potrebno.

Kod šifriranja bloka od dva slova, ovisno o položaju slova u matrici, razlikujemo tri slučaja:

1. Slova su u istom retku – zamijenimo ih sa slovima koja se nalaze za jedno mjesto udesno, pri čemu poštujemo cikličnost. Npr. AB ↔ BC, MS ↔ QL.
2. Slova su u istom stupcu – zamijenimo ih sa slovima koja se nalaze za jedno mjesto ispod pri čemu poštujemo cikličnost. Npr. AF ↔ FM, RX ↔ XH.
3. Inače, pogledajmo pravokutnik koji određuju ta dva slova, te ih zamijenimo s preostala dva vrha tog pravokutnika. Redoslijed je određen tako da najprije dođe ono slovo koje se nalazi u istom retku kao prvo slovo u polaznom bloku. Npr. MH ↔ RY.

Primjer 3.

Šifriramo otvoreni tekst „PROGRAMIRANJE“ uz pomoć ključa „PYTHON“. Koristeći gornju tablicu lako možemo odrediti šifrat:

PROGRAMIRANJE → PR OG RA MI RA NJ EX → HL TK MC RF MC CE IU

```
>>> playfair_tablica('PYTHON')
[[ 'P', 'Y', 'T', 'H', 'O' ], [ 'N', 'A', 'B', 'C', 'D' ], [ 'E', 'F', 'G', 'I', 'K' ],
[ 'L', 'M', 'Q', 'R', 'S' ], [ 'U', 'V', 'W', 'X', 'Z' ]]
>>> playfair_sifriraj('PROGRAMIRANJE', 'PYTHON')
'HLTKMCRFMCCEIU'
>>> playfair_desifriraj('HLTKMCRFMCCEIU', 'PYTHON')
'PROGRAMIRANJEX'
>>>
```

12. Kriptografija i Tkinter

Od problema do rješenja

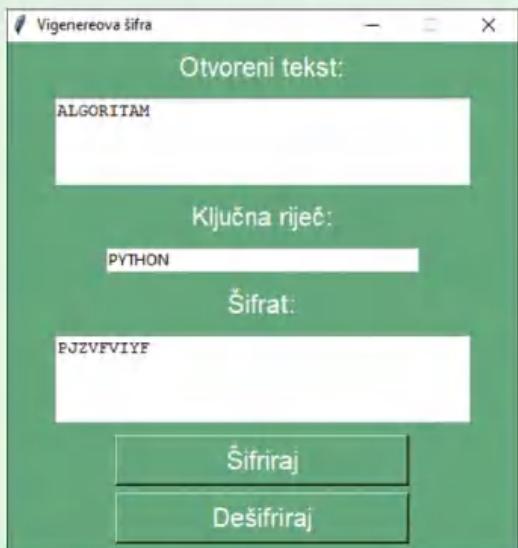
Nakon što smo opisali osnovne algoritme za šifriranje otvorenog teksta, objedinimo ih u program s grafičkim sučeljem. Prijedlog jednog rješenja možete vidjeti u nastavku, a samostalno ga možete analizirati i nadograđivati ako s mrežne stranice uz udžbenik preuzmete pripadne programske kodove.



Slika 2. GUI šifriranje (glavni program)



Slika 3. GUI Cezarova šifra



Slika 4. GUI Vigenereova šifra



Slika 5. GUI Playfairova šifra

Programske kodove možete preuzeti s mrežne stranice uz udžbenik: [3g12glavni.py](#), [3g12cezar.py](#), [3g12vigenere.py](#) i [3g12playfair.py](#).

13. A sad, nešto sasvim drugačije

U želji da se učenicima približe temeljne ideje programiranja, a u skladu s kurikulumom koji pokušava biti neutralan prema odabiru programske jezike, često se čine didaktički ustupci pa se određene činjenice prešućuju. Dizajn Pythona počiva na nekoliko izuzetno jednostavnih ideja, prilagođenih prvom susretu učenika s programiranjem. Međutim, Python se ponekad ponaša bitno drugačije od većine ostalih "konvencionalnih" programskih jezika. Takvi se detalji obično zanemaruju tijekom početne faze učenja. Ali kad učenik počne izraditi naprednije programe, to može dovesti do frustracija jer mora mijenjati dotad naučene obrasce razmišljanja.

U ovome poglavlju činimo prvi korak prema popravljanju takvog stanja – nudimo sadržaj s naglaskom na Pythonovo "shvaćanje svijeta". Namijenjen je učenicima koji počinju razvijati svijest o tome da razlike među programskim jezicima nisu samo stvar vokabulara (slično kao i kod govornih jezika).

13.1. Varijable u Pythonu

O kutijama i naljepnicama

Često se u udžbenicima informatike nalazi definicija poput „Varijabla je memorijska lokacija čiji se sadržaj može mijenjati tijekom izvođenja programa“. Iako se ta rečenica može shvatiti na način da bude točna, često se shvaća na način koji je, iako točan za neke programske jezike, vrlo pogrešan za Python i kasnije vodi do velikih zabuna.

append

Naime, u objektno orijentiranim jezicima često se vrijednost nekog objekta može mijenjati. Ako imamo listu, možemo metodom `list.append` promijeniti njezinu vrijednost. Zabuna nastaje kad učenik koji je svjestan i gornje definicije i mijenjanja vrijednosti objekta pomisli da se radi o istom procesu. To je pogrešno.

Najprije, pogledajmo sljedeći kod:

```
>>> shopping = ['jaja', 'mlijeko']
>>> id(shopping)
54007688
>>> shopping.append('kruške')
>>> id(shopping)
54007688
```

Kao što vidimo, `id(shopping)` ostao je isti (`id` možemo smatrati svojevrsnim OIB-om objekta i u najpopularnijoj implementaciji Pythona on doista jest adresa memorijske lokacije na kojoj je objekt zapisan). Odnosno, *nismo* promijenili varijablu. Varijabla je i dalje ime za jedan te isti objekt, samo je taj objekt promijenio svoju vrijednost. Dakle, možemo mijenjati vrijednost objekta bez mijenjanja varijable koja ga imenuje.

13. A sad, nešto sasvim drugačije

A sada pogledajmo sljedeći kod:

```
>>> učenici = ['Ivana', 'Boris', 'Jurica']
>>> id(učenici)
54165128
>>> učenici = ['Marko', 'Ivana']
>>> id(učenici)
51339656
```

Za razliku od prethodnog koda, sada se `id(učenici)` promjenio: naredba pridruživanja doista može promijeniti varijablu. Čak i da smo pridružili novu listu s istim članovima, varijabla bi se promijenila — jer to nije ista lista, u što se lako možemo uvjeriti ako pogledamo gore: `append` na jednoj od njih mijenjao bi samo nju, a ne i onu drugu. Važnije, *nijedan* objekt u donjem primjeru nije promijenio svoju vrijednost. Imamo dvije liste na različitim mjestima u memoriji – najprije jednu od njih zovemo imenom `učenici`, a kasnije onu drugu počnemo zvati tim imenom. Dakle, možemo mijenjati vrijednost varijable bez mijenjanja vrijednosti objekta.

Treba razlikovati ta dva procesa. Objekti nisu "spremljeni u" varijable — varijable su samo imena za objekte koji postoje nezavisno o njima. Zato kažemo da u Pythonu „varijable nisu kutije, nego naljepnice“. Vrijednost *varijable* identitet je imenovanog objekta i može se mijenjati pridruživanjem. Vrijednost *objekta* pak ovisi o tipu objekta (za listu su to vrijednosti i redoslijed njezinih elemenata, za skup su to samo elementi,...) i može se mijenjati metodama na objektu.

Promjenjivi i nepromjenjivi objekti

Ipak, ne mogu *svi* objekti mijenjati svoju vrijednost. Čitava aritmetika zasnova je na ideji da brojevi kao objekti imaju nepromjenjivu vrijednost.

Možda ste čuli šalu „ $2+2=5$ za dovoljno velike vrijednosti od 2“. Razlog zbog kojeg je smatramo smješnom upravo je naše duboko uvjerenje o nepromjenjivosti vrijednosti brojeva. Zato pogledajmo sljedeći kod:

```
>>> starost = 15
>>> id(starost)
1842778368
>>> starost += 3
>>> id(starost)
1842778464
```

Iako nam se čini da je „`+=3`“ samo metoda pozvana na objektu `starost`, vidimo da se `id` promjenio. Python odbija (precizno, ne pruža mogućnost) promijeniti vrijednost objekta tipa `int` pa radije stvara novi objekt `18` i na njega lijepi naljepnicu `starost`. Ako imamo `a=b=2`, tada nakon `a+=1` objekt imena `b` i dalje ima istu vrijednost, broj dva. Tada je `a` ime za novi objekt vrijednosti tri.



U literaturi se često promjenjivi i nepromjenjivi objekti zovu **mutable** i **immutable** objekti. (eng. *Mutable and Immutable objects*)

13. A sad, nešto sasvim drugačije

Možemo to interpretirati kao „ništa čudno, `a+=1` samo je pokrata za `a=a+1`, dakle to je očito naredba pridruživanja i mijenja varijablu“, ali onda nas iznenađenje čeka u ovom kodu:

```
>>> shopping = ['jaja', 'mlijeko']
>>> id(shopping)
54165128
>>> shopping += ['kruške']
>>> id(shopping)
54165128
```

Vidimo da se tu doista promijenila lista imena `shopping` jer Python zna da liste mogu mijenjati svoju vrijednost. Ako imamo `a=b=[2]`, tada je nakon `a+=[1]` objekt imena `b` (istи као и objekt imena `a`) promijenio svoju vrijednost.

Extend metoda

Prava je istina sljedeća: `a+=t` jest pokrata za `a=a+t`, međutim, metodi koja zbraja poslana je dodatna informacija da se zbroj može izvesti „na licu mjesta“, ne stvarajući novi objekt. Razni tipovi interpretiraju tu informaciju na razne načine: `int` je jednostavno znamaruje, dok `list` u tom slučaju poziva metodu `extend` i vraća lijevi operand (istи objekt kojem je `extend` promijenila vrijednost). To dovodi do čudnog ponašanja u sljedećem slučaju:

```
>>> predmet = ('Informatika', [5, 4, 5])
>>> predmet[1] += [4, 4]      # dobili smo još 2 četvorke
TypeError: 'tuple' object does not support item assignment
>>> predmet                  # greška, ali objekt jest promijenjen
('Informatika', [5, 4, 5, 4, 4])
```

Ako pogledamo gornji odlomak, jasno je što se dogodilo: modificirani + obavio je dodavanje na licu mjesta, dok je pokušaj „pridruživanja“ promijenjenog objekta u `predmet[1]` javio grešku jer vrijednost n-torce ne možemo mijenjati. `predmet` je i dalje par istih objekata, samo je drugi od njih promijenio vrijednost. Mogli smo izbjegići grešku pisanjem `predmet[1].extend([4, 4])`.

Stvaranje i uništavanje objekata

Mogli bismo se pitati sljedeće: ako varijable postoje neovisno o objektima, što je to što stvara objekte? Programski jezici u kojima varijable predstavljaju „kutije“ obično zahtijevaju njihovu deklaraciju, koja onda stvori varijablu: ime u nekom dosegu, memorijski prostor (fiksne veličine ovisne o tipu varijable) i početnu vrijednost objekta koji je u toj varijabli spremljen. Python je očito bitno drugačiji. Postoje određeni sintaktički simboli čije pojavljivanje u kodu zahtijeva od interpretera stvaranje novog objekta.

Tako uglate zagrade stvaraju listu, vitičaste rječnik ili skup (ovisno o dvotočkama), zarez (ako nije unutar nekog drugog konstrukta) stvara n-torku, navodnici stvaraju string, znamenke stvaraju broj itd. Naredba `a=[3]` ne znači „stvori kutiju `a` i u nju stavi vrijednost `[3]`“, već „stvori objekt `[3]` i nalijepi na njega naljepnicu `a`“. Pridruživanje ne stvara

13. A sad, nešto sasvim drugačije

objekt; uglate zgrade stvaraju objekt, a pridruživanje mu samo dodjeljuje ime. Također, tipovi se mogu pozivati (kao funkcije) i tako stvarati nove objekte. `list()` je isto što i `[]`, `str` je isto što i `''`, `bool()` je isto što i `False`. Prazan skup čak nema sintaktički posebnu oznaku (jer `{}` označava prazan rječnik), ali uvijek ga možemo stvoriti s pomoću `set()`. Aritmetičke operacije, kao što smo vidjeli, također stvaraju nove objekte.

Sad imamo drugi problem: ako Python stvara objekte „šakom i kapom“, kako to da mu se memorija ne prepuni? Pogledajmo kod:

```
zbroj = 0
for broj in range(10**6):
    zbroj += broj
```

Sigurno ne želimo u memoriji posebno držati sve parcijalne zbrojeve (0, 1, 3, 6, 10, 15,...), a da i ne spominjemo sve vrijednosti varijable `broj`. Python ima vrlo jednostavno pravilo: *ako na objektu nema nijedne naljepnice, taj se objekt uništava i njegova memorija oslobađa*. Naime, ako ne postoji način kojim možemo dozvati određeni objekt, on kao da i ne postoji za ostatak koda: može se obrisati bez promjene značenja ostalih naredbi jer one ionako nemaju pristup njegovoj vrijednosti. (Pokazuje se da to pravilo nije dovoljno da se sprijeći „curenje memorije“ i zato Python ima i neka dodatna pravila, ali samo to već predstavlja ogromnu uštedu.)

Svaki Pythonov objekt ima brojač koji se povećava pridruživanjem nekog imena tom objektu (ili njegovim stavljanjem u neke spremnike, npr. listu ili skup), smanjuje kad neko njegovo ime izade iz dosega ili se drugačije uništi (npr. naredbom `del`), a ako se smanji na nulu, objekt je označen za brisanje. Dakle, ako napišemo `a=b=3125`, tada na objektu 3125 postoje dvije naljepnice. Nakon `a+=5` postoji samo jedna (`b`, jer `a` je sad naličljena na drugi objekt, 3130), a nakon još `b=712` ne postoji više nijedna i taj se objekt može uništiti. To ne znači da će odmah biti uništen (jer to zahtijeva vrijeme), ali kad Pythonu zatreba još memorije, sigurno će prije oslobađati memoriju takvih objekata nego tražiti još od operativnog sustava.



Sustav kojim se u pojedinim programskim jezicima automatski brišu nepotrebni objekti popularno se zove Garbage Collector (sakupljač smeća).

Hashiranje i cachiranje

Objekti čije su vrijednosti potpuno nepromjenjive (jer njihovi tipovi nemaju nikakve metode kojima bi se mogla promijeniti vrijednost i ne dopuštaju pristup svojim podelementima ili zahtijevaju da i oni budu potpuno nepromjenjivi) imaju nekoliko zanimljivih svojstava. Prvo je da Python može „varati“ kad zatražimo novi objekt i pružiti nam neki koji već ima u memoriji. Nećemo primjetiti razliku jer ionako ne možemo mijenjati njegovu vrijednost.

Primjerice, prema dosad napisanom, svaki put kad u kodu napišemo `12`, Python bi morao stvoriti novi objekt koji predstavlja broj dvanaest. To je očito rasipno, a postaje još gore ako primijetimo da se isto odnosi i na brojeve poput `1` ili `0`, koji su izuzetno česti u programiranju. (Ne radi se samo o programima koje mi pišemo; gotovo čitava standardna biblioteka dopušta Pythonu pristup svojim objektima.)

13. A sad, nešto sasvim drugačije

cache

Python zato ima *cache* (čitaj: *keš*), posebni dio memorije gdje drži objekte (najčešće cijele brojeve) za koje smatra da bi mu mogli često trebati. Tradicionalno su cijeli brojevi od -5 do 256 u *cacheu* i uvijek se iz njega dohvaćaju, dok se ostali brojevi konstruiraju prema potrebi — ali ništa ne sprečava Python da i druge potpuno nepromjenjive objekte stavi u *cache* ako uoči da se često konstruiraju. Čest su primjer kratki stringovi: ako na puno mesta u kodu imamo '*hej*', moguće je da će ga Python *cachirati*.

hashiranje

Drugo zgodno svojstvo potpuno nepromjenjivih objekata jest mogućnost njihova distribuiranog preslikavanja u male brojeve, popularno zvano *hashiranje* (razlikujte od *cachiranja*!). Da bismo shvatili čemu to služi, pogledajmo sljedeći primjer: zamislimo da imamo skup od milijun elemenata i želimo vidjeti je li broj 18 među njima. Naravno, uvjek možemo ispitivati elemente skupa jedan po jedan i vidjeti je li ikoji od njih jednak 18 — ali to je presporo, a, osim toga, onda nam ne treba skup, to smo mogli i s listom. Možemo li nekako za brže pretraživanje iskoristiti činjenicu da redoslijed nije bitan?

Ako su u skupu samo cijeli (ili realni) brojevi, možemo ga držati sortiranog u memoriji i tada ćemo binarnim traženjem lako naći postoji li 18 među njegovim elementima, ali što ako u skupu nisu samo cijeli (ili realni) brojevi? Skupovi mogu kao elemente imati kompleksne brojeve ili elemente različitih tipova koji nisu međusobno usporedivi. Osim toga, sortirani skup jako je nezgodan za ubacivanje elementa: ako želimo u sortirani skup cijelih brojeva ubaciti -389, vjerojatno ćemo puno elemenata morati pomicati „udešno“ da bismo oslobođili mjesto za njega. Taj problem možemo riješiti binarnim stablom traženja, ali i dalje ostaje problem neusporedivih elemenata.

hash()

Kad bismo znali da su u skupu samo brojevi između 0 i dva milijuna, zapis skupa bio bi vrlo jednostavan: samo na mjestima brojeva koji jesu elementi skupa zapišemo `True`, a na ostalima `False`. Tada je jako lako i provjeriti pripadnost konkretnog elementa skupu, kao i ubaciti ili izbaciti element bez pregledavanja i mijenjanja ostalih elemenata. Funkcija *hash* omogućava nam upravo to s objektima koji ne moraju biti cijeli brojevi: to je preslikavanje raznih potpuno nepromjenjivih objekata u cijele brojove, takvo da jednakе objekte preslikava u isti broj, ali je vrlo mala šansa da dva različita objekta preslika u isti broj. Tako proizvoljni objekt `x` možemo zapisati u skup na mjesto `hash(x) % N`, gdje je `N` broj mesta koji smo rezervirali za elemente svojeg skupa.

```
>>> hash('Python')      # programski jezik
753818751286872997
>>> hash('python')     # zmija
7395930184026331499
```

Ako se koristimo tom tehnikom (Python se njome koristi kod skupova i rječnika), možemo vrlo brzo ustanoviti je li bilo koji objekt u skupu: samo pogledamo na mjesto njegova *hasha* modulo `N` (i eventualno nekoliko sljedećih, za slučaj da se različiti elementi s istim *hashom* nalaze u skupu — no i dalje bitno manje nego sve elemente redom). No, primijetimo da tehnika ima smisla samo za objekte koji su potpuno nepromjenjivi: promjenom vrijednosti objekta promjenio bi se i *hash*, odnosno mjesto na kojem ga treba tražiti. Zato Python zahtijeva da elementi skupova i ključevi rječnika budu potpuno ne-

13. A sad, nešto sasvim drugačije

promjenjivi: brojevi, stringovi, logičke vrijednosti, ili pak n-torce takvih objekata sigurno su u redu.

Zapravo, Python zahtijeva da takvi objekti budu *hashabilni*, odnosno da je funkcija *hash* na njima definirana, a *hash* je najkorisnije definirati s pomoću vrijednosti objekta. Objekti **mogu** *hash* vezati na svoj identitet (Python to čini za instance korisnički definiranih klasa, osim ako one definiraju svoj *hash*), i tada se mogu mijenjati po volji, ali loša strana toga jest što možemo upotrijebiti samo izvorni objekt za gledanje u skup ili rječnik. Bilo kakav pokušaj „rekonstrukcije“ jednakog objekta rezultirao bi odgovorom da elementa nema jer bi *id* (a time i *hash*) bio različit. Primjerice, kad bi liste vezale *hash* na identitet, mogli bismo konstruirati skup {[3]}, ali bi [3]in{[3]} svejedno bilo False jer bi objekt s lijeve strane operatora *in* imao različitu *hash* vrijednost od objekta u skupu.

Zašto Python stavlja toliko težište na optimizaciju skupova i rječnika? (Zapravo samo rječnika; skup je implementiran kao specijalni slučaj rječnika koji ima samo ključeve.) Početniku programeru može se činiti da se rječnici u Pythonu ne upotrebljavaju toliko često, ali se zapravo upotrebljavaju posvuda. Prenošenje imenovanih argumenata u funkciju, atributi instanci korisnički definiranih klasa, čak i sama preslikavanja imena u objekte u Pythonu su obično implementirani s pomoću rječnika.

Jedan od primjera upotrebe rječnika: ako ste gornje primjere pisali u interaktivnom sučelju, upišite *vars()* i pritisnite Enter. Dobit ćete rječnik s mnogim čudnim ključevima, ali nekoliko njih, kao i pripadajuće vrijednosti, sigurno ćete prepoznati.

```
>>> vars()
{..., '__package__': None, '__name__': '__main__',
 'učenici': ['Marko', 'Ivana'], 'shopping': ['jaja',
 'mljekko', 'kruške'], 'starost': 18, 'predmet':
 ('Informatika', [5, 4, 5, 4, 4]), ...}
```

I tako smo saznali što su **zapravo** varijable u Pythonu.

13. A sad, nešto sasvim drugačije

13.2. Matrice na Pythonov način

Kopiranje

Vidjeli smo u priči "Variable u Pythonu" da varijable nisu kutije u koje stavljamo svoje objekte, nego naljepnice koje na njih lijepimo. Iz toga proizlazi da samim pridruživanjem imena objektu ne možemo ni na koji način promjeniti taj objekt ni stvoriti novi. Također smo rekli da vrijednost objekta ovisi o njegovu tipu: "vrijednost" otvorene datoteke na disku u mnogočemu se razlikuje od vrijednosti neke liste, koja se opet bitno razlikuje od vrijednosti logičkog objekta True. Za objekt None možemo čak tvrditi da *nema* vrijednost — zapravo, on predstavlja izostanak vrijednosti.

Kopiranje je akcija kojom stvaramo novi objekt iste vrijednosti kao neki već postojeći objekt; a kako je vrijednost vrlo "subjektivan" pojam, kopiranje također ovisi o tipu objekta. Za neke čak vjerojatno nema smisla: primjerice, Pythonovi generatori (funkcije koje se mogu zaustaviti i nastaviti usred izvršavanja) imaju vrijednost koja postoji samo u širem smislu kompletног stacka izvršavanja. Bilo bi prilično teško to stanje "zamrznuti" i nekako duplicirati — iako je teoretski moguće, i postoje jezici (kao Ruby) koji podržavaju tu mogućnost u obliku tzv. *kontinuacija*.

Također, za nepromjenjive objekte (kao što su cijeli brojevi) kopiranje je potpuno bespotrebna operacija: budući da broj 5 ima vrijednost koju je nemoguće promjeniti, sve jedno je imamo li dva takva objekta s istom vrijednošću ili samo jedan. To smo također vidjeli u priči o *cachiranju*.

Zapravo, vidimo da imamo neku vrstu spektra: na jednom su kraju potpuno nepromjenjivi objekti, čija je vrijednost uvijek ista i čije je kopiranje samo gubitak vremena, dok su na drugom kraju vrlo komplikirani objekti čija vrijednost seže izvan granica Pythonova objektnog modela (kao što su pauzirani generatori ili otvorene datoteke na disku) i čije se kopiranje ne može svesti pod neko jednostavno ujednačeno sučelje.

Ipak, postoji i "siva zona" između tih dvaju ekstrema: to su promjenjivi spremnici koji sačuvaju nepromjenjive objekte i njih čemo ponekad zaista trebati kopirati (iako ne toliko često koliko mislimo ako smo već programirali u jezicima u kojima je kopiranje implicitno). Kanonski je primjer lista cijelih brojeva, iako to može biti i skup kompleksnih brojeva ili rječnik logičkih vrijednosti. Pravilo je u Pythonu da takvi objekti imaju metodu copy, koja vraća "plitku" kopiju objekta na kojem je pozvana.

```
>>> original = alias = [2, 5, -7, 3]
>>> kopija = original.copy()
>>> alias is original, kopija is original
(True, False)
```

U slučaju listi, ista funkcionalnost može se postići i naredbom kopija=original[:]. Ipak, u profesionalnom programiranju savjetuje se upotreba metode copy jer je jasnije što se želi postići i funkcionira na svim promjenjivim spremnicima, a ne samo na listama.

Kopiranje objekta

Plitka kopija objekta

13. A sad, nešto sasvim drugačije

Ni liste nisu kutije

Primijetimo da smo govorili o promjenjivim spremnicima koji sadržavaju *nepromjenjive* objekte, i to je jako bitno. Recimo, vrijednost liste možemo smatrati jednostavno vrijednostima njezinih elemenata koje su naslagane jedna do druge — ali Python tako ne misli. Zašto? Mnoge objekte čija je vrijednost Pythonu komplikirana ili je ne može u potpunosti dohvatiti također bismo htjeli staviti u listu ili neki drugi tip podataka implementiran s pomoću liste.

Recimo, Python ima `asyncio`, modul za asinkrono izvršavanje koda, koji posjeduje tzv. petlju događaja (*event loop*) kako bi se osiguralo da svi zadaci koji se počnu izvršavati dobiju priliku da se izvrše do kraja. U trenutku kad neki od njih dođe do mjesta gdje čeka na vanjski podražaj (baza, datoteka, mreža, tipkovnica...), Python ga stavlja na kraj reda i nastavlja s izvršavanjem sljedećeg zadatka s početka reda. Taj je red (*queue*) više-manje obična lista i u njoj očito nisu kopije zadataka koji se izvršavaju (rekli smo da je jako teško kopirati napola izvršenu funkciju), već zadaci sami.

Asynco
petlja dogadaja

Dakle, liste se ne brinu o vrijednostima svojih elemenata, već samo o njihovim identitetima. "Stavljanje elementa u listu", baš kao i "stavljanje u varijablu", ne znači da ga *kopira* u listu, nego samo da mu daje novo ime oblika `lista[index]`. To ima mnoge prednosti (bilo kakav objekt može biti element liste), ali ima i neke mane. Najvidljivija je vjerojatno ona koju vidimo kad pokušamo napraviti listu čiji su elementi također liste.

```
>>> redak = [0, 0, 0]           # skraćeno: redak = [0] * 3
>>> dupli = [redak, redak]     # skraćeno: dupli = [redak] * 2
```

Ako pogledamo literale (parove uglatih zagrada; pogledajte priču "Stvaranje i uništavanje objekata"), vidimo da u Pythonovoj memoriji sada postoje sveukupno samo dvije liste. Jedna se zove `dupli`, a druga ima bar tri imena: `redak`, `dupli[0]` i `dupli[1]`. U svakom slučaju, `dupli` se sastoji od dvaju elemenata, ali ti elementi predstavljaju istu listu, baš kao što običnim pridruživanjem dvaju imena istoj listi nismo napravili nikakvu kopiju.

Naravno, kao i u slučaju običnih varijabli, ako se radi o nepromjenjivim objektima, zapravo nije bitno jesu li kopirani. Nema nikakve veze jesu li `redak[1]` i `redak[2]` različiti objekti s istom vrijednošću ili jedan te isti objekt jer je nula ionako nepromjenjiva. No, u slučaju *promjenjivih* elemenata izuzetno je bitno predstavljaju li različite objekte jer želimo znati hoće li promjena jednog elementa utjecati na drugi.

```
>>> dupli[0].append(8)
>>> dupli[1]
[0, 0, 0, 8]
```

Ponekad je to upravo ono što želimo, ali često nije. Ako želimo implementirati matricu, često je početničko rješenje lista listi, no tada *sigurno* želimo da su nam retci predstavljeni različitim objektima, čak i kad su im vrijednosti iste.

13. A sad, nešto sasvim drugačije

Matrice kao liste listi

Dakle, dupli nije matrica 2×3 . Ako bismo htjeli proizvesti takvu matricu, morali bismo napraviti tri liste – samu matricu kao listu redaka i njezina dva retka kao zasebne liste – odnosno tri puta izvrijedniti neki literal s uglatim zagradama. Najjednostavniji je način, naravno,

```
>>> matrica = [[0, 0, 0], [0, 0, 0]]
```

ali što ako želimo veću matricu, recimo 52×68 ? To bi zahtijevalo 53 para uglatih zagrada. Pojedine retke i dalje možemo pisati kao $[0] * 68$, odnosno ne moramo pisati 68 nula, ali moramo li doista taj par uglatih zagrada napisati još 51 put (i još jednom oko svega)?

Naravno, ne moramo. Uz sekvencijalno izvršavanje, znamo i druge načine na koje Pythonu možemo reći da neke naredbe izvrši više puta. Možemo se koristiti običnom `for` petljom

```
matrica = []
for i in range(52):
    matrica.append([0] * 68)
```

ili, jednostavnije, komprehenzijom

```
>>> matrica = [[0] * 68 for i in range(52)]
```

Ono što je ključno jest da se doista 53 puta izračunava neki literal s uglatim zagradama kako bismo doista imali 53 nezavisne liste u memoriji i kasnije ih mogli mijenjati.

Zanimljiva posljedica toga da se spremnici brinu samo o identitetu, a ne o vrijednosti svojih "spremljenih" objekata jest da ako ne planiramo mijenjati dimenzije matrice nego samo njezine elemente, vanjski spremnik uopće ne mora biti promjeniv:

```
>>> M = [0] * 3, [0] * 3
>>> M[0][2] = 5                                     # ne mijenjamo M, mijenjamo M[0]
>>> M
([0, 0, 5], [0, 0, 0])
```

Duboko kopiranje

Pokušajmo sad kopirati svoju matricu:

```
>>> kopija = matrica.copy()
>>> matrica[5][7] = 23
>>> kopija[5][7]
23
```

Kako se to dogodilo? Ne bi li metoda `copy` trebala stvoriti novu listu? Svakako, ali samo jednu, ne njih 53. Sjetimo se, lista nema ništa s unutarnjom strukturom svojih elemenata, ona samo pamti njihove identitete pod indeksima 0, 1, 2,...; `copy` će stvoriti novu listu, ali će njezini elementi biti isti oni elementi koji su bili u originalnoj listi. Opet, ako su ti elementi ne-promjenjivi, to je sasvim u redu, ali ako su oni sami promjenjivi spremnici nepromjenjivih objekata, možda bismo željeli i njih kopirati... a što ako su njihovi elementi promjenjivi?

Zapravo, vidimo da nam treba rekurzivno (uskoro ćemo saznati što je to) kopiranje: ako smo pozvani na potpuno nepromjenjivom elementu, samo ga vratimo. Ako smo pozvani na spremniku, rekurzivno se pozovemo na svakom njegovu elementu i rezultate sta-

Duboko kopiranje

13. A sad, nešto sasvim drugačije

vimo u novi spremnik.

Ta metoda zove se "duboko kopiranje", za razliku od "plitkog kopiranja" koje radi metoda `copy`. Python u svojoj standardnoj biblioteci ima i algoritam za duboko kopiranje, bar za uobičajene spremnike. Treba reći da, iako sasvim dobro radi za jednostavne stvari kao što su liste listi cijelih brojeva, to je *heuristički* algoritam: ne može raditi općenito jer ne postoji jedinstveni način za kopiranje proizvoljnih objekata.

```
>>> from copy import deepcopy  
>>> kopija = deepcopy(matrica)  
>>> matrica[12][17] = 35  
>>> kopija[12][17]  
0
```

`deepcopy()`

Naravno, u tako jednostavnim slučajevima gdje je dubina ograničena, mogli smo i "plitko" kopirati svaki redak i kopije spremiti u novu listu:

```
>>> kopija = [redak.copy() for redak in matrica]
```

Uočimo da su načini za kreiranje kopije liste lista koji su korišteni u poglavljiju "Od problema do rješenja" bili na tragu osmišljavanja metode `deepcopy`.

Matrice kao rječnici

Moraju li matrice biti baš toliko komplikirane? Zašto ne bi bile poput običnih listi, jednostavni spremnici nepromjenjivih brojeva, samo dvodimenzionalni? Nažalost, Python nema ugrađenu podršku za takav tip podataka. Postoje biblioteke (kao `numpy`) koje ih podržavaju, ali nisu dio standardne Pythonove instalacije. Ipak, pogledajmo što uz malo maštice možemo napraviti s Pythonovim standardnim spremnicima.

Pažljivim gledanjem možemo uočiti da je osnovni problem dubina reprezentacije. Obično matematičke objekte kao što su brojevi doživljavamo nepromjenjivima, a u Pythonu je vrlo jednostavno raditi sa spremnicima nepromjenjivih objekata. Naravno, Python podržava i vrlo komplikirane ugniježđene strukture, ali u njima očekuje *objekte* s identitetom, koji se ne kopiraju bez dobrog razloga.

```
>>> from types import SimpleNamespace as Učenik  
>>> Marko = Učenik(ime='Marko Matovina', razred='3.C',  
... ocjene={'Informatika': [4], 'Matematika': [3, 4, 4]})  
>>> Petar, Kristijan = Učenik(), Učenik(nadimak='Kiki')  
>>> Luka = Učenik(ime='Luka', prijatelji=[Petar, Marko])
```

U ovakvoj situaciji *sigurno* želimo da Lukin prijatelj nije nekakav Markov klon, nego baš Marko sâm:

```
>>> Marko.ocjene['Informatika'].append(5)  
>>> Luka.prijatelji[1].ocjene  
{'Informatika': [4, 5], 'Matematika': [3, 4, 4]}
```

Python je optimiziran za takve stvari: pamćenje i ažuriranje vrlo kompleksnih struktura stvarnih objekata s identitetom izuzetno je brzo i lagano. Samo, matrice se tu ne uklapaju, bar ne na način kako smo ih izvorno zamislili, jer njihovo "spremničko" ponašanje

13. A sad, nešto sasvim drugačije

seže razinu dublje nego što to Python očekuje. Retci u matrici *ne smiju* se ponašati kao pametni objekti s identitetom, već kao obični spremnici "druge razine". To je glavno mjesto koje se protivi Pythonovoj filozofiji i zato nailazimo na čudne poteškoće.

Postoji li način da matrice također budu spremnici nepromjenjivih objekata? Tada ne bismo dolazili do elemenata s dvama indeksiranjima, nego jednim — ali to nije problem jer možemo upotrijebiti *par* brojeva kao indeks. To i više nalikuje matematičkoj definiciji, gdje element matrice prirodno dobijemo indeksirajući matricu istodobno s dvama brojevima. Naravno, lista (niti ikakav niz) nam neće omogućiti da je indeksiramo parom brojeva, ali zato rječnik hoće: ključevi rječnika mogu biti bilo kakvi potpuno nepromjenjivi objekti.

```
>>> matrica = {}  
>>> matrica[5, 2] = 172  
>>> matrica[5, 2]  
172
```

Što smo izgubili? Na prvi pogled, dosta. Za početak, inicijalizacija rječnika ne daje nikakvu informaciju o dimenzijama matrice ni o početnim vrijednostima njezinih elemenata. Što se početnih vrijednosti tiče, ako nam je nula dovoljno dobra, možemo se koristiti rječnikom s podrazumijevanom vrijednošću:

```
>>> from collections import defaultdict  
>>> matrica = defaultdict(int) # ili npr. float za 0.0  
>>> matrica[12, 58]  
0
```

Dimenzije, ako su nam bitne za račun, možemo držati i u nekim varijablama sa strane (na što smo naviknuti). Problem je ispis, jer ispis rječnika nije baš prilagođen za čitanje kao što smo naviknuti čitati matricu — iako nije problem napisati funkciju koja prima rječnik i dimenzije i ispisuje ga koliko god lijepo želimo:

```
def ispiši_matricu(matrica, m, n, širina=3):  
    for i in range(m):  
        for j in range(n):  
            print(format(matrica[i, j], str(širina)), end=' ')  
        print()  
    Ipak, prilikom inicijalizacije nečeg što nije nulmatrica bilo  
    bi dobro napisati elemente izravno u konstruktor, a i ispis bi  
    ponekad bilo dobro imati u jednom redu...
```

Objektno orijentirani pristup

Zapravo, kad malo bolje razmislimo, lista listi fantastičan je format i za ulazne i za izlazne podatke, samo što nije baš za internu reprezentaciju (i to je vjerojatno razlog zbog kojeg se većina početnika muči upravo s tim pristupom). Možemo napisati funkcije za pretvorbu u obama smjerovima, ali sad već vidimo da imamo strukturu s članskim varijablama (dimenzije), specijalno formatiranim unosom i izlazom, vjerojatno jednog dana i operatorima (matrice znamo zbrajati, množiti,...) — drugim riječima, imamo *klasu*.

Dobra je vijest da Python ima vrlo snažnu podršku za objektno programiranje, s pomoću koje se ta klasa može prilično jednostavno implementirati. (Neke od upotrijebljenih

13. A sad, nešto sasvim drugačije

tehnika možda još niste obradili, ali odgovarajuće ključne riječi lako možete pogledati u Pythonovoj dokumentaciji.)

```
class matrix(dict):
    def __init__(self, *data):
        self.m, self.n = len(data), min(map(len, data))
        for i, row in enumerate(data):
            for j, number in enumerate(row):
                self[i, j] = number
    def __iter__(self):
        for i in range(self.m):
            yield [self[i, j] for j in range(self.n)]
```

I već možemo raditi sa svojim matricama, gotovo jednako kao što smo mogli i prije.

```
>>> A = matrix([1, 2], [3, 4])
>>> A[1, 0] = 8
>>> list(A)                                # print(*A, sep='\n') za 2D ispis
[[1, 2], [8, 4]]
```

Naravno, ima prostora za poboljšanje. Jednom kad imamo klasu, u njezino sučelje možemo ugraditi razne funkcije koje olakšavaju rad s njeziniminstancama. Primjerice, ako želimo lijep ispis koji se može nanovo evaluirati, dodamo (uvučeno ispod `class matrix`)

```
def __repr__(self): return 'matrix' + str(tuple(self))
```

Također, plitko kopiranje naših objekata sada radi, ali metodu `copy` naslijedili smo od rječnika pa ona vraća rječnik. Možemo upotrijebiti modul `copy` (gdje postoji i funkcija `copy` za plitko kopiranje), ali ako želimo imati sučelje kao ugrađeni spremnici, nije teško dodati pravu metodu za kopiranje:

```
def copy(self): return matrix(*self)
```

Kad već imamo toliko korisnih funkcija, mogli bismo i konstrukciju nulmatrice zadanih dimenzija izdvojiti u posebnu funkciju (primijetimo da više nije problem što u konstrukciji imamo ponovljene retke jer su u internoj reprezentaciji svi njihovi elementi na prvoj razini):

```
def nul(m, n): return matrix(*[[0] * n] * m)
```

Sada radi sljedeći kod:

```
>>> A, B = matrix([1, 2], [3, 4]), matrix.nul(2, 3)
>>> C, D = A.copy(), B.copy()
>>> B[1, 2] = C[1, 0] = A[1, 1]
>>> print(C, D)
matrix([1, 2], [4, 4]) matrix([0, 0, 0], [0, 0, 0])
```

Za kraj: pogledajte sami specijalne metode `__str__`, `__add__` i `__mul__` i s pomoću njih implementirajte dvodimenzionalni ispis, zbrajanje i množenje matrica. Također implementirajte metodu za transponiranje matrice. Razmislite o tome koje bi još funkcije mogle biti korisne.

13. A sad, nešto sasvim drugačije

13.3. Rekurzije – detalji i zanimljivosti

Uvod

Poznata je izreka "Da biste razumjeli rekurziju, morate prvo razumjeti rekurziju." Usprkos tomu što je očito šaljiva, ta izreka u dobroj mjeri upućuje na osnovno svojstvo rekurzivnih objekata, a to je da referiraju na sebe same. Vjerojatno su vam od takvih objekata najpoznatije rekurzivne *funkcije*, koje među naredbama koje izvršavaju imaju i poziv sebe samih.



Rekurzija M može se zapisati i na „školski“ način:

```
def M(n):
    if n > 100:
        return n-10
    else:
        return M(M(n+11))
```

```
def M(n):
    return n-10 if n > 100 else M(M(n+11))
```

Definirali smo funkciju M, koja unutar svojeg koda ima poziv funkcije M. Štoviše, taj će se poziv doista izvršiti, i to dvaput, kad god argument s kojim je funkcija pozvana nije veći od 100. Zato nije odmah jasno da će svaki prirodni broj poslan u funkciju uvjetovati njezino izvršavanje samo konačno mnogo puta.

Zadatak

Pokušajte ustanoviti kako zapravo djeluje funkcija M (napisati pravilo pridruživanja koje nije rekurzivno), barem na prirodnim brojevima. Pokušajte pokazati da, funkcija M na cijelim brojevima poštuje isto pravilo. Malo je zanimljivije njezino djelovanje na realnim brojevima, ali također ga nije teško utvrditi.

Granice rekurzije u Pythonu

Iako možemo dokazati da za matematičku funkciju *M*, definiranu pravilom analognim gore definiranoj Pythonovoј funkciji, vrijedi $M(-12345)=91$, Python to ne zna. Pokušamo li to izračunati, dobit ćemo zanimljivu poruku.

```
>>> M(-12345)
RecursionError: maximum recursion depth exceeded in ...
```

Granica dubine
rekurzije

Python (u uobičajenoj implementaciji) ima ugrađenu granicu dubine rekurzije, odnosno broja međusobno ugniježđenih funkcijskih poziva. U verziji 3.5 njezina je podrazumijevana vrijednost najčešće 1000, iako se može razlikovati s obzirom na operativni sustav i okruženje u kojem je Python instaliran. Čemu ta granica služi?

Kad god jedna funkcija poziva drugu, povratak iz druge funkcije mora prenijeti kontrolu toka izvršavanja na točno određeno mjesto unutar prve funkcije. To mjesto mora negdje biti zapamćeno za vrijeme izvršavanja druge funkcije, a ako ona poziva treću (čak i ako su prva, druga i treća zapravo ista funkcija), također mora biti zapamćeno mjesto u drugoj funkciji na koje se povratak iz treće treba vratiti. I tako dalje... svaki poziv treba zapamtiti jedno mjesto u kodu, ono neposredno nakon poziva, dok svaki povratak treba

13. A sad, nešto sasvim drugačije

“pokupiti” zadnju zapamćenu neupotrijebljenu lokaciju i “upotrijebiti” je, odnosno nastaviti izvršavati program od tog mesta nadalje.

Prirodna struktura podataka za to pamćenje jest stog jer će zadnja zapamćena lokacija biti upotrijebljena prva. Standardno, računala rezerviraju statički određeni segment svoje memorije za taj stog i popunjavaju ga redom. Tako je puno brže raditi sa stogom, ali stog sa sobom nosi jednu zamku: ako se prepuni, program se ruši (tzv. *stack overflow*), bez obzira na to što izvan stogova segmenta računalo ima još raspoložive memorije.

Stack overflow

Kako je jedan od ciljeva dizajna Pythona bio razumljivost početnicima, procijenjeno je da je rušenje procesa u slučaju prepunjivanja stoga neprihvatljivo. Zato je u Python ugrađena eksplicitna granica broja ugniježđenih poziva, čije prelaženje generira iznimku `RecursionError`. Granica je odabrana tako da ne utječe na većinu normalnih rekurzivnih funkcija pisanih u skladu s Pythonovom filozofijom, a da ipak bude dovoljno ispod granice na kojoj se proces ruši. Kako različiti operativni sustavi imaju stogove segmente različitih veličina, i kako nisu svi okviri iste veličine, još uvijek se tu i tamo dogodi da netko uspije srušiti proces, no to se smatra *bugom* i vrlo brzo ispravlja; ako nikako drugačije, smanjenjem granice.

Granica se može dobiti pozivom sugestivno imenovane funkcije u sistemskom modulu:

`getrecursionlimit()`

```
>>> import sys  
>>> sys.getrecursionlimit()  
1000
```

Dubina i širina rekurzije

Granica se može i *promijeniti* s pomoću funkcije `sys.setrecursionlimit`, ali to skoro nikada nije dobro rješenje. Povećanje preko sistemski postavljene granice izlaze proces većoj opasnosti od rušenja, dok smanjenje može čak onemogućiti uobičajeno funkcioniranje Pythona. Uzmite u obzir da je npr. IDLE, Pythonovo najpoznatije razvojno okruženje, napisano u Pythonu: dakle, sasvim je uobičajeno da, kad mislite da je stog izvršavanja prazan, ispod funkcije koju upravo izvršavate leži još nekoliko poziva funkcija koje se brinu za normalno funkcioniranje IDLE-a.

`setrecursionlimit()`

Ako dobijete poruku o iznimci `RecursionError`, najbitnije je da shvatite da je pogreška gotovo sigurno u načinu na koji ste napisali rekurzivnu funkciju, a ne u Pythonovim ograničenjima. Zašto? U osnovi, postoje dvije vrste rekurzivnih funkcija: one kod kojih svaki poziv rezultira još najviše jednim pozivom iste funkcije i one kod kojih se funkcija grana u više poziva same sebe u istom izvršavanju (osim, dakako, kad je ispunjen uvjet zaustavljanja rekurzije).

Rekurzivne funkcije prvog tipa gotovo je uvijek puno jednostavnije za čitatelja i manje opasno za računalo napisati iterativno: s pomoću petlji. Klasični je primjer faktorijel koji je bio jedan od početnih primjera prilikom uvođenja ideje rekurzivnog razmišljanja. Ali,

13. A sad, nešto sasvim drugačije

kako Python već ima tu funkciju implementiranu u standardnoj biblioteci (`math.factorial`), sada ćemo promotriti “dvofaktorijel” $n!!$, odnosno umnožak prirodnih brojeva do n koji su iste parnosti kao n . Jednostavan način da se to napiše u Pythonu jest

```
def factorial2(n):
    product = 1
    for factor in range(n, 0, -2):
        product *= factor
    return product
```

Iako postoje programski jezici koji preferiraju rekurzivna rješenja (kao LISP), Python nije jedan od njih. Autor Pythona, Guido van Rossum, u nekoliko je navrata rekao da nije zainteresiran za dodavanje optimizacija u Python koje bi omogućile rekurzivna rješenja jer bi time samo promicao pisanje nečitkog koda.



Jedan od problema s udžbenicima informatike jest što prezentiraju smiješne primjere rekurzije. Tipični su primjeri računanje faktorijela ili Fibonaccijevih brojeva. Rekurzija je moćan alat i doista je glupo upotrijebiti je u bilo kojem od tih slučajeva. Da programer koji radi za mene upotrijebi rekurziju za računanje faktorijela, zaposlio bih nekog drugog.

Steve McConnell, autor knjige *Kod iznutra (Code Complete)*

Rekurzivne funkcije drugog tipa, pak, nemaju problem s dubinom rekurzije iz vrlo jednostavnog razloga: širina rekurzije puno im je bitniji ograničavajući faktor. Prepostavimo da se svaki poziv koji nije završni grana u samo dva poziva. To bi značilo da na dubini gdje nam Pythonova granica postaje bitna imamo oko 2^{950} poziva, što je očita besmislica: taj broj veći je od broja elementarnih čestica u vidljivom svemiru.

Zaključak je da u ogromnom broju slučajeva kad pomislite da vam Pythonovo ograničenje kvari planove, tada ili imate neoptimalan kod koji je puno lakše i bolje napisati u obliku petlje ili imate rekurziju koja se neće stići izvršiti prije toplinske smrti svemira. McConnell s margine udžbenika daje “školske” primjere za svaki od tih slučajeva: faktorijel i Fibonaccijev niz. I jedno i drugo znači da trebate nešto redizajnirati: ili kod ili algoritam. Srećom, u tim dvama primjerima, a i u mnogim drugima, to doista nije teško.

Rekurzivni objekti

Objekti koji referiraju na sebe same ne moraju nužno biti funkcije. Vidjeli smo u priči “Ni liste nisu kutije” da je lista zapravo zbirka identiteta svojih elemenata, a ne njihovih vrijednosti. Iz toga, kao i iz činjenice da su liste promjenjivi objekti, slijedi da kao jedan od elemenata liste možemo staviti samu tu listu.

```
>>> selflist = [3, 5, 2]
>>> selflist.append(selflist)
>>> selflist
[3, 5, 2, [...]]# pomislili ste da je autor skratio zapisivanje elemenata liste?
```

13. A sad, nešto sasvim drugačije

Vidimo da Python čak zna ispisivati takve "rekurzivne objekte" na način da umjesto novog ugniježđenog ispisa liste koja se već ispisuje, stavi [...]. Slične manipulacije moguće su i s vrijednostima u rječniku: tada na mjestu rekurzivne vrijednosti dobijemo {...}. U mnogočemu je `selflist` sasvim obična lista, duljine 4, s elementom na indeksu 1 jednakim 5 itd. Možemo je čak i uspoređivati s drugim listama, ali pokušaj uspoređivanja s "jednako konstruiranom", ali ne istom listom dat će iznimku `RecursionError`.

`selflist`

Pokušajte plitko i duboko kopirati `selflist` (pogledajte priču "Duboko kopiranje"). Možete li objasniti rezultate koje ste dobili?

Ishodi i međupredmetne teme

Poglavlje	Ishodi učenja	Međupredmetne teme
1. Stringovi 2. Lista i lista 3. Rekurzije 4. Datoteke u programskom jeziku Python 5. Korisnička grafička sučelja 6. Koordinatna grafika 7. Složenost algoritama	<p>B.3.1. Koristeći neki grafički modul vizualizirati i grafički prikazati neki problem iz svoje okoline.</p> <p>B.3.1. Primijeniti standardne algoritme definirane nad cijelim brojevima.</p> <p>B.3.2. Rješavati problem primjenjujući složene tipove podataka definirane zadanim programskim jezikom.</p> <p>B.3.2. Analizirati sortiranje podataka kao važan koncept za rješavanje različitih problema.</p> <p>B.3.3. Rješavati problem primjenjujući rekursivnu funkciju.</p> <p>B.3.4. Usporedivati različite algoritme sortiranja i pretraživanja podataka.</p> <p>B.3.5. Vrednovati algoritme prema njihovoj vremenskoj složenosti.</p> <p>B.3.8. Definirati problem iz stvarnoga života i stvarati programsko rješenje prolazeći sve faze programiranja; predstaviti programsko rješenje ostalima i vrednovati ga.</p>	<p><i>Učiti kako učiti</i></p> <p>B.5.4. Samovrednovati proces učenja i svoje rezultate, procijeniti ostvareni napredak te na temelju toga planirati buduće učenje.</p> <p><i>Osobni i socijalni razvoj</i></p> <p>A.5.3. Razvijati svoje potencijale.</p> <p>B.5.2. Suradnički učiti i raditi u timu.</p> <p><i>Poduzetništvo</i></p> <p>B.5.1. Razvijati poduzetničku ideju od koncepta do realizacije.</p>
8. Zapis	B.3.6. Osmisliti objektni model s pri-padnim složenim strukturama podataka, implementira ga u zadanome programskom jeziku.	<p><i>Poduzetništvo</i></p> <p>B.5.2. Planirati i upravljati aktivnostima.</p>
9. Multimedijski projekt	C.3.1. Planirati, razvijati, stvarati, predstaviti i vrednovati multimedijski projekt.	<p><i>Uporaba IKT</i></p> <p>B.5.3. Promicati toleranciju, različitosti, međukulturno razumijevanje i demokratsko sudjelovanje u digitalnome okružju.</p> <p>C.5.4. Samostalno i odgovorno upravljati prikupljenim informacijama.</p> <p>D.5.1. Svrishodno primjenjivati vrlo različite metode za razvoj kreativnosti kombinirajući stvarno i virtualno okružje.</p> <p>D.5.3. Samostalno ili u suradnji s kolegama predočavati, stvarati i dijeliti nove ideje i uratke s pomoću IKT-a.</p> <p>D.5.4. Samostalno štititi svoje intelektualno vlasništvo i odabirati načine dijeljenja sadržaja.</p> <p><i>Osobni i socijalni razvoj</i></p> <p>B.5.2. Suradnički učiti i raditi u timu.</p> <p>C.5.2. Preuzimati odgovornost za pridržavanje zakonskih propisa te društvenih pravila i normi.</p> <p><i>Poduzetništvo</i></p> <p>B.5.1. Razvijati poduzetničku ideju od koncepta do realizacije.</p> <p>B.5.2. Planirati i upravljati aktivnostima.</p>

10. Baze podataka	A.3.1. Za jednostavni problem iz stvarnoga života oblikovati bazu podataka te ju realizirati u nekom sustavu za rad s bazama podataka.	<i>Uporaba IKT</i> D.5.2. Samostalno predlagati moguća i primjenjiva rješenja složenih problema s pomoću IKT-a. <i>Osobni i socijalni razvoj</i> C.5.2. Preuzimati odgovornost za pridržavanje zakonskih propisa te društvenih pravila i normi.
11. Izrada mrežnih stranica	A.3.1. Dizajnirati, razvijati i objaviti strukturu povezanih mrežnih stranica s pomoću alata i tehnologija koje se izvode na računalu korisnika.	<i>Učiti kako učiti</i> B.5.1. Samostalno odrediti ciljeve učenja, odabrati pristup učenju te planirati učenje. <i>Osobni i socijalni razvoj</i> B.5.2. Suradnički učiti i raditi u timu. <i>Poduzetništvo</i> B.5.1. Razvijati poduzetničku ideju od koncepta do realizacije.
12. Kriptografija i Tkinter	B.3.7. Analizirati tradicionalne kriptografske algoritme i opisati osnovnu ideju modernih kriptografskih sustava.	<i>Poduzetništvo</i> B.5.2. Planirati i upravljati aktivnostima.

O – Opće, jezične, klasične i prirodoslovne gimnazije

Kazalo

- __add__, 15, 245
- __mul__, 245
- __repr__, 245
- __str__, 245
- <body> tag, 216
- <head> tag, 216
- <html> tag, 216
- <iframe> tag, 223
- <title> tag, 216
- Atbash algoritam , 229
- Amélie film, 188
- append, 234
- asyncio modul, 241
- Audacity, 195, 227
- audio kanal (HitFilm Express), 193
- autentifikacija, 228
- baza podataka, 201
- baze podataka, 200-210
- Benfordov zakon, 153
- BFS pretraživanje, 90
- bilinearna interpolacija boja, 142
- binarno pretraživanje, 161
- bind(), 108
- bitrate, 227
- brisanje uzastopnih znakova, 26
- cache, 238
- cachiranje, 237
- Cezarov algoritam, 229
- Cezarova šifra, 229
- checkbutton(), 109
- chr(), 7
- close(), 94
- CMS sustav, 225
- colormode(), 139
- config(), 107, 109
- copy(), 240
- count(), 13
- Crop alat (GIMP), 190
- CSS, 227
- Data Base Management System, 201
- datoteke u Pythonu, 94-105
- DBMS, 201
- deepcopy(), 243
- delete (Tkinter), 110
- DFS pretraživanje, 89
- dir(), 9
- dizajn algoritma, 56
- duboko kopiranje objekta, 242
- dvodimenzionalni niz, 36
- embed, 222
- entry(), 109
- Euklidov algoritam, 63
- extend, 236
- Feather edges alat (GIMP), 189
- Fibonaccijevi brojevi, 60, 63, 76, 154, 248
- FIFO struktura, 89
- filedialog, 116
- fill (Tkinter), 111, 113
- find(), 16
- formati slika, 226
- formati zvuka, 227
- funkcije za string, 7
- Garbage Collector, 237
- geometry(), 107
- get() (Tkinter), 110, 112
- getrecursionlimit(), 247
- GIF, 226
- GIMP, 187, 226
- Google Sites, 225
- graf baze podataka, 200
- granica dubine rekurzije, 246
- grid(), 114
- hash(), 238
- hashiranje, 237, 238
- help(), 9
- hijerarhijski model podataka, 202
- HitFilm Express, 191
- home page, 217
- HTML jezik, 215
- HTML oznake, 216
- HTML tagovi, 216
- HTML5, 227
- HTTPS protokol, 228
- insert (Tkinter), 110
- insertion sort, 163
- izrada mrežnih stranica, 214-227
- JavaScript, 227
- jednostavna rekurzija, 67
- join(), 11
- JPEG, 226
- Kerckhoffsova doktrina, 229
- klasa, 178, 244
- klasa, 244
- Kochova krivulja, 128
- Kochova pahuljica, 128
- konstantna složenost algoritma, 160
- koodinatna grafika, 122-150
- kopija liste, 52
- kopiranje matrice, 240
- korisnička grafička sučelja, 106-115
- kreiranje tablica, 205
- kretanje u svim smjerovima matrice, 44
- criptoanaliza, 228
- criptografija, 228-232
- criptografija, 228
- criptografski algoritam, 228
- criptologija, 228
- kvadratična složenost algoritma, 160

Kazalo

- L.insert(), 161
Label(), 107
len(), 7
LIFO struktura, 88
LIKE, 210
linearna
 interpolacija boja, 140
linearna
 složenost algoritma, 160
link, 214
lista, 30
lista listâ, 30-37
lista lista kao pomoćna
 struktura, 40
lista red, 89
lista stog, 88
listen(), 136

mainloop(), 107
maksimum liste, 35
matrica, 36
matrice u matematici, 36
max(), 7
memoizacija, 82-84
Menu(), 113
merge sort, 163
messagebox(), 115
metoda brute-force, 169
metoda grube sile, 169
metode za
 rad s datotekama, 94
metode za string, 44082
min(), 7
MP3, 227
MP4, 227
mrežna stranica, 214
mrežni model podataka, 202
mrežni preglednik, 214
mrežno mjesto, 214

NULL, 210

objektni
 model podataka, 202
obrada
 zvuka (Audacity), 196
čuvanje tajnosti
 podataka, 228

ODER BY naredba, 209
Ogg, 227
onclick(), 135
open(), 94
operator konkatenacije, 6
operator repliciranja, 6
ord(), 7
osiguranje integriteta
 podataka, 228
otvaranje datoteke, 94

pack(), 109, 113
Paint.NET, 226
partition, 10
petlja događaja, 241
PhotoImage(), 107
PhotoScape, 226
pivot liste, 166
place(), 114
Playfairova šifra, 231
plitka kopija objekta, 240
PNG, 226
poslužitelj, 225
poveznica, 214
pravilo referencijskog
 integriteta, 203
prebrajanje, 42
pretraživanje u dubinu, 89
pretraživanje u širinu, 90
primarni ključ, 202

quick sort, 166

rad s više pera, 137
radiobutton() (Tkinter), 112
randint(), 138
random modul, 137
random.choice(), 158
read(), 94
readline(), 94-95
rekurzija s dva parametra, 72
rekurzije, 58-99
rekurzivna funkcija, 60-61
rekurzivna relacija, 59
rekurzivni poziv
 u rekurziji, 70
relacija, 202

relacijske baze podataka,
 200, 202-203
relacijski model
 podataka, 202
replace(), 18
repr(), 7
resizable(), 107
rezolucija slike, 226
RGB model, 138

sakupljač smeća, 237
Scissors select
 alat (GIMP), 189
scrollbar (Tkinter), 110
seek(), 95
SELECT naredba, 209
selflist, 249
servisi za generiranje
 embed koda, 224
setrecursionlimit(), 247
Shannonova maksima, 229
side (Tkinter), 111, 114
složena simulacija, 28
složenost
 algoritama, 156-168
složenost algoritma
 merge sort, 165
složenost algoritma
 quick sort, 168-169
složenost
 funkcije insert(), 162
split(), 18
SQL upiti, 208
SQLite Viewer, 206
SQLite3, 203
SSH protokol, 228
SSL/TLS protokol, 228
stack, 88
stack overflow, 247
stog, 88
str(), 7
strani ključ, 203
string, 6
stringovi, 6-19
supstitucijske šifre, 229
sistem za upravljanje bazom
 podataka, 201
swapcase(), 11

Kazalo

- tell(), 95
tijek izvršavanja rekurzije, 62
time.clock(), 157
title(), 107
tk(), 106
tkinter modul, 106
Toplevel(), 112
tornjevi Hanoja, 65
transponiranje liste, 36
Trimmer
 (HitFilm Express), 192
turtle modul, 122
 back(), 122
 backward(), 122
 begin_fill(), 123
 bk(), 122
 bye(), 123
 circle(), 122, 126
 clear(), 123
 color(), 123
 distance(), 123
 distance(), 123
 dot(), 122
 down(), 123
 end_fill(), 123
 exitonclick(), 123
 fd(), 122
 fillcolor(), 123
 filling(), 123
 forward(), 122
 goto(), 122
 heading(), 122
 hideturtle(), 123
 home(), 122
 ht(), 123
 isdown(), 123
 isvisible(), 123
 left(), 122
 lt(), 122
numinput(), 123, 125
onclick(), 123
onrelease(), 123
pd(), 123
pen(), 123
pencolor(), 123
pendown(), 123
pensize(), 123
penup(), 123
pos(), 122
position(), 122
pu(), 123
reset(), 123
right(), 122
rt(), 122
seth(), 122
setheading(), 122
setpos(), 122
setposition(), 122
setx(), 122
sety(), 122
showturtle(), 123
speed(), 122
st(), 123
stamp(), 123
textinput(), 123
title(), 123
towards(), 122
tracer(), 123
undo(), 122
up(), 123
width(), 123
write(), 123
xcor(), 122
ycor(), 122
ugradnja multimedijskog
 sadržaja u HTML
 dokument, 222
umetanje poveznica u HTML
 dokument, 220
umetanje slike u HTML
 dokument, 219
URL adresa, 214
uvjet završetka rekurzije, 59
vanjski ključ, 203
video kanal (HitFilm
 Express), 193
Vigenereov kvadrat, 231
Vigenereova šifra, 230
Vimeo, 227
vremenska složenost
 algoritma, 156
WAV, 227
web hosting, 225
web-stranica, 214
Weebly, 225
WHERE naredba, 207
WIX, 225
WordPress, 225
write(), 96
writeln(), 97
WYSIWYG uređivač, 225
YouTube, 227
zapis kao klasa, 178
zapis u Pythonu, 178-181
zatvaranje datoteke, 94
učitavanje i ispisivanje liste
 lista, 33
učitavanje liste lista, 32