METODIKA NASTAVE RAČUNARSTVA A

Python – razni zadaci za vežbanje

Stefan Mišković

1. Data su dva kompleksna broja *z*1 = *a*1 + *b*1*i* i *z*2 = *a*2 + *b*2*i*. Odrediti njihov zbir i proizvod.

Zbir dva kompleksna broja *z*1 = *a*1 + *b*1*i* i *z*2 = *a*2 + *b*2*i* je *z*1 + *z*2 = (*a*1 + *a*2) + (*b*1 + *b*2)*i*, a proizvod

*z*1*z*2 = (*a*1*a*2 − *b*1*b*2) + (*a*1*b*2 + *a*2*b*1)*i*.

**class** KompleksanBroj :

**def**

\_\_init\_\_( s e l f , r , i ) : s e l f . r = r

s e l f . i = i

**def** i s p i s ( s e l f ) :

**print** ( "%d.+.%di " % ( s e l f . r , s e l f . i ))

**def** z b i r ( s e l f , z2 ) :

zbir R = s e l f . r + z2 . r z b i r I = s e l f . i + z2 . i

**return** KompleksanBroj ( zbirR , z b i r I )

**def** proizvod ( s e l f , z2 ) :

proizvod R = s e l f . r z2 . r s e l f . i z2 . i p r o i z v o d I = s e l f . r z2 . i + s e l f . i z2 . r

∗ ∗

∗ − ∗

**return** KompleksanBroj ( proizvodR , p r o i z v o d I )

z1 = KompleksanBroj ( 2 , 4 ) z2 = KompleksanBroj ( 3 , 5 ) z b i r = z1 . z b i r ( z2 )

proizvod = z1 . proizvod ( z2 ) z b i r . i s p i s ( )

proizvod . i s p i s ( )

1. Za dva prirodna broja *m* i *n*, odrediti njihov najveći zajednički delilac i najmanji zajednički sadržalac.

Za najveći zajednički delilac se koristi rekurentna relacija nzd(x, y) = nzd(y, x mod y), a za naj- manji zajednički sadržalac činjenica da je proizvod nzd(x, y) i nzs(x, y) jednak proizvodu brojeva *x* i *y*.

**def** nzd ( x , y ) :

**while** y > 0 :

x , y = y , x % y

**return** x

**def** nzs ( x , y ) :

z = ( x y ) // nzd ( x , y )

∗

**return** z a = 12

b = 8

**print** ( nzd ( a , b ) )

**print** ( nzs ( a , b ) )

1. Zadat je broj u osnovi *k*. Prevesti ga u dekadnu osnovu. Obratno, dati broj u dekadnoj osnovi prevesti u zapis sa osnovom *k*.

Broj se *an . . . a*1*a*0 se iz osnove *k* prevodi u dekadnu osnovu prema obrascu Σ*n aiki*. Pri prevođenju

*k*=0

iz dekadne osnove se broj u svakom koraku deli osnovom *k*, a trenutna cifra rezultujućeg broja je ostatak pri deljenju trenutne vrednosti *n* sa *k*. U oba smera je korišćena činjenica da se za veće cifre od 9, koriste oznake A, B, C, D, E i F (uzeto je da je 2 ≤ *k* ≤ 16).

**def** iz Dekadne ( n , osnova ) : prevod = ""

c i f r e = "ABCDEF"

**while** n > 0 :

c = **str** ( n % osnova )

**i f** ( n % osnova >= 1 0 ) :

c = c i f r e [ n % osnova 1 0 ] prevod = c + prevod

−

n = n // osnova

**return** prevod

**def** uDekadnu ( broj , osnova ) : prevod = 0

n = **len** ( b r o j )

**for** i **in range** ( n ) :

r e c n i k = {"A" : 1 0 , "B" : 1 1 , "C" : 1 2 , "D" : 1 3 , "E" : 1 4 , "F" : 15 }

c i f r a = 0

**i f** b r o j [ i ] **in** r e c n i k . keys ( ) : c i f r a = r e c n i k [ b r o j [ i ] ]

**else** :

c i f r a = **int** ( b r o j [ i ] )

prevod += c i f r a osnova ( n 1 i )

∗ ∗∗ − −

**return** prevod

**print** ( iz Dekadne ( 717 , 16 ))

**print** ( uDekadnu ( "2CD" , 16 ))

1. Za prirodne brojeve *n* i *k*, odrediti vrednost stepena *nk* sa što manje množenja (broj množenja bi trebalo da bude približno jednak log2 *k*).

Stepenovanje se može uraditi u znatno manjem broju koraka umesto da se *k* puta množi broj *n* sa samim sobom. Pritom za koristi činjenica da je za paran broj *k* ispunjeno *nk* = *nk/*2*nk/*2, a za neparan broj *k* važi *nk* = *nk/*2*nk/*2*n*. U oba slučaja se u stepenu radi o celobrojnom deljenju.

**def** stepen ( n , k ) : r e z u l t a t = 1 **while** k > 0 :

**i f** k % 2 == 0 : k = k / 2 n = n n

:

∗

**else**

k = k − 1

r e z u l t a t = r e z u l t a t ∗ n

**return**

k = k / 2 n = n n r e z u l t a t

**print** ( stepen ( 2 , 4 ))

∗

**print** ( stepen ( 12 , 3 ) == 12 ∗∗ 3 )

1. Za dati prirodan broj *n*, izračunati vrednost izraza *n*! mod 1000.

Imajući u vidu da je *n*! = (*n* 1)!*n*, faktorijel se može izračunati jednostavno u *n* iteracija, vodeći računa o traženom modulu.

−

**def** f a c t o r i a l ( n ) : p = 1

**for** i **in range** ( n ) :

p = p ( i + 1 ) % 1000

∗

**return** p

**print** ( f a c t o r i a l ( 5 ) ) **print** ( f a c t o r i a l ( 7 ) ) **print** ( f a c t o r i a l ( 1 3 ))

1. Niz *f* (*n*) je definisan na sledeći način: *f* (0) = 0, *f* (1) = 1 i *f* (*n*) = *af* (*n* 1) + *bf* (*n* 2). Odrediti vrednost *f* (*n*).

− −

Niz se računa slično nalaženju *n*-tog Fibonačijevog broja, imajući pritom i u vidu parametre *a* i *b* u rekurzivnoj relaciji.

**def** f ( n , a , b ) : prethodni = 0 tr e n u tn i = 1

i = 2

**while** i <= n :

novi = a tr e n u tn i + b prethodni prethodni = tr e n u tn i

∗ ∗

tr e n u tn i = novi i += 1

**return** tr e n u tn i

**print** ( f ( 10 , 1 , 1 ))

**print** ( f ( 2 , 4 , 7 ))

1. Za dati niz, odrediti vrednost razlike maksimalnog i minimalnog elementa.

I minimum i maksimum niza se mogu odrediti jednostavnom linearnom pretragom.

n i z = [ 1 3 , 2 , 11 , 5, 100 , 34 , 62 , 4 , 1 , 15]

− −

**min** = n i z [ 0 ] **max** = n i z [ 0 ] **for** i **in** n i z :

**i f** i < **min** :

**min** = i

**i f** i > **max**:

**max** = i

**print** (**max** − **min**)

1. Odrediti zbir cifara broja koje su deljive sa 3.

Cifre broja *n* se redom određuju računanjem modula po broju 10, a u svakom koraku se skida trenutna poslednja cifra broja *n* celobrojnim deljenjem sa 10.

**def** suma ( n ) : s = 0

**while** n > 0 :

c i f r a = n % 10

**i f** c i f r a % 3 == 0 : s += n % 10

n //= 10

**return** s

**print** ( suma ( 4 1 2 3 ))

**print** ( suma ( 9632210 ))

1. Ispitati da li je data niska palindrom.

Krećući će istovremeno sa obe strane, string nije palindrom ukoliko se naiđe na dva različita karaktera, a inače jeste.

**def** palindrom ( s ) :

**for** i **in range** ( 0 , **len** ( s ) / 2 ) :

**i f** s [ i ] != s [ **len** ( s ) i 1 ] :

− −

**return** False

**return** True

**print** ( palindrom ( " ana" ) )

**print** ( palindrom ( " jana " ))

1. Odrediti vrednost traga matrice.

Trag matrice *M* čine svi elementi oblika *Mii*, koji su na glavnoj dijagonali.

matrica = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] , [ 7 , 8 , 9 ] ]

trag = 0

**for** i **in range** ( **len** ( matrica ) ) : trag += matrica [ i ] [ i ]

**print** ( trag )

1. Za datu matricu *M* i prirodan broj *k*, odrediti matricu *M k*.

Osnov za stepenovanje matrica je jedinična matrica. Sporije stepenovanje koristi činjenicu da je *M k* =

*M k−*1*M* , a za brže se vrši polovljenje stepena *k* na sličan način kao što se to radi u skalarnom slučaju.

**import** numpy as np

**def** s te p e n S p o r i j e ( matrica , k ) :

r e z u l t a t = np . eye ( **len** ( matrica ) )

**for** i **in range** ( k ) :

r e z u l t a t = np . dot ( r e z u l ta t , matrica )

**return** r e z u l t a t

**def** stepen Brze ( matrica , k ) :

r e z u l t a t = np . eye ( **len** ( matrica ) )

**while** k > 0 :

**i f** k % 2 == 0 :

k = k / 2

matrica = np . dot ( matrica , matrica )

**else** :

k = k 1

−

r e z u l t a t = np . dot ( r e z u l ta t , matrica ) k = k / 2

matrica = np . dot ( matrica , matrica )

**return** r e z u l t a t

matrica = np . array ( [ [ 1 , 2 ] , [ 3 , 4 ] ] ) **print** ( s te p e n S p o r i j e ( matrica , 4 )) **print** ( stepen Brze ( matrica , 4 ))

1. Odrediti redni broj vrste u matrici čiji je zbir maksimalan.

Ovde se koristi funkcija sum iz numpy modula radi kompaktnijeg zapisa.

**import** numpy as np

matrica = np . array ( [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] , [ 7 , 8 , 9 ] ] )

**max** = np . **sum**( matrica [ 0 ] ) best = 0

**for** i **in range** ( 1 , **len** ( matrica ) ) : s = np . **sum**( matrica [ i ] )

**i f** s > **max**:

**max** = s best = i

**print** ( i + 1 )

1. Sa ulaza se učitavaju koordinate temena trougla. Odrediti njegov obim i površinu.

√ − − −

Ako su stranice trougla *a*, *b* i *c*, obim trougla je *O* = *a* + *b* + *c*, a površina *P* = *s*(*s a*)(*s b*)(*s c*), gde je *s* = *O/*2 poluobim (Heronov obrzac).

x1 , y1 , x2 , y2 , x3 , y3 = [ **int** ( i ) **for** i **in input** ( ) . s p l i t ( ) ] a = (( x1 x2 ) 2 + ( y1 y2 ) 2 ) 0 . 5

− ∗∗ − ∗∗ ∗∗

b = ( ( x1 x3 ) 2 + ( y1 y3 ) 2 ) 0 . 5

− ∗∗ − ∗∗ ∗∗

c = (( x2 x3 ) 2 + ( y2 y3 ) 2 ) 0 . 5

− ∗∗ − ∗∗ ∗∗

**print** ( a , b , c ) O = a + b + c s = O / 2

P = ( s ( s a ) ( s b ) ( s c )) 0 . 5

∗ − ∗ − ∗ − ∗∗

**print** ( "O.=." + **str** (O))

**print** ( "P.=." + **str** (P))

1. Sa standardnog ulaza se najpre učitava broj *n*, koji predstavlja broj temena konveksnog poligona. U narednih *n* redova se učitavaju koordinate svakog temena u smeru kazaljke na satu. Odrediti njegov obim i površinu.

Obim konveksnog mnogougla je jednak zbiru njegovih stranica. Ako su koordinate temena konveksnog mnogougla (*x*1*, y*1)*,* (*x*2*, y*2)*, ,* (*xn, yn*), vrednost površine je

*P* = . (*x y*

1

+ *x y*

+ · · · + *x*

1

*y* ) − (*x y*

+ *x y*

+ · · · + *x y*

). *.*

. 2 1 2 2 3

*n* 1 2 2 1 3 2

1 *n* .

n = **int** ( **input** ( ) ) x , y = [ ] , [ ]

**for** i **in range** ( n ) :

xi , y i = [ **int** ( t ) **for** t **in input** ( ) . s p l i t ( ) ] x . append ( x i )

y . append ( y i )

O, P = 0 , 0

**for** i **in range** ( n ) :

O += ( ( x [ i ] x [ ( i + 1 ) % n ] ) 2 + ( y [ i ] y [ ( i + 1 ) % n ] ) 2 ) 0 . 5

− ∗∗ − ∗∗ ∗∗

P += 0 . 5 ( x [ i ] y [ ( i + 1 ) % n ] y [ i ] x [ ( i + 1 ) % n ] )

∗ ∗ − ∗

**print** ( "O.=." + **str** (O))

**print** ( "P.=." + **str** ( **abs** (P) ) )

1. Metodom polovljenja intervala, odrediti nulu funkcije cos *x* na intervalu [0*,* 2].

Za interval [*a, b*] se u svakom koraku određuje srednji element intervala *c*. Ako su *f* (*a*) i *f* (*c*) istog znaka, razmatra se interval [*c, b*], a ako su različitog, interval [*a, c*].

**import** math

**def** b i s e k c i j a ( l e v i , d e s n i ) :

**for** i **in range** ( 1 0 0 ) :

s r e d n j i = ( l e v i + d e s n i ) / 2 . 0

**i f** math . cos ( l e v i ) math . cos ( s r e d n j i ) >= 0 : l e v i = s r e d n j i

∗

**else** :

d e s n i = s r e d n j i

**return** s r e d n j i

x = b i s e k c i j a ( 0 , 2 )

**print** ( x )

1. Odrediti indeks prvog elementa u nizu čija je vrednost manja od 0.

Čim se naiđe na negativan element niza, ciklus se prekida i vraća vrednost True. Inače, povratna vrednost funkcije je False.

**def** indeks ( n i z ) :

**for** i **in range** ( **len** ( n i z ) ) :

**i f** n i z [ i ] < 0 :

**return** i

**return** −1

n i z = [ 2 , 3 , 4 , 1 , 1, 5 , 6 ]

−

i = indeks ( n i z )

**i f** i == 1:

−

**print** ( " Svi . e l e m e n t i . su . n e n e g a t i v n i " )

**else** :

**print** ( i )

1. Ispitati da li su dve niske anagrami.

Kreiraju se dva rečnika, čiji su ključevi slova engleske abecede, a vrednosti broj njihovih pojavljivanja u rečima. Ako su ta dva rečnika jednaka, niske su anagrami.

**def** anagrami ( s1 , s 2 ) :

a l f a b e t = " abcdefghijklmnopqrstuvwxyz "

recnik 1 , r e c n i k 2 = {} , {}

**for** c **in** a l f a b e t :

r e c n i k 1 [ c ] = 0 r e c n i k 2 [ c ] = 0

**for** c **in** s 1 . lower ( ) : r e c n i k 1 [ c ] += 1

**for** c **in** s 2 . lower ( ) : r e c n i k 2 [ c ] += 1

**return** r e c n i k 1 == r e c n i k 2

s 1 = " vrata " s 2 = " Trava" s 3 = " brava "

**print** ( anagrami ( s1 , s 2 ) )

**print** ( anagrami ( s2 , s 3 ) )

1. Odrediti vrednost u nizu koja se pojavila najveći broj puta.

Nakon sortiranja niza, dovoljno je proći kroz niz i naći najveći broj pojavljivanja susednih elemenata.

n i z = [ 1 , 2 , 11 , 5 , 4 , 13 , 4 , 13 , 5 , 19 , 1 3 ]

−

prethodni = n a j b o l j i = None p o n a v l ja n je = **max** = 0

**for** tr e n u tn i **in sorted** ( n i z ) :

**i f** prethodni == tr e n u tn i : p o n a v l ja n je += 1

**else** :

p o n a v l ja n je = 0

**i f** p o n a v l ja n je > **max**: **max** = p o n a v l ja n je

n a j b o l j i = tr e n u tn i prethodni = tr e n u tn i

**print** ( n a j b o l j i )

1. Napisati metod kojim se dodaje čvor u listi nakon čvora sa unapred zadatom vrednošću.

Ovde je korišćen modul gde je implementirana prethodno povezana lista (zadatak iz dela o strukturama podataka). Argument metoda koji dodaje novi čvor je i sama lista. Kroz listu se iterira sve dok se naiđe element sa traženom vrednošću. Kada se takav element pojavi, nakon njega se, menjanjem vrednosti za sledei, ubacuje novi element na odgovarajuće mesto.

**from** l i s t a **import** Povezana Lista

**from** l i s t a **import** Cvor

**def** dodaj ( l i s t a , broj , vrednost ) : tr e n u tn i = l i s t a . glava

**while** tr e n u tn i != None :

**i f** tr e n u tn i . vrednost == vrednost : novi = Cvor ( b r o j )

novi . s l e d e c i = tr e n u tn i . s l e d e c i tr e n u tn i . s l e d e c i = novi

**return** True

tr e n u tn i = tr e n u tn i . s l e d e c i

**return** False

l i s t a = Povezana Lista ( )

l i s t a . dodaj ( 7 ) l i s t a . dodaj ( 4 )

l i s t a . dodaj ( 15 ) l i s t a . i s p i s i ( )

dodaj ( l i s t a , 13 , 4 ) l i s t a . i s p i s i ( )

dodaj ( l i s t a , 2 , 15 ) l i s t a . i s p i s i ( )

dodaj ( l i s t a , 9 , 7 ) l i s t a . i s p i s i ( )

1. Napisati funkciju koja objedinjuje dva sortirana niza. Pritom se podrazumeva da je rezultujući niz sortiran.

Iterira se kroz oba niza istovremeno, počev od prvog elementa. U svakom koraku se u novi niz ubacuje veći element, nakon čega se prelazi na naredni element u nizu iz koga je element prebačen u novi niz. Ukoliko se iteriranje kroz jedan niz završi ranije, svi elementi iz preostalog niza se jednostavno dodaju na kraj.

**def** s p o j ( niz 1 , n i z 2 ) : n1 = **len** ( niz 1 )

n2 = **len** ( niz 2 )

niz 3 = [ None ] ( n1 + n2 ) i = j = k = 0

∗

**while** i < n1 **and** j < n2 :

**i f** niz 1 [ i ] < niz 2 [ j ] : niz 3 [ k ] = niz 1 [ i ] k = k + 1

i = i + 1

**else** :

niz 3 [ k ] = niz 2 [ j ] k = k + 1

j = j + 1

**while** i < n1 :

niz 3 [ k ] = niz 1 [ i ] ; k = k + 1

i = i + 1

**while** j < n2 :

niz 3 [ k ] = niz 2 [ j ] ; k = k + 1

j = j + 1

**return** niz 3 niz 1 = [ ]

niz 2 = [ ]

**for** i **in range** ( 1 0 ) :

**i f** i % 2 == 0 :

niz 1 . append ( i )

**else** :

niz 2 . append ( i )

**print** ( niz 1 )

**print** ( niz 2 )

**print** ( s p o j ( niz 1 , n i z 2 ) )

1. Napisati funkciju koja objedinjuje dve sortirane liste.

Objedinjavanje dve sortirane liste se radi slično kao kod nizova. Ovde je jedina razlika što se na naredni element prelazi, imajući u vidu na šta pokazuje promenljiva sledeci.

**from** l i s t a **import** Povezana Lista

**def** s p o j ( l i s t a 1 , l i s t a 2 ) : i = l i s t a 1 . glava

j = l i s t a 2 . glava

l i s t a = Povezana Lista ( )

**while** i **and** j :

**i f** i . vrednost < j . vrednost : l i s t a . dodaj ( i . vrednost ) i = i . s l e d e c i

**else** :

l i s t a . dodaj ( j . vrednost ) j = j . s l e d e c i

**while** i :

l i s t a . dodaj ( i . vrednost ) i = i . s l e d e c i

**while** j :

l i s t a . dodaj ( j . vrednost ) j = j . s l e d e c i

**return** l i s t a

l i s t a 1 = Povezana Lista ( ) l i s t a 2 = Povezana Lista ( )

**for** i **in range** ( 1 0 ) :

**i f** i % 2 == 0 :

l i s t a 1 . dodaj ( i )

**else** :

l i s t a 2 . dodaj ( i )

l i s t a 1 . i s p i s i () l i s t a 2 . i s p i s i ()

s p o j ( l i s t a 1 , l i s t a 2 ) . i s p i s i ( )

1. Napisati program koji proverava da li su zagrade u izrazu dobro uparene. Koristiti stek.

Ovde su korišćene tri standardne vrste zagrada, a pretpostavlja se da se razmatra ispravan matematički izraz, kome su eventualno zagrade pogrešno napisane. Kad god se naiđe na otvorenu zagradu, ona se dodaje na stek. Kada se naiđe na zatvorenu zagradu, proverava se da li je na vrhu steka bila odgovarajuća otvorena. Ako nije, zagrade nisu dobro uparene. Ukoliko je stek prazan na kraju ili kada se naiđe na zatvorenu zagradu, tada takođe izraz nije dobro definisan. U prvom slučaju postoji više otvorenih zagrada u izrazu. U drugom slučaju za datu zatvorenu zagradu nije postojala odgovarajuća otvorena.

**from** s te k **import** Stek

**def** uparene ( i z r a z ) : s te k = Stek ( ) **for** c **in** i z r a z :

**i f** c **in** " { [ ( " :

s te k . dodaj ( c )

**e l i f** c **in** " } ) ] " :

**i f** s te k . prazan ( ) :

**return** False zagrada = s te k . o b r i s i ( )

**i f** zagrada + c **not in** [ "{}" , " [ ] " , " ( ) " ] :

**return** False

**i f not** s te k . prazan ( ) :

**return** False

**return** True

i z r a z 1 = "{[7+3]∗(2 −6) −7}+9 "

i z r a z 2 = "((7+3)∗(2 −6) −7)+9)"

i z r a z 3 = "{(7+3)∗(2 −6) −7]+9 " i z r a z 4 = "{(7+3)∗(2 −6)−7+9 "

**print** ( uparene ( i z r a z 1 ) ) **print** ( uparene ( i z r a z 2 ) ) **print** ( uparene ( i z r a z 3 ) ) **print** ( uparene ( i z r a z 4 ) )

1. Napisati funkciju koja određuje vrednost najvećeg elementa u stablu.

Kako se radi o binarnom pretraživačkom stablu, za vrednost najvećeg elementa je potrebno kretati se po desnim čvorovima, dok god je to moguće.

**from** s ta b l o **import** BinarnoDrvo

**def** n a jv e c i ( s ta b l o ) :

tr e n u tn i = s ta b l o . koren

**while** tr e n u tn i . d e s n i :

tr e n u tn i = tr e n u tn i . d e s n i

**return** tr e n u tn i . vrednost

s ta b l o = BinarnoDrvo ( ) s ta b l o . dodaj ( 8 )

s ta b l o . dodaj ( 4 ) s ta b l o . dodaj ( 17 ) s ta b l o . dodaj ( 16 ) s ta b l o . dodaj ( 3 ) s ta b l o . dodaj ( 10 ) s ta b l o . dodaj ( 12 )

**print** ( n a jv e c i ( s ta b l o ))

1. Napisati funkcju koja određuje vrednost najmanjeg elementa u stablu.

Kako se radi o binarnom pretraživačkom stablu, za vrednost najmanjeg elementa je potrebno kretati se po levim čvorovima, dok god je to moguće.

**from** s ta b l o **import** BinarnoDrvo

**def** najmanji ( s ta b l o ) :

tr e n u tn i = s ta b l o . koren

**while** tr e n u tn i . l e v i :

tr e n u tn i = tr e n u tn i . l e v i

**return** tr e n u tn i . vrednost

s ta b l o = BinarnoDrvo ( ) s ta b l o . dodaj ( 8 )

s ta b l o . dodaj ( 4 ) s ta b l o . dodaj ( 17 ) s ta b l o . dodaj ( 16 ) s ta b l o . dodaj ( 3 ) s ta b l o . dodaj ( 10 ) s ta b l o . dodaj ( 12 )

**print** ( najmanji ( s ta b l o ) )

1. Odrediti maksimalnu dubinu stabla.

Ukoliko se naiđe na čvor None, povratna vrednost rekurzije je 0. Inače, za svaki čvor se rekurzivno razmatra povratna vrednost funkcije za levo podstablo *l* i za desno podstablo *d*. Vrednost funkcije za taj čvor iznosi max{*l* + 1*, d* + 1}.

**from** s ta b l o **import** BinarnoDrvo

**from** s ta b l o **import** Cvor

**def** dubinaPomocna ( cvor ) :

**i f not** cvor :

**return** 0

l e v o = dubinaPomocna ( cvor . l e v i ) + 1 desno = dubinaPomocna ( cvor . d e s n i ) + 1 **i f** l e v o > desno :

**return** l e v o

**else** :

**return** desno

**def** dubina ( s ta b l o ) :

**return** dubinaPomocna ( s ta b l o . koren )

s ta b l o = BinarnoDrvo ( ) s ta b l o . dodaj ( 8 )

s ta b l o . dodaj ( 4 ) s ta b l o . dodaj ( 17 ) s ta b l o . dodaj ( 16 ) s ta b l o . dodaj ( 3 ) s ta b l o . dodaj ( 10 ) s ta b l o . dodaj ( 12 )

**print** ( dubina ( s ta b l o ) )

1. Implementirati funkcije za prefiksni i postfiksni ispis stabla.

Ispis se vrši slično kao i kod slučaja infiksnog ispisa, s tim što se kod prefiksnog prvo ispisuje čvor, a zatim funkcija rekurzivno poziva za levo i desno podstablo, a kod podfiksnog se čvor ispisuje nakon rekurzivnih poziva.

**from** s ta b l o **import** BinarnoDrvo

**from** s ta b l o **import** Cvor

**def** prefiksni Pomocna ( cvor ) :

**i f not** cvor :

**return**

**print** ( cvor . vrednost , end = "." ) prefiksni Pomocna ( cvor . l e v i ) prefiksni Pomocna ( cvor . d e s n i )

**def** p r e f i k s n i ( s ta b l o ) : prefiksni Pomocna ( s ta b l o . koren ) **print** ( )

**def** postfiksni Pomocna ( cvor ) :

**i f not** cvor :

**return**

postfiksni Pomocna ( cvor . l e v i ) postfiksni Pomocna ( cvor . d e s n i ) **print** ( cvor . vrednost , end = "." )

**def** p o s t f i k s n i ( s ta b l o ) : postfiksni Pomocna ( s ta b l o . koren ) **print** ( )

s ta b l o = BinarnoDrvo ( ) s ta b l o . dodaj ( 8 )

s ta b l o . dodaj ( 4 ) s ta b l o . dodaj ( 17 ) s ta b l o . dodaj ( 16 ) s ta b l o . dodaj ( 3 ) s ta b l o . dodaj ( 10 ) s ta b l o . dodaj ( 12 )

p r e f i k s n i ( s ta b l o ) p o s t f i k s n i ( s ta b l o )

1. Odrediti broj listova stabla.

List stabla je svaki čvor koji nema potomke. Za sve takve čvorove, povratna vrednost funkcije je 1. Kada se naiđe na čvor None, povratna vrednost funkcije je 0. Inače, za unutrašnje čvorove gleda se zbir za levo i desno podstablo.

**from** s ta b l o **import** BinarnoDrvo

**from** s ta b l o **import** Cvor

**def** l i stovi Pomocna ( cvor ) :

**i f not** cvor :

**return** 0

**i f not** cvor . l e v i **and not** cvor . d e s n i :

**return** 1

**return** l i stovi Pomocna ( cvor . l e v i ) + l i stovi Pomocna ( cvor . d e s n i )

**def** l i s t o v i ( s ta b l o ) :

**return** l i stovi Pomocna ( s ta b l o . koren ) s ta b l o = BinarnoDrvo ( )

s ta b l o . dodaj ( 8 ) s ta b l o . dodaj ( 4 ) s ta b l o . dodaj ( 17 ) s ta b l o . dodaj ( 16 ) s ta b l o . dodaj ( 3 ) s ta b l o . dodaj ( 10 ) s ta b l o . dodaj ( 12 )

**print** ( l i s t o v i ( s ta b l o ))

1. Odrediti broj čvorova koji se nalaze na *i*-tom nivou u stablu.

Kao dodatni argument, dodaje se promenljiva *i* koja određuje trenutni nivo stabla. Za koren je *i* = 0, a za rekurzivne poziva za podstabla se vrednost *i* uvećava za 1. Kada god se naiđe na traženi nivo, izlazi se iz rekurzije.

**from** s ta b l o **import** BinarnoDrvo

**from** s ta b l o **import** Cvor

**def** nivoPomocna ( cvor , t , i ) :

**i f not** cvor :

**return** 0

**i f** i == t :

**return** 1

**return** nivoPomocna ( cvor . l e v i , t + 1 , i ) + nivoPomocna ( cvor . desni , t + 1 , i )

**def** nivo ( s tablo , i ) :

**return** nivoPomocna ( s ta b l o . koren , 0 , i )

s ta b l o = BinarnoDrvo ( ) s ta b l o . dodaj ( 8 )

s ta b l o . dodaj ( 4 ) s ta b l o . dodaj ( 17 ) s ta b l o . dodaj ( 16 ) s ta b l o . dodaj ( 3 ) s ta b l o . dodaj ( 10 ) s ta b l o . dodaj ( 12 )

**for** i **in range** ( 5 ) :

**print** ( i + 1 , nivo ( s tablo , i ) )

1. Za učitani broj sa ulaza, odrediti razliku sume neparnih i parnih cifara broja.

U svakoj iteraciji, cifra je ostatak pri deljenju broja sa 10, a trenutni broj se deli sa 10 (u pitanju je celobrojno deljenje). Zatim se za svaku cifru razmatra ostatak pri deljenju brojem 2.

n = **int** ( **input** ( ) ) suma = 0

**while** n > 0 :

c i f r a = n % 10

**i f** c i f r a % 2 == 1 : suma += c i f r a

**else** :

suma = c i f r a n //= 10

−

**print** ( suma )

1. Napisati funkciju kojom se proverava da li je dati broj prost.

Broj *n* je prost ukoliko je deljiv bilo kojim brojem između 2 i |√*n*|.

**def** p r o s t ( n ) :

**for** i **in range** ( 2 , **int** ( n 0 . 5 ) + 1 ) :

∗∗

**i f** n % i == 0 :

**return** False

**return** True

**for** i **in range** ( 2 , 1 0 ) :

**i f** p r o s t ( i ) :

**print** ( **str** ( i ) + ". j e . p r o s t . b r o j " )

**else** :

**print** ( **str** ( i ) + ". j e . s l o z e n . b r o j " )

1. Broj se učitava sa standardnog ulaza. Ispitati da li su njegove cifre rastuće, opadajuće ili nisu u uređenom poretku.

Sve cifre broja se smeštaju u niz, nakon čega se, iterirajući kroz niz, razmatra rastući i opadajući poredak.

n = **int** ( **input** ( ) ) n i z = [ ]

**while** n > 0 :

n i z = [ n % 1 0 ] + n i z n //= 10

r a s tu c e = True opadajuce = True

**for** i **in range** ( **len** ( n i z ) 1 ) :

−

**i f** n i z [ i + 1 ] >= n i z [ i ] : opadajuce = False

**i f** n i z [ i + 1 ] <= n i z [ i ] : r a s tu c e = False

**i f** r a s tu c e :

**print** ( " C i f r e . su . r a s tu c e " )

**i f** opadajuce :

**print** ( " C i f r e . su . opadajuce " )

**i f not** r a s tu c e **and not** opadajuce :

**print** ( " C i f r e . nisu . uredjene " )

1. Napisati program koji iz ulatnog fajla učitava datum u formatu dan.mesec.godina, a u izlazni fajl ispisuje dan koji mu prethodi u istom formatu. Na primer, za ulaz 1.10.2013, izlaz treba da bude 30.9.2013.

Ukoliko nije prvi datum u mesecu, vrednost dana se smanjuje za 1. Inače, ako jeste, a ukoliko nije januar u pitanju, vrednost meseca se smanjuje za jedan. Inače, ako je prvi januar u pitanju, vrednost godine se smanjuje za jedan. U sva tri slučaja se, po potrebi, ažurira i vrednost dana i meseca. Broj dana u mesecu se smešta u niz, a posebno se ima u vidu da februar ima 29 dana ako je godina prestupna. Godina je prestupna ako je deljiva sa 400 ili ako je deljiva sa 4, a nije sa 100. Tako su npr. godine 1996. i 2000. prestupne, a 2100, 1900. ili 2019. nisu.

meseci = [ 3 1 , 28 , 31 , 30 , 31 , 30 , 31 , 31 , 30 , 31 , 30 , 3 1 ]

**def** prestupna ( godina ) :

**i f** godina % 400 == 0 :

**return** True

**i f** godina % 100 == 0 :

**return** False

**i f** godina % 4 == 0 :

**return** True

**return** False

**input** = **open** ( " input . txt " , " r " )

dan , mesec , godina = [ **int** ( i ) **for** i **in input** . r e a d l i n e ( ) . s p l i t ( ’ . ’ ) ]

**input** . c l o s e ()

**i f** dan > 1 :

dan = 1 mesec > 1 :

**e l i f** −

**i f** prestupna ( godina ) **and** mesec == 3 : dan = 29

**else** mesec −= 1

:

dan = meseci [ mesec − 2 ]

**else** :

mesec −= 1

dan , mesec = 31 , 12 godina −= 1

output = **open** ( " output . txt " , "w" )

output . w r i te ( **str** ( dan ) + " . " + **str** ( mesec ) + " . " + **str** ( godina ) + "\n" ) output . c l o s e ()

1. Sa standardnog ulaza se učitava prirodan broj. Na standardni izlaz ispisati broj koji se dobija izbaci- vanjem cifara stotina i hiljada od početnog broja.

Nakon što se odrede cifre stotina *s* i hiljada *h* broja *n*, od broja *n* se oduzima vrednost 1000*h* + 100*s* i poslednje dve cifre *a* i *b*. Na tu vrednost se na kraju dodaje 10*a* + *b*.

n = **int** ( **input** ( ) )

s t o t i n e = n // 100 % 10 h i l j a d e = n // 1000 % 10 os tatak = n % 100

n = s t o t i n e 100 + h i l j a d e 1000

− ∗ ∗

n //= 100

n += os tatak

**print** ( n )

1. Za prirodne brojeve *x* i *n*, odrediti vrednost sume *x* + 2*x*2 + 3*x*3 + + *nxn*.

Za sve *i* = 1*, , n*, na trenutnu sumu se dodaje vrednost *ixi*.

**def** suma ( x , n ) : s = 0

**for** i **in range** ( 1 , n + 1 ) : s += i x i

∗ ∗∗

**return** s

**print** ( suma ( 2 , 4 ))

1. Iz fajla se učitava tekst, liniju po liniju. Ispisati tekst u izlazni fajl, tako što se linije sa neparnim indeksom prepravljaju tako da su im sva slova mala, a linije sa parnim indeksom tako da su im sva slova velika.

Za linije sa neparnim indeksom se koristi funkcija lower, a za one sa parnim funkcija upper.

**input** = **open** ( " input . txt " , " r " ) output = **open** ( " output . txt " , "w" ) i = 1

**for** l i n e **in input** . r e a d l i n e s ( ) :

**i f** i % 2 :

output . w r i te ( l i n e . lower ( ) )

**else** :

output . w r i te ( l i n e . upper ( ) ) i += 1

**input** . c l o s e () output . c l o s e ()

1. Sa standardnog ulaza se učitava niz u prvom redu, a u drugom broj *k*. Na standardni izlaz treba ispisati niz koji se dobija od polaznog rotacijom za *k* elemenata ulevo.

Ovde se koristi intervalno indeksiranje u Pythonu. Za polazni niz a, rezultujući niz je oblika a[k:]+a[:k].

n i z = [ **int** ( i ) **for** i **in input** ( ) . s p l i t ( ) ] k = **int** ( **input** ( ) )

niz 2 = n i z [ k : ] + n i z [ : k ]

**print** ( niz 2 )

1. Odrediti uniju, presek i razliku dva niza.

Presek su svi oni elementi koji se nalaze u oba niza, a razlika oni koji se nalaze samo u jednom. Svi ovi elementi se jednostavno mogu odrediti pomoću dve petlje. Pretposavka je da se svaki niz sastoji od različitih elemenata. Za uniju je iskorišćena činjenica da je

*A* ∪ *B* = (*A* ∩ *B*) ∪ (*A* \ *B*) ∪ (*B* \ *A*)*.*

**def** presek ( niz 1 , niz 2 ) : n i z = [ ]

**for** i **in** niz 1 :

**for** j **in** niz 2 :

**i f** i == j :

n i z . append ( i )

**break**

**return** n i z

**def** r a z l i k a ( niz 1 , niz 2 ) : n i z = [ ]

**for** i **in** niz 1 :

s a d r z i = False

**for** j **in** niz 2 :

**i f** j == i :

s a d r z i = True

**break i f not** s a d r z i :

n i z . append ( i )

**return** n i z

**def** u n i ja ( niz 1 , niz 2 ) :

**return** presek ( niz 1 , niz 2 ) + r a z l i k a ( niz 1 , niz 2 ) + r a z l i k a ( niz 2 , n i z 1 )

niz 1 = [ 1 , 2 , 3 , 4 , 5 , 6 , 8 , 1 0 ]

niz 2 = [ 1 , 2 , 3 , 4 , 5 , 7 , 9 , 1 1 ]

**print** ( u n i ja ( niz 1 , niz 2 ) ) **print** ( presek ( niz 1 , niz 2 ) ) **print** ( r a z l i k a ( niz 1 , n i z 2 ) )

1. Iz niza izbaciti sve neparne elemente.

Ovde se koristi pogodnost modula numpy, gde sa sa niz[niz 2 == 0] jednostavno mogu izdvojiti parni elementi niza.

**import** numpy as np

n i z = np . array ( [ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ] ) n i z = n i z [ n i z % 2 == 0 ]

**print** ( n i z )

1. Iz niza izbaciti sve elemente sa neparnim indeksom.

Za sve elemente čiji je indeks *i* neparan, izbacivanje se vrši funkcijom pop, pri čemu se iteriranje vrši od kraja niza.

n i z = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ]

i = **len** ( n i z ) 1

−

**while** i >= 0 :

**i f** i % 2 :

n i z . pop ( i )

i = 1

−

**print** ( n i z )

1. Iz niza izbaciti sve proste brojeve.

Za sve elemente koji su prosti, izbacivanje se vrši funkcijom pop, pri čemu se iteriranje vrši od kraja niza.

**def** p r o s t ( n ) :

**for** i **in range** ( 2 , n ) :

**i f** n % i == 0 :

**return** False

**return** True

n i z = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ]

i = **len** ( n i z ) 1

−

**while** i >= 0 :

**i f** p r o s t ( n i z [ i ] ) : n i z . pop ( i )

i = 1

−

**print** ( n i z )

1. Napisati funkciju kojom se razmenjuju vrednosti *i*-te i *j*-te kolone matrice.

Za indeks *k* kojim se iterira kroz sve redove matrice, zamena vrednosti se jednostavno vrši zamenom odgovarajućih elemenata u svakoj iteraciji.

**import** numpy as np

matrica = np . array ( [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] , [ 7 , 8 , 9 ] ] )

**print** ( matrica ) i , j = 0 , 2

**for** k **in range** ( ( **len** ( matrica ) ) ) :

matrica [ k , i ] , matrica [ k , j ] = matrica [ k , j ] , matrica [ k , i ]

**print** ( matrica )

1. Napisati funkciju kojom se proverava da li je data matrica simetrična. Matrica *M* je simetrična ako je *M* = *M T* .

**import** numpy as np

matrica 1 = np . array ( [ [ 1 , 2 ] , [ 3 , 4 ] ] )

matrica 2 = np . array ( [ [ 1 , 3 ] , [ 3 , 1 ] ] )

**print** ( matrica 1 )

**i f** np . array\_equal ( matrica 1 , matrica 1 . T) :

**print** ( " s i m e tr i c n a . j e " )

**else** :

**print** ( " n i j e . s i m e tr i c n a " )

**print** ( matrica 2 )

**i f** np . array\_equal ( matrica 2 , matrica 2 . T) :

**print** ( " s i m e tr i c n a . j e " )

**else** :

**print** ( " n i j e . s i m e tr i c n a " )

1. Odrediti sumu elemenata ispod sporedne dijagonale matrice.

Elementi ispod sporedne dijagonale su svi oni elementi *Mij* za koje je *i* + *j > n* + 1, gde je *n* dimenzija matrice. Kako u Pythonu indeksiranje počinje od nule, odgovarajući uslov je *i* + *j > n* − 1.

matrica = [ [ 1 , 2 , 3 , 4 ] , [ 5 , 6 , 7 , 8 ] , [ 9 , 10 , 11 , 1 2 ] , [ 1 3 , 14 , 15 , 1 6 ] ]

n = **len** ( matrica ) s = 0

**for** i **in range** ( n ) :

**for** j **in range** ( n ) :

**i f** i + j > n 1 :

−

s += matrica [ i ] [ i ]

**print** ( s )

1. Iz datoteke se u prvom redu učitava broj *n*, koji predstavlja broj tačaka, a u narednih *n* redova se učitavaju njihove koordinate. Odrediti dve tačke sa najmanjim i najvećim rastojanjem.

Razmotre se rastojanja svake dve tačke, na osnovu čega se određuju one sa najmanjim, odnosno najvećim rastojanjem.

**def** r a s t o j a n j e ( x1 , y1 , x2 , y2 ) :

**return** ( ( x1 − x2 ) ∗∗ 2 + ( y1 − y2 ) ∗∗ 2 ) ∗∗ 0 . 5

**input** = **open** ( " input . txt " , " r " ) n = **int** ( **input** . r e a d l i n e ( ) )

x = [ 0 ] n

∗

y = [ 0 ] n

∗

**for** i **in range** ( n ) :

x [ i ] , y [ i ] = [ **int** ( i ) **for** i **in input** . r e a d l i n e ( ) . s p l i t ( ) ]

**input** . c l o s e ()

**min** = r a s t o j a n j e ( x [ 0 ] , y [ 0 ] , x [ 1 ] , y [ 1 ] )

**max** = **min**

najvece = [ 0 , 1 ] najmanje = [ 0 , 1 ]

**for** i **in range** ( n 1 ) :

−

**for** j **in range** ( i + 1 , n ) :

d = r a s t o j a n j e ( x [ i ] , y [ i ] , x [ j ] , y [ j ] )

**i f** d < **min** :

**min** = d

najmanje = [ i , j ]

**i f** d > **max**:

**max** = d

najvece = [ i , j ]

**print** ( " Najvece . r a s t o j a n j e : ." , end = "" )

**print** ( najvece , **max**)

**print** ( " Najmanje. r a s t o j a n j e : ." , end = "" )

**print** ( najmanje , **min**)

1. Prepisati iz ulazne u izlaznu datoteku sve one linije koje se završavaju datim karakterom.

Dovoljno je proveriti da li je poslednji element stringa koji predstavlja trenutnu učitanu liniju jednak datom karakteru.

c = "a"

**input** = **open** ( " input . txt " , " r " ) output = **open** ( " output . txt " , "w" ) **for** l i n e **in input** . r e a d l i n e s ( ) :

**i f** l i n e [ **len** ( l i n e ) 2 ] == c : output . w r i te ( l i n e )

−

**input** . c l o s e () output . c l o s e ()

1. Odrediti koja reč se najčešće pojavljuje u tekstu s ulaza.

Učitana linija se funkcijom split razdvoji na reči. Zatim se kreira rečnik, koji za svaku reč pamti njen broj pojavljivanja, odakle se jednostavno može izvući informacija o najvećem broju pojavljivanja.

**input** = **open** ( " input . txt " , " r " ) r e c i = **input** . read ( ) . s p l i t () **input** . c l o s e ()

r e c n i k = {}

**for** r e c **in** r e c i :

r e c n i k [ r e c ] = 0

**for** r e c **in** r e c i :

r e c n i k [ r e c ] += 1

**max** = 0

best = 0

**for** r e c **in** r e c i :

**i f** r e c n i k [ r e c ] > **max**: **max** = r e c n i k [ r e c ] best = r e c

**print** ( best )

1. Dve kraljice su date svojim pozicijama na šahovskoj tabli. Odrediti da li se međusobno napadaju.

Kraljice se međusobno napadaju ako su u istom redu, istoj koloni ili na istoj dijagonali. Drugim rečima, ako su njihove pozicije (*p, q*) i (*r, t*), potrebno je da je *p* = *r*, *q* = *t*, *p* + *r* = *q* + *t* ili *p* − *r* = *q* − *t*.

**def** napadaju ( p , q , r , t ) :

**i f** p == r **or** q == t :

**return** True

**i f** p + r == q + t :

**return** True

**i f** p r == q t :

− −

**return** True

**return** False

k r a l j i c a 1 = [ "b" , 2 ] k r a l j i c a 2 = [ "g" , 7 ]

s l o v a = {"a" : 1 , "b" : 2 , " c " : 3 , "d" : 4 , " e " : 5 , " f " : 6 , "g" : 7 , "h" : 8 }

k r a l j i c a 1 [ 0 ] = s l o v a [ k r a l j i c a 1 [ 0 ] ] k r a l j i c a 2 [ 0 ] = s l o v a [ k r a l j i c a 2 [ 0 ] ] **print** ( k r a l j i c a 1 , k r a l j i c a 2 )

**i f** napadaju ( k r a l j i c a 1 [ 0 ] , k r a l j i c a 1 [ 1 ] , k r a l j i c a 2 [ 0 ] , k r a l j i c a 2 [ 1 ] ) :

**print** ( " napadaju . se " )

**else** :

**print** ( " ne. napadaju . se " )

1. Odrediti da li je niz prirodnih brojeva permutacija prvih *n* prirodnih brojeva.

Niz je permutacija prvih *n* prirodnih brojeva ako se njegovim sortiranjem dobija upravo niz [1*,* 2*, . . . , n*].

**import** numpy as np

n i z = [ 4 , 1 , 3 , 5 , 2 ]

prvihN = [ ]

**for** i **in range** ( 1 , **len** ( n i z ) + 1 ) : prvihN . append ( i )

**i f sorted** ( n i z ) == prvihN :

**print** ( " j e s t e . permutacija " )

**else** :

**print** ( " n i j e . permutacija " )

1. Za prirodne brojeve *n* i *k*, odrediti vrednost binomnog koeficijenta .*n*Σ.

*k*

Ovde je korišćena rekurentna relacija

.*n*Σ = .*n* − 1Σ + .*n* − 1Σ*,*

*k k* − 1 *k*

. Σpri čemu je = 1.

*n*

0

**def** binomial ( n , k ) :

C = [ [ 0 **for** x **in range** ( k + 1 ) ] **for** x **in range** ( n + 1 ) ]

**for** i **in range** ( n + 1 ) :

**for** j **in range** ( k + 1 ) :

**i f** j == 0 :

C[ i ] [ j ] = 1

**else** :

− − −

**return**

C[ i ] [ j ] = C[ i 1 ] [ j 1 ] + C[ i 1 ] [ j ] C[ n ] [ k ]

n = 10

k = 5

**print** ( binomial ( n , k ) )

1. Za učitani tekst sa standardnog ulaza, ispisati na standardni izlaz tekst koji se dobija obrtanjem redosleda reči.

Učitana linija se najpre razdvoji na reči, čime se dobija niz reči. Zatim se kroz taj niz vrše zamene, tako što se dva operatora kreću sa raznih strana niza do trenutka dok se ne poklope.

r e c i = **input** ( ) . s p l i t ( ) n = **len** ( r e c i )

**for** i **in range** ( n // 2 ) :

− − − −

**for**

r e c i [ i ] , r e c i [ n 1 i ] = r e c i [ n 1 i ] , r e c i [ i ] r e c **in** r e c i :

**print** ( rec , end = "." )

**print** ( )

1. Napisati funkciju kojom se proverava da li je dati graf povezan.

Ovde je korišćena klasa Graf implementirana u delu o strukturama podataka. Pretpostavka je da graf neusmeren (ako postoji ivica od čvora *i* do čvora *j*, tada uvek postoji ivica od *j* do *i*). Graf je povezan, ako se pretragom po dužini može stići iz proizvoljnog čvora do svih ostalih čvorova. Ukoliko to nije slučaj, tj. ukoliko se nakon poziva funkcije poseti nađe bar jedan čvor u nizu posecen sa vrednošću False, graf nije povezan.

**from** g r a f **import** Graf

**def** p o s e t i ( graf , tre nutni , posecen ) : posecen [ tr e n u tn i ] = True

**for** i **in** g r a f . g r a f [ tr e n u tn i ] :

**i f not** posecen [ i ] :

p o s e t i ( graf , i , posecen )

matrica 1 = [ [ 1 , 2 , 5 ] , [ 0 , 2 , 4 , 5 ] , [ 0 , 1 , 3 , 5 ] , [ 2 , 4 ] , [ 1 , 3 , 5 ] , [ 0 , 1 , 4 ] ]

matrica 2 = [ [ 1 , 2 , 5 ] , [ 0 , 2 , 4 , 5 ] , [ 0 , 1 , 3 , 5 ] , [ 2 , 4 ] , [ 1 , 3 , 5 ] , [ 0 , 1 , 4 ] ,

[ 7 , 9 ] , [ 6 , 8 , 9 ] , [ 7 , 9 ] , [ 6 , 7 , 8 ] ]

g r a f 1 = Graf ( matrica 1 )

posecen 1 = [ False ] **len** ( g r a f 1 . g r a f ) povezan1 = True

∗

p o s e t i ( graf 1 , 0 , posecen 1 )

**for** p **in** posecen 1 :

**i f not** p :

povezan1 = False

**break i f** povezan1 :

**print** ( " Prvi . g r a f . j e . povezan " )

**else** :

**print** ( " Prvi . g r a f . n i j e . povezan " )

g r a f 2 = Graf ( matrica 2 )

posecen 2 = [ False ] **len** ( g r a f 2 . g r a f ) povezan2 = True

∗

p o s e t i ( graf 2 , 0 , posecen 2 )

**for** p **in** posecen 2 :

**i f not** p :

povezan2 = False

**break i f** povezan2 :

**print** ( " Drugi . g r a f . j e . povezan " )

**else** :

**print** ( " Drugi . g r a f . n i j e . povezan " )

1. Odrediti sa koliko boja se dati graf može obojiti. Pretpostaviti da su svi čvorovi u grafu koji su povezani iste boje.

Ovde je takođe korišćena pretraga po dubini koja kreće od jednog čvora i sve čvorove na koje naiđe boji istom bojom kao polazni čvor. Ukoliko su svi čvorovi posećeni tom pretragom, graf se može obojiti jednom bojom. Inače se iterira kroz sve čvorove, i kad god se naiđe na čvor koji nije obojen, postupak se ponavlja. Nakon što se razmotri svaki čvor, svi čvorovi u grafu će biti obojeni nekom bojom. Broj različitih boja u grafu obojenim na ovaj način se naziva i brojem komponenti povezanosti grafa, a svi čvorovi koji su obojeni istom bojom čine jednu komponentu povezanosti.

**from** g r a f **import** Graf

**def** oboj ( graf , trenutni , boje , boja ) : boje [ tr e n u tn i ] = boja

**for** i **in** g r a f . g r a f [ tr e n u tn i ] :

**i f** boje [ i ] == 0 :

oboj ( graf , i , boje , boja )

matrica = [ [ 1 , 2 , 5 ] , [ 0 , 2 , 4 , 5 ] , [ 0 , 1 , 3 , 5 ] , [ 2 , 4 ] , [ 1 , 3 , 5 ] ,

[ 0 , 1 , 4 ] , [ 7 , 9 ] , [ 6 , 8 , 9 ] , [ 7 , 9 ] , [ 6 , 7 , 8 ] , [ 1 1 ] , [ 1 0 ] ]

g r a f = Graf ( matrica )

boje = [ 0 ] **len** ( g r a f . g r a f ) boja = 0

∗

**for** i **in range** ( **len** ( matrica ) ) :

**i f** boje [ i ] > 0 :

**continue**

boja += 1

oboj ( graf , i , boje , boja )

**print** ( " Graf . se .moze. o b o j i t i . sa .%d. boje " % boja )

**for** i **in range** ( **len** ( matrica ) ) :

**print** ( "−. cvor .%d. j e . obojen .bojom.%d." % ( i , boje [ i ] ) )

1. Data je binarna matrica dimenzija *m n*, gde 0 označava prohodan put, a 1 prepreku. Odrediti minimalan broj koraka potrebnih da se dođe iz početne pozicije (*i*1*, j*1) do krajnje pozicije (*i*2*, j*2),

×

1 *i*1*, i*2 *m*, 1 *j*1*, j*2 *n*, krećući se jedino po poljima označenim sa 0. Mogući smerovi kratanja iz trenutnog polja su nalevo, nadesno, nagore ili nadole. Treba voditi računa da se ne sme izaći iz okvira same matrice.

≤ ≤ ≤ ≤

Na osnovu matrice se kreira graf sa *mn* čvorova, gde postoji ivica između svaka dva susedna polja koja su prohodna (gde je vrednost u matrici 0). Pritom se vodi i računa da se ne izađe iz okvira matrice. Zatim se, u duhu pretrage po širini grafa, određuje rastojanje svakog čvora od polaznog, a traženi broj koraka je rastojanje krajnjeg čvora od polaznog. Primetimo da se u ovu svrhu ne može primeniti pretraga po širini, jer ona ne nalazi najkraći broj koraka za svaki čvor.

**from** g r a f **import** Graf

**from** g r a f **import** Red

**def** duzina ( graf , pocetni , r a s t o j a n j e ) : red = Red ( )

red . dodaj ( p o c e tn i )

r a s t o j a n j e [ p o c e tn i ] = 0

**while not** red . prazan ( ) :

tr e n u tn i = red . o b r i s i ( )

**for** i **in** g r a f . g r a f [ tr e n u tn i ] :

**i f** r a s t o j a n j e [ i ] == 1: red . dodaj ( i )

−

r a s t o j a n j e [ i ] = r a s t o j a n j e [ tr e n u tn i ] + 1

matrica = [ [ 0 , 0 , 0 , 0 , 0 ] ,

[ 0 , 1 , 0 , 0 , 0 ] ,

[ 0 , 0 , 0 , 1 , 0 ] ,

[ 0 , 0 , 0 , 1 , 0 ] ,

[ 0 , 1 , 0 , 0 , 0 ] ]

i 1 , j 1 = 4 , 3

i 2 , j 2 = 1 , 3

m = **len** ( matrica )

n = **len** ( matrica [ 0 ] ) graf Matrica = [ ]

**for** i **in range** (m n ) : graf Matrica . append ( [ ] )

∗

**for** i **in range** (m) :

**for** j **in range** ( n ) :

**i f** ( j > 0 **and** matrica [ i ] [ j − 1 ] != 1 ) :

graf Matrica [ i ∗ n + j ] . append ( i ∗ n + ( j − 1 ))

**i f** ( j < n − 1 **and** matrica [ i ] [ j + 1 ] != 1 ) : graf Matrica [ i ∗ n + j ] . append ( i ∗ n + ( j + 1 ))

**i f** ( i < m − 1 **and** matrica [ i + 1 ] [ j ] != 1 ) : graf Matrica [ i ∗ n + j ] . append ( ( i + 1 ) ∗ n + j )

**i f** ( i > 0 **and** matrica [ i − 1 ] [ j ] != 1 ) :

graf Matrica [ i ∗ n + j ] . append ( ( i − 1 ) ∗ n + j )

g r a f = Graf ( graf Matrica )

r a s t o j a n j e = [ − 1 ] ∗ **len** ( g r a f . g r a f ) duzina ( graf , i 1 ∗ n + j1 , r a s t o j a n j e ) **print** ( r a s t o j a n j e [ i 2 ∗ n + j 2 ] )