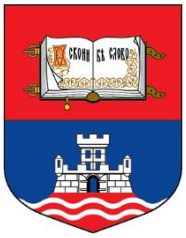


Matematički fakultet
Univerzitet u Beogradu



# Elektronske lekcije o rekurziji u programskom jeziku Pajton

- master rad -

mentor:
doc. dr Miroslav Marić

kandidat:
Saša Petrović

Beograd 2015.

# Sadržaj



## Spisak slika

## Spisak primera



## Spisak algoritama

# Uvod

Programiranje, jedno od najtraženijih zanimanja današnjice sa osnovanom pretpostavkom da će biti sve traženije, predstavlja proces stvaranja i izvršavanja algoritama korišćenjem određenih programskih jezika. Pri odabiru prvog programskog jezika za učenje treba obratiti pažnju na njegovu trenutnu zastupljenost u praksi, kao i na predviđanja za njegovo korišćenje u narednom periodu. Za savladavanje programiranja nije potrebno do detalja naučiti jedan programski jezik već je dovoljno savladati osnovne koncepte programiranja koji se mogu primeniti u raznim programskim jezicima. Jedan od najpogodnijih programskih jezika za razumevanje osnovnih koncepata programiranja je Pajton (engl. *Python*) [1].

Cilj ovog rada je da ukaže na prednosti korišćenja elektronskog kursa za programski jezik Pajton kao modernog i originalnog načina za savladavanje prvih programerskih koraka u školi ili van nje. Na ovaj način učenici imaju mogućnost učenja programiranja na lako pristupačan i zanimljiv način. U ovom radu biće detaljnije obrađena jedna od tema elektronskog kursa, „Rekurzija“.

# 1 Programski jezik Pajton

## 1.1 Opšte karakteristike programskog jezika Pajton

Pajton je interpreterski, platformski nezavisan jezik visokog nivoa sa raznovrsnom primenom (interpreterski jezik je jezik koji prevodi napisani kod u mašinski jezik tek u trenutku izvršenja programa, a ne ranije kao kod kompajlerskih jezika [2]). Pajton se može koristiti za proceduralno, funkcionalno i objektno orijentisano programiranje, a može se koristiti i kao skript jezik. Razvijen je od strane holandskog programera Gvida van Rosuma početkom devedesetih godina dvadesetog veka koji je inspiraciju za ime programskog jezika Pajton dobio od britanske televizijske komedije „Leteći cirkus Montija Pajtona“ [3].

## 1.2 Prednosti i mane programskog jezika Pajton

Prednosti programskog jezika Pajton su mnogobrojne. Pajton je besplatan, jednostavan za korišćenje, moćan i objektno orijentisan programski jezik koji pored toga što može da se integriše u druge programske jezike (C, C++, Java), može da se pokrene i na raznim platformama (Windows, Macintosh, Linux) [4, 5]. Dovoljno je jak programski jezik kako bi ga koristile kompanije kao što su Microsoft, Google, Yahoo, NASA, a dovoljno lak programski jezik koji može poslužiti u pravljenju prvih programerskih koraka.

Jedan od osnovnih zadataka programskih jezika je da premoste jaz između programerskog mozga i računara. Iako je većina programskih jezika bliža jeziku ljudi u odnosu na jezik mašine, čitanje Pajton koda usled svoje jednostavnosti i jasnoće podseća na čitanje engleskog jezika [6]. Pajton programi su kraći i kod profesionalnih programera zahtevaju manje vremena za izradu u poređenju sa drugim programskim jezicima.

Pajton je objektno orijentisan programski jezik i u odnosu na programske jezike kao što su C# i Java, koristi drugačiji pristup. Kod Pajtona korišćenje objektno orijentisanog programiranja je opciono, što znači da pisanje kratkih programa nije kompleksno kao kod C# i Jave, dok se i dalje može koristiti kao objektno orijentisani programski jezik za pisanje velikih projekata sa timom programera.

U slučaju rada na velikim projektima Pajton iskazuje jednu manu, a to je da ne dostiže efikasnost u kompajliranju i dinamičkom prevođenju poput drugih bržih programskih jezika kao što su C++, Java i C#, ali on pronalazi mnoge primene uključujući i učenje programiranja gde brzina kompajliranja nije od prevelikog značaja [7].

Pošto Pajton ne pravi razliku između objekata i primitivnih tipova i kako se ne mora puno razmišljati o tipovima podataka, Pajton predstavlja jedan lak programski jezik koji je koristan za učenje i sticanje veština samog programiranja. Tokom korišćenja ovog programskog jezika programeru je omogućeno da više razmišlja o problemu nego o načinu pisanja koda [8]. Stil pisanja koda u Pajtonu (korišćenje praznina umesto vitičastih zagrada) stvara naviku kod početnika za elegantno unošenje koda.

Rad sa bilo kojim programskim jezikom zahteva određeno znanje o značenju kodova kako bi se izvršavala određena funkcija. Najjednostavniji primeri za program u Pajtonu su zaista

jednostavni, dok u ekvivalentnim programima u drugim programskim jezicima, na primer u Javi, to nije slučaj.

*Algoritam 1.*
***„Hello world“ u Pajtonu***

```
print(„Hello world“)
```

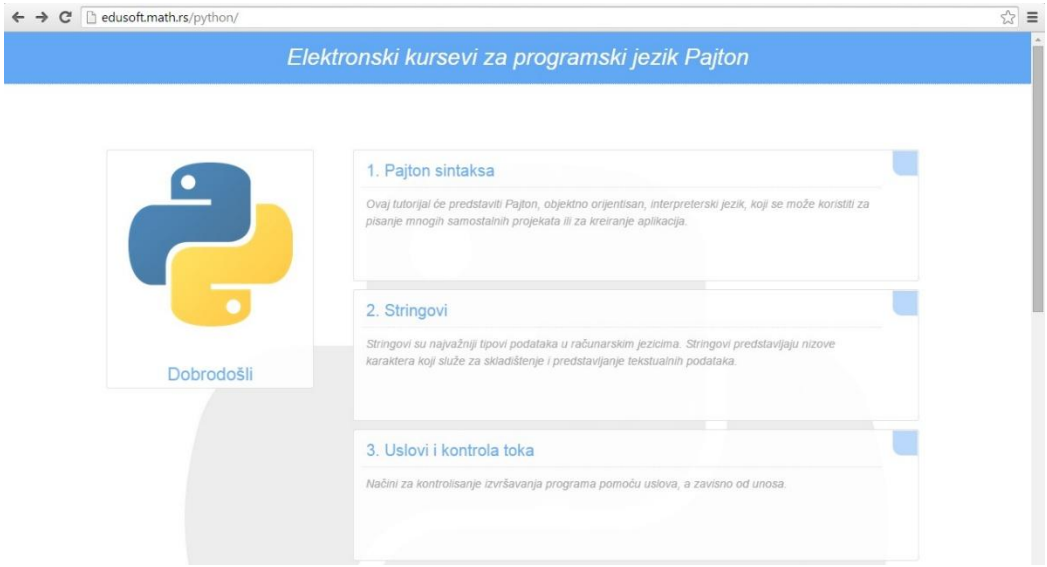
*Algoritam 2.*
***„Hello world“ u Javi***

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println(„Hello world“);
    }
}
```

## 1.3 Elektronski kurs za programski jezik Pajton

Primeri i objašnjenja za učenje programskog jezika Pajton obrađeni u ovom radu predstavljaju jedan deo elektronskog kursa koji je dostupan na sledećoj adresi:

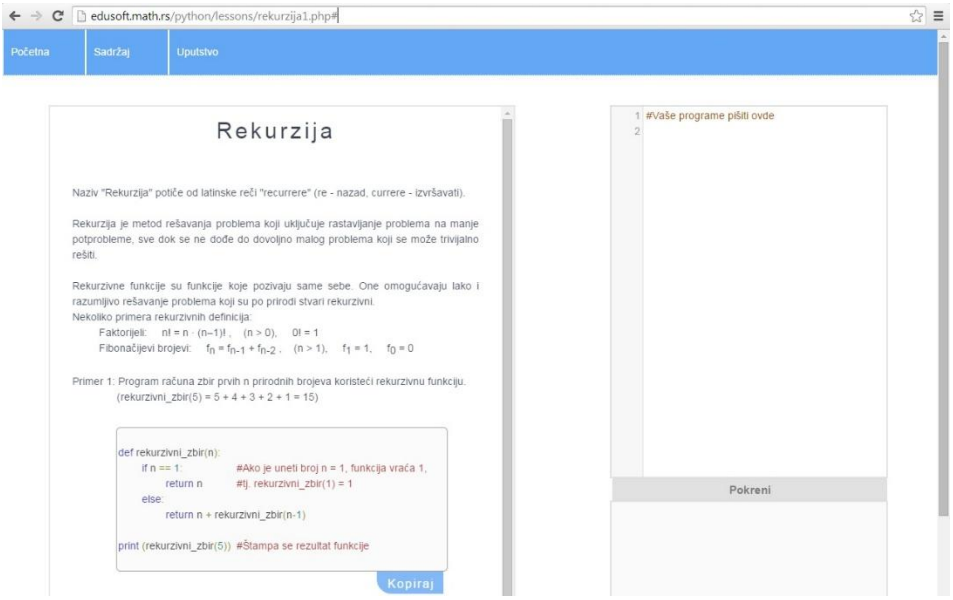
http://edusoft.math.rs/python/




*Slika 1. Naslovna strana elektronskog kursa*

U okviru ovog elektronskog kursa se nalazi mnoštvo tematskih celina (na primer liste, torke i rečnici, klase…) koje su detaljno obrađene kroz slike i primere tako da ovaj kurs predstavlja odličnu pomoć i vodilju za učenike koji se prvi put susreću sa programiranjem. Na slici 1. prikazana je uvodna strana elektronskog kursa.

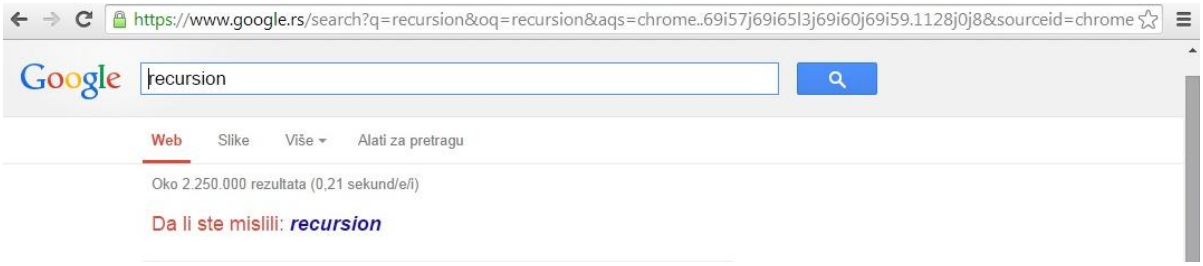
Na početnoj strani sajta nalazi se kompletan sadržaj odnosno tematske celine sa kratkim uvodom u problematiku. Klikom na određenu temu otvara se stranica sa materijalima (slika 2.). U gornjem levom uglu postoji dugme koje vraća na početnu stranu kursa, zatim dugme za sadržaj kompletnog kursa pomoću kojeg se brže dolazi do određene teme, kao i dugme „Uputstvo“ koje ispisuje način da se preuzme i instalira programski jezik Pajton.




*Slika 2. Naslovna strana teme „Rekruzija“*

U okviru svake teme, glavni deo strane je podeljen na tekstualni deo sa leve strane i na konzolu za pisanje programa sa desne strane što omogućava učeniku da kroz samostalno ispisivanje programa prolazi kroz primere i teoriju koji su navedeni. U okviru tematske celine „Rekurzija“ moguće je korišćenjem ove konzole izvršiti sve programe koji su navedeni kroz primere u ovom radu.

# 2 Rekurzija




*Slika 3. Da li ste mislili rekurzija*

Naziv "Rekurzija" potiče od latinske reči "recurrere" (re - nazad, currere – izvršavati). Rekurzija je metoda rešavanja problema koji uključuje rastavljanje problema na manje potprobleme, sve dok se ne dođe do dovoljno malog problema koji se može trivijalno rešiti.

U matematici, rekurzija je usko povezana sa matematičkom indukcijom. Bazni slučaj matematičke indukcije se obično trivijalno dokazuje (za $n = 0$), a u induktivnom koraku se dokazuje da tvrđenje važi za $n + 1$, pod pretpostavkom da je tvrđenje tačno za $n$. Definicija rekurzije ima sličan oblik. Bazni slučaj je slučaj koji se može rešiti bez rekurzivnog poziva, dok se u rekurzivnom koraku, za vrednost $n$, pretpostavlja da je definicija raspoloživa za $n – 1$ [9].

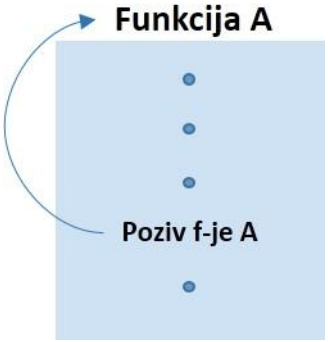
Nekoliko primera rekurzivnih definicija [10]:

Faktorijeli: $n! = n \cdot (n–1)!\,, \quad (n > 0), \quad 0! = 1;$

Fibonačijevi brojevi: $f_n = f_{n-1} + f_{n-2}\,, \quad (n > 1), \quad f_1 = 1, \quad f_0 = 0.$

Rekurzija u programiranju nastaje u slučaju kada funkcija pozove samu sebe bilo direktno ili indirektno. Pojava kada funkcija pozove drugu funkciju iz koje se ponovo poziva početna funkcija naziva se indirektna rekurzija [11].

Prosto rečeno, rekurzivne funkcije su funkcije koje pozivaju same sebe. One omogućavaju lako i razumljivo rešavanje problema koji su po prirodi stvari rekurzivni.




*Slika 4. Definicija rekurzivne funkcije*

Svaka funkcija se predstavlja sa dva entiteta:

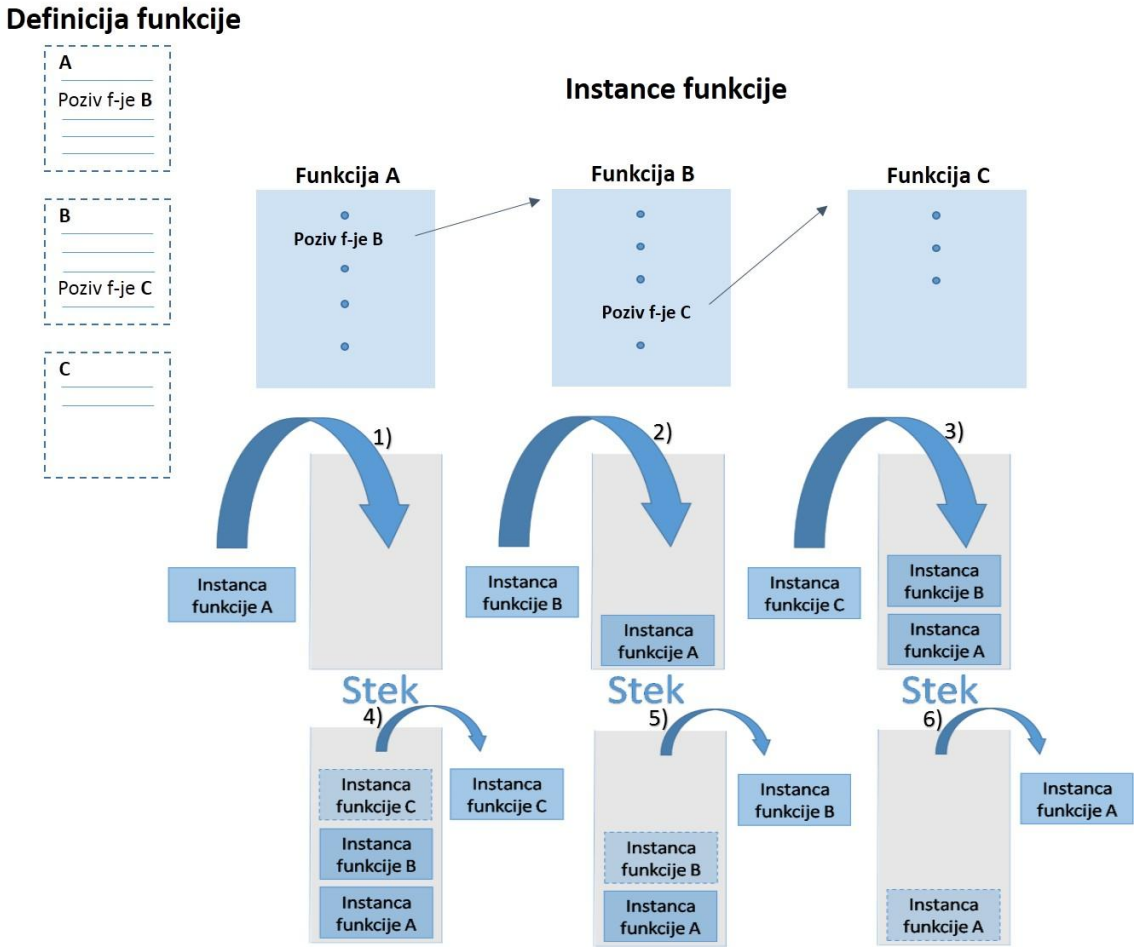
- Definicija funkcije
- Instanca funkcije

Definicija funkcije predstavlja kod funkcije koji diktira njeno izvršavanje, dok je instanca funkcije programsko izvršavanje same funkcije, odnosno pod terminom **instanca** se konkretno

misli na jedan primerak funkcije. Svakim novim pozivom funkcije stvara se nova instanca funkcije koja se skladišti u stek (engl. *stack*) segment memorije, odnosno svakim novim pozivom funkcije stvara se novi stek okvir za novu instancu funkcije [9, 12].

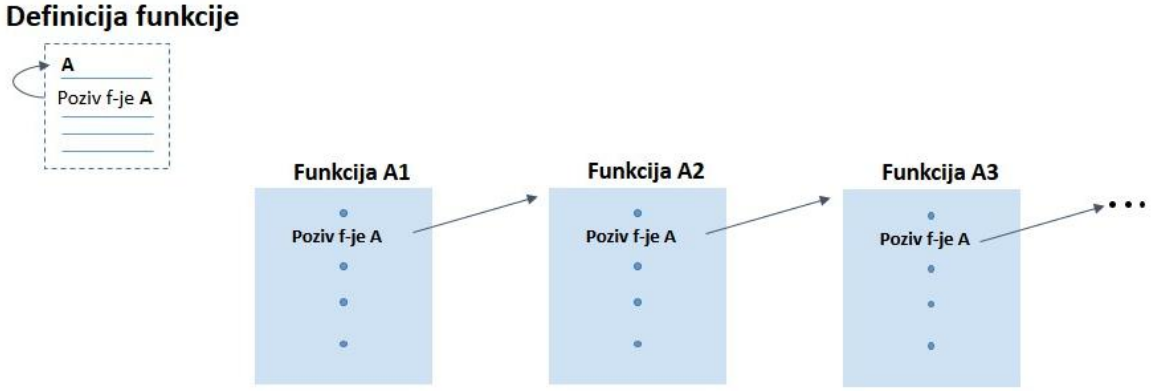
Stek memorija se sastoji od memorijskih lokacija ili registra u operativnoj memoriji koji funkcionišu po *LIFO* (*last in, first out*) principu [13]. Na primer, kao stek se ponaša štap sa kompakt diskovima. Ako se može skidati samo po jedan disk, da bi se došlo do diska koji je na dnu, treba skinuti jedan po jedan disk sa štapa kako bi se došlo do traženog diska.

Kod funkcija koje nisu rekurzivne lako se vizualizuje redosled poziva funkcija. Kao što se može videti sa slike 5., nakon poziva funkcije A, u steku se formira stek okvir za instancu funkcije A koja počinje sa izvršavanjem sve dok ne dođe do poziva funkcije B. Tada se suspenduje izvršavanje instance funkcije A, kreira se novi stek okvir za instancu funkcije B i započinje se izvršavanje instance funkcije B. Kada dođe do poziva funkcije C, suspenduje se izvršavanje instance B, kreira se stek okvir za instancu funkcije C i instanca C započinje svoje izvršavanje. Nakon završetka instance funkcije C, stek okvir instance funkcije C se briše i instanca funkcije B nastavlja svoje izvršavanje odakle je bila suspendovana. Nakon završetka instance funkcije B, stek okvir instance funkcije B se briše, vraća se kontrola instanci funkcije A i instanca funkcije A nastavlja sa izvršavanjem od mesta gde je bila suspendovana. Nakon završetka instance funkcije A, stek okvir instance funkcije A se briše.




*Slika 5. Instanca funkcije i stek*

Kod rekurzivnih funkcija, prilikom svakog poziva rekurzivne funkcije A, stvara se nova instanca te funkcije. Kako su instance identične, poziv funkcije A će se javljati svaki put na istom mestu (slika 6.).




*Slika 6. Rekurzivni poziv*

Jasno je da ako je funkcija A napisana bez uslova koji će prekinuti rekurzivno pozivanje, doći će do beskonačne rekurzije, odnosno funkcija A će se pozivati u beskonačnost. Zato se rekurzivne funkcije uvek moraju pisati sa uslovom koji će prekinuti rekurzivni poziv.

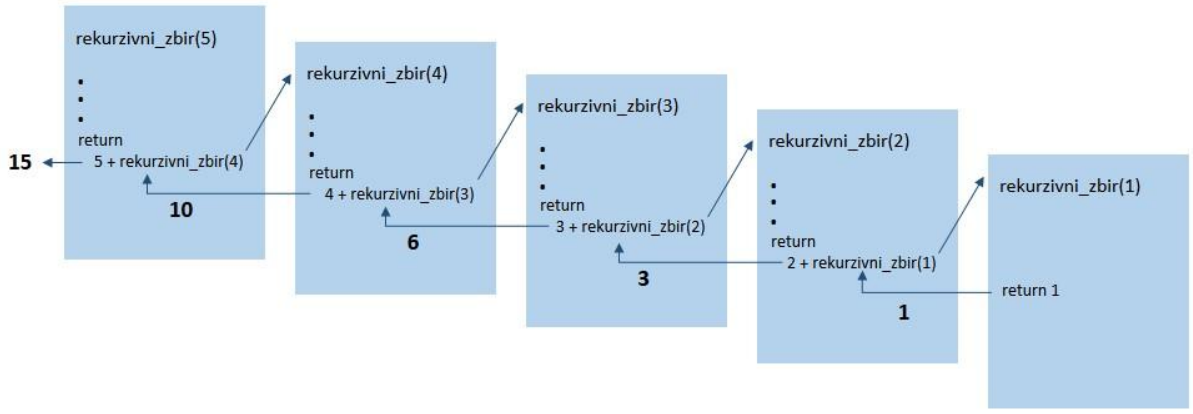
Jednostavan primer rekurzivnog algoritma je program koji računa zbir prvih *n* prirodnih brojeva.

**Primer 1.** Rekurzivna funkcija za prvih *n* brojeva

*Algoritam 3.* ***Rekurzivni zbir***

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

Funkcija rekurzivni_zbir računa zbir prirodnih brojeva i ukoliko je uneti parametar 1, funkcija će napraviti samo jednu instancu koja će po završetku vratiti vrednost 1. Pošto je u definiciji funkcije postavljen uslov da *n* bude veće ili jednako jedinici i pošto se svaki sledeći poziv funkcije izvršava sa manjim parametrom (*n – 1*), zagarantovano je da će se ova rekurzivna funkcija završiti, odnosno neće doći do beskonačne rekurzije. Redosled izvršavanja instanci funkcije *rekurzivni_zbir* za parametar 5 (rekurzivni_zbir(5) = 5 + 4 + 3 + 2 + 1 = 15) je prikazan na slici 7.




*Slika 7. Rekurzivni zbir*

Izvršavanje svake instance funkcije *rekurzivni_zbir* je suspendovano dok se ne izvrši izraz *n + rekurzivni_zbir(n – 1)*. Kada se funkcija *rekurzivni_zbir* pozove sa parametrom 1, svih pet instanci funkcije i dalje postoje, s tim da je izvršavanje prve četiri instance suspendovano dok se ne izvrši instanca *rekurzivni_zbir(1)*. Kada *rekurzivni_zbir(1)* vrati vrednost 1, evaluacija izraza *2 + rekurzivni_zbir(1)* može da se završi, vraćajući 3 kao vrednost instance funkcije *rekurzivni_zbir(2)*. Taj proces se nastavlja na isti način sve dok se ne izvrši korak 5 + *rekurzivni_zbir(4)* i instanca funkcije *rekurzivni_zbir(5)* ne vrati vrednost.

Dakle, prilikom pozivanja funkcije **rekurzivni_zbir(5)**, Pajton prevodilac će uraditi sledeće:

```
rekurzivni_zbir(5)
= 5 + rekurzivni_zbir(4)
= 5 + (4 + rekurzivni_zbir(3))
= 5 + (4 + (3 + rekurzivni_zbir(2)))
= 5 + (4 + (3 + (2 + rekurzivni_zbir(1))))
= 5 + (4 + (3 + (2 + 1)))
= 15
```

## 2.1 Tri pravila rekurzije

Kako bi se napisala dobra i funkcionalna rekurzivna funkcija, potrebno je pratiti pravila pisanja rekurzivnog algoritma:

1. Rekurzivni algoritam mora da ima **bazni slučaj** (engl. *base case*);
2. Rekurzivni algoritam mora da menja svoje stanje i da napreduje ka osnovnom slučaju – rekurzivni korak ili **rekurzivni slučaj** (engl. *recursive case*);
3. Rekurzivni algoritam mora da zove samog sebe.

Bazni slučaj je slučaj u kojem problem može da se reši bez dalje rekurzije, odnosno to je slučaj koji omogućava algoritmu da prestane sa rekurzijom (izlazni kriterijum). Obično je to dovoljno mali problem koji se može rešiti direktno (trivijalno) [14].

U prethodnom primeru (primer 1.), bazni slučaj je bio:

*Algoritam 4.* ***Bazni slučaj***

```
if n == 1:
    return n
```

a rekurzivni korak:

*Algoritam 5.* ***Rekurzivni korak***

```
else:
    return n + rekurzivni_zbir(n-1)
```

Ukoliko bi se izostavio bazni slučaj, rekurzivna funkcija bi pozivala samu sebe beskonačan broj puta što bi dovelo do preopterećenja i pucanja programa (primer 2.).

**Primer 2.** Loš primer rekurzivne funkcije (beskonačna rekurzija):

*Algoritam 6.* ***Beskonačna rekurzija***

```
def brojac(broj):
    print (broj)
    #broj se uvećava za 1
    broj += 1
    #rekurzivni poziv fukcije brojac
    brojac(broj)
```

Pošto nema baznog slučaja, odnosno izlaznog kriterijuma, ovaj program bi se na nekim kompajlerima izvršavao u beskonačnost.

# 3 Izračunavanje faktorijela

Faktorijel nenegativnog celog broja je definisan kao proizvod svih brojeva manjih ili jednakih broju n, npr. $4! = 4 \cdot 3 \cdot 2 \cdot 1$, i pri tom važi $\mathbf{0! = 1}$. Dakle, definicija faktorijela ima oblik $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$, odnosno:

$$n! = n \cdot (n-1)!, \qquad n \geq 1;$$
$$0! = 1.$$

Znak *n!* se čita kao faktorijel, a sam naziv potiče od reči *factor* = množilac.

Termin *faktorijel (factorielle)* prvi uvodi francuski matematičar Luj Arbogast (Louis François Arbogast, 1759 – 1803) sa gore navedenim značenjem i to u svojoj knjizi *Calcul des Dérivations* iz 1800. godine. Nešto kasnije, 1808. godine, nemački matematičar Kramp (Christian Kramp, 1760 – 1826) uvodi oznaku za faktorijel „!“ u obliku *n*!.

Međutim, Krampovo obeležavanje *n*! nije prihvaćeno u 19. veku, pa je Gaus (Carl Friedrich Gauss, 1777 – 1851) za faktorijel koristio oznaku $\Pi(n)$, npr. $\Pi(n) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, a Jakobi (Carl Gustav Jacobi, 1804 – 1851) oznaku $\Pi_n$.

Savet Londonskog matematičkog društva je za vreme Prvog svetskog rata, preciznije 1916. godine, predložio da se faktorijel obeležava isto onako kako je 1808. godine predložio Kramp (*n*!). Društvo je izdalo 1916. godine dokument *Sugestion for Notation and Printing* kojim se vratila oznaka *n*! [15].

Sama definicija faktorijela se može posmatrati kao rekurzivna definicija čiji je bazni slučaj „faktorijel(n) = 1, ako je n = 0“, dok je rekurzivni korak „faktorijel(n) = n * faktorijel(n – 1)“, čime je obezbeđeno uprošćavanje problema koji teži baznom slučaju (primer 3.).

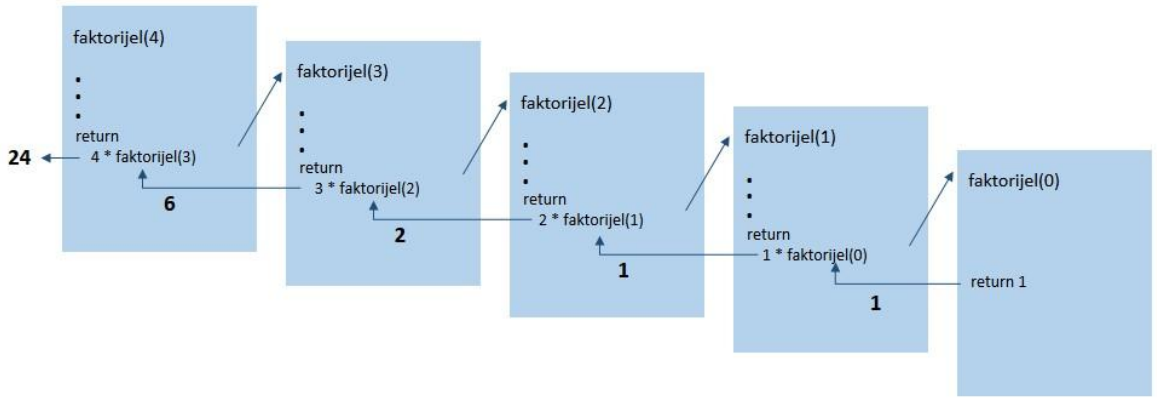
**Primer 3.** Program za izračunavanje faktorijela koji koristi rekurzivni poziv, pod pretpostavkom da je n pozitivan ceo broj.

*Algoritam 7.* ***Rekurzivna faktorijel funkcija***

```
def faktorijel (n):
    #bazni slučaj
    if n == 0:
        return 1
    #rekurzivni korak
    else:
        return n * faktorijel(n-1)
```

Funkcija *faktorijel* računa faktorijel unetog nenegativnog celog broja. Ukoliko je uneti parametar 0, funkcija će napraviti samo jednu instancu koja će na kraju vratiti vrednost 1. Pošto je u definiciji funkcije postavljen uslov da *n* bude veći ili jednak nuli i pošto se svaki sledeći poziv funkcije izvršava sa umanjenim parametrom (n – 1), zagarantovano je da će se ova

rekurzivna funkcija završiti, odnosno, neće doći do beskonačne rekurzije. Redosled izvršavanja instanci funkcije *faktorijel* za parametar 4 je prikazan na slici 8.



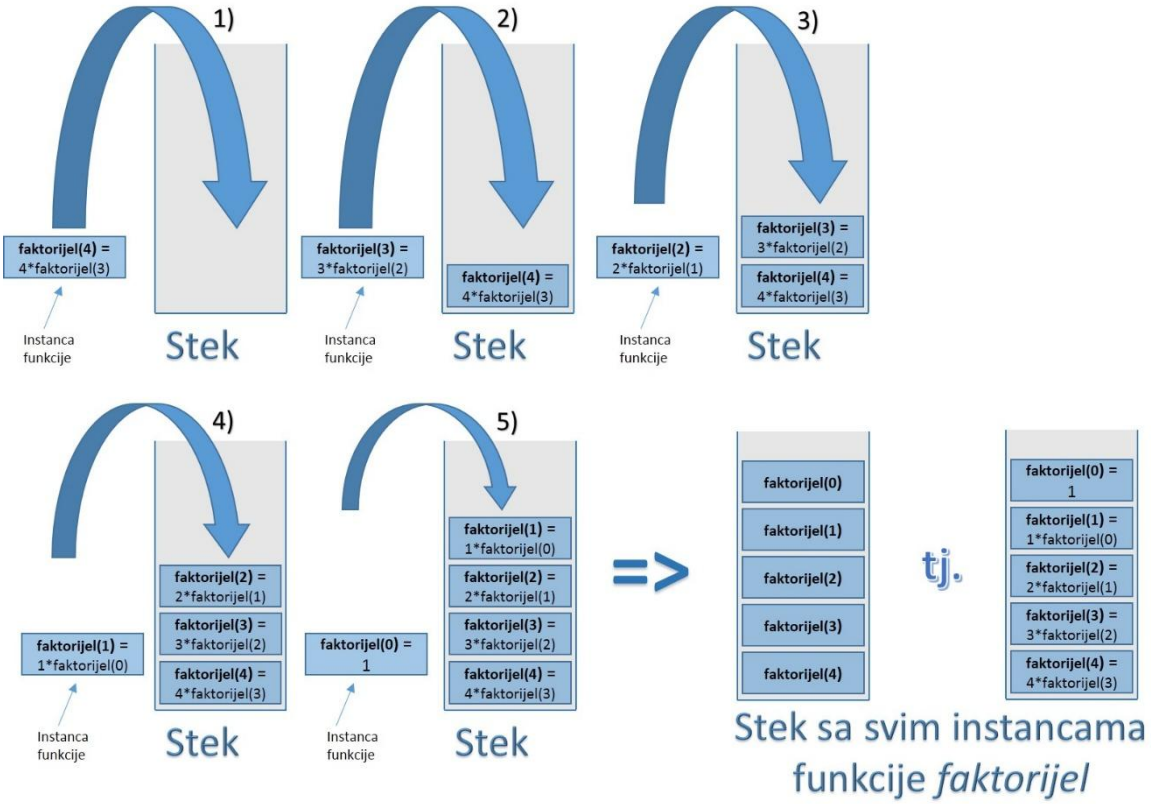

*Slika 8. Faktorijel za parametar 4*

Izvršavanje svake instance funkcije *faktorijel* je suspendovano dok se ne izvrši izraz *n * faktorijel(n – 1)*. Kada se funkcija *faktorijel* pozove sa parametrom 0, svih pet instanci funkcije *faktorijel* i dalje postoje, s tim da je izvršavanje prve četiri instance suspendovano dok se ne izvrši instanca *faktorijel(0)*. Kada *faktorijel(0)* vrati vrednost 1, evaluacija izraza *1 * faktorijel(0)* može da se završi, vraćajući 1 kao vrednost instance funkcije *faktorije(1)*. Taj proces se nastavlja na isti način sve dok se ne izvrši *4 * faktorijel(3)* i dok instanca funkcije *faktorijel(4)* ne vrati vrednost 24.

Ukoliko se pozove funkcija *faktorijel(4)*, Pajton prevodilac će uraditi sledeće:

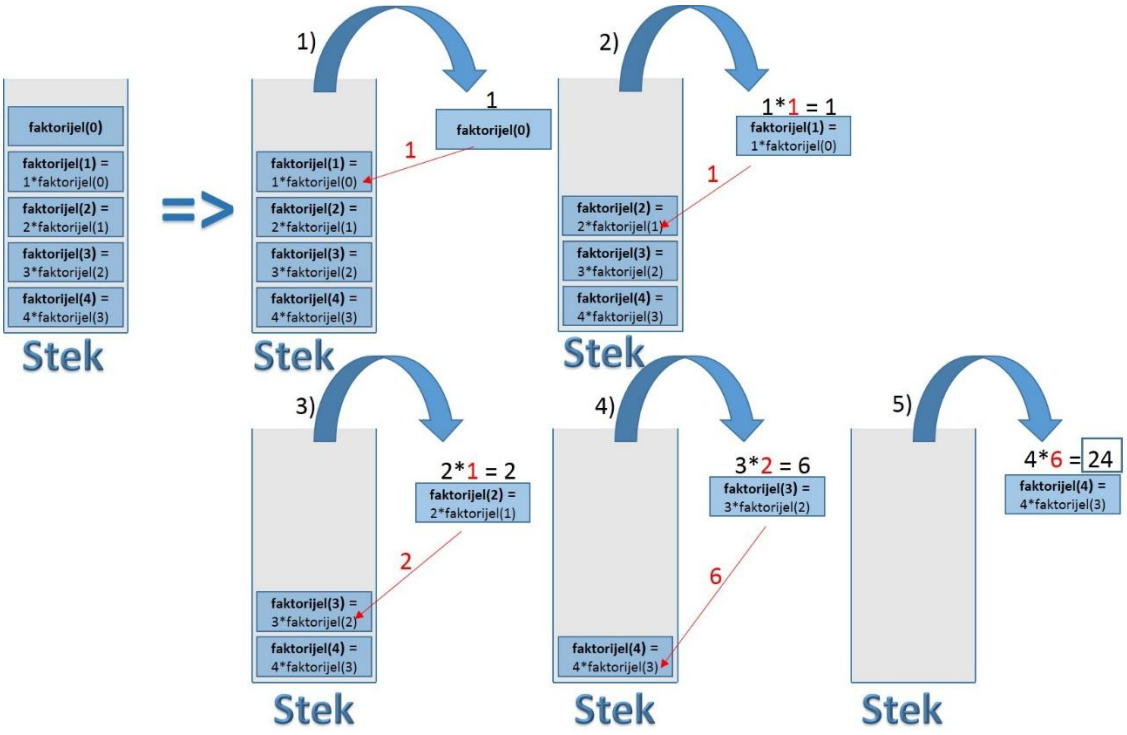
```
fakt (4)
Prvi poziv = 4 * fakt(3)
Drugi poziv = 4 * 3 * fakt(2)
Treći poziv = 4 * 3 * 2 * fakt(1)
Četvrti poziv = 4 * 3 * 2 * 1 * fakt(0)
Peti poziv = 4 * 3 * 2 * 1 * 1 [Prva povratna vrednost iz petog poziva (1 * 1)]
= 4 * 3 * 2 * 1 [Druga povratna vrednost iz četvrtog poziva (2 * 1)]
= 4 * 3 * 2 [Treća povratna vrednost iz trećeg poziva (3 * 2)]
= 4 * 6 [Četvrta povratna vrednost iz drugog poziva (4 * 6)]
= 24 [Peta (poslednja) povratna vrednost iz prvog poziva]
```

Prilikom poziva funkcije *faktorijel(4)*, stek memorija se popunjava stek okvirima koje sadrže nove instance funkcije *faktorijel* koje sadrže isti primerak koda funkcije *faktorijel*. Informacija o tome dokle je instanca funkcije stigla se pamti u stek okviru svake instance i na taj način je omogućen nastavak izvršavanja koda od momenta kada instanca funkcija ponovo postane aktivna (slika 9.).




*Slika 9. Popunjavanje steka - faktorijel*

Na slici ispod se može videti na koji način se oslobađa memorija u steku nakon što su izvršene sve instance funkcije *faktorijel*.




*Slika 10. Oslobađanje steka - faktorijel*

Da bi se videlo šta se dešava u programu, koristi se **print** funkcija.

*Algoritam 8.* ***Rekurzivna faktorijel funkcija sa print funkcijom***

```
#Pretpostavimo da je n nenegativan ceo broj.
def fakt (n):
    print ("Faktorijel je pozvan za broj n = " + str(n))
    if n == 0:
        print ("n = 0 : Faktorijel od 0 je 1")
        return 1
    else:
        rezultat = n * fakt (n-1)
        print ("n = %d : Rezultat za %d * faktorijel(%d) je %d"%(n, n,
               n-1, rezultat))
        return rezultat
```

Program će ispisati sledeće:

```
Faktorijel je pozvan za broj n = 4
Faktorijel je pozvan za broj n = 3
Faktorijel je pozvan za broj n = 2
Faktorijel je pozvan za broj n = 1
Faktorijel je pozvan za broj n = 0
n = 0 : Faktorijel od 0 je 1
n = 1 : Rezultat za 1 * faktorijel(0) je 1
n = 2 : Rezultat za 2 * faktorijel(1) je 2
n = 3 : Rezultat za 3 * faktorijel(2) je 6
n = 4 : Rezultat za 4 * faktorijel(3) je 24
```

# 4 Korišćenje rekurzije nad stringovima

String je niz karaktera koji se koristi za predstavljanje i skladištenje tekstualnih podataka. U programskom jeziku Pajton koriste se jednostruki, dvostruki i trostruki navodnici za deklarisanje stringa. U suštini, ne postoji razlika između jednostrukih i dvostrukih navodnika, osim ako se ne naiđe na string koji u sebi sadrži karakter (’). Na primer, string *Je l’ je to tačno?* se ne može pisati između jednostrukih navodnika zato što Pajton ne prepoznaje kada string počinje i kada se završava. U tom slučaju se koristiti izlazna sekvenca (\). Korišćenjem trostrukih navodnika, string se može deklarisati u više redova.

Stringovi spadaju u nepromenljive tipove podataka, što znači da se njihova vrednost ne može promeniti.

*Algoritam 9.* ***Korišćenje navodnika***

```
print ("Je l' to tacno")       #Dobra upotreba navodnika
print ('Je l' to tacno')       #Loša upotreba navodnika, javlja se greška
print ('Je l\' to tacno')      #Dobra upotreba navodnika
print ('''Moje ime je Bond,
Dzejms Bond.''')               #Dobra upotreba navodnika
```

Napomena: Palindromi su smislene rečenice i reči koje se isto čitaju i sa leve i sa desne strane.

**Primer 4.** Program koji ispituje da li je neka reč palindrom.

*Algoritam 10.* ***Da li je reč palindrom***

```
def da_li_je_palindrom(rec):
    #sva slova se prebacuju u mala radi preciznijeg upoređivanja
    rec = rec.lower()

    #ako se reč sastoji od jednog slova, ona je palindrom
    #bazni slučaj
    if len(rec) == 1 or len(rec) == 0:
        return True

    #rekurzivni slučaj
    else:
    #proverava se prvo i poslednje slovo
        if rec[0] == rec[-1]:
    #koristi se isecanje reči
            return da_li_je_palindrom(rec[1:-1])
        else:
            return False
```

Funkcija *da_li_je_palindrom* prvo prebacuje sva slova u mala kako bi se izbeglo upoređivanje malih i velikih slova. Pošto su reči koje se sastoje od jednog slova palindromi, za

bazni slučaj se uzima uslov da je string dužine 1 ili 0 (u slučaju da se unese prazan string), prilikom čega funkcija *da_li_je_palindrom* vraća *True*. Za određivanje dužine stringa koristi se ugrađena funkcija len().

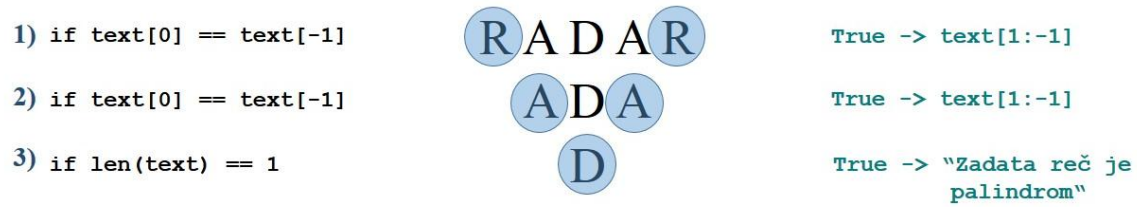
Ako je uneti string dužine veće od jedan, upoređuje se prvi i poslednji karakter. U slučaju da su karakteri različiti, funkcija će vratiti *False* i završiti svoje izvršavanje, a ukoliko su karakteri isti, dolazi do rekurzivnog poziva i pritom se kreira nova instanca funkcije *da_li_je_palindrom* čiji je parametar reč bez prvog i poslednjeg karaktera. Kako se svakim rekurzivnim pozivom prvi i poslednji karakter odstranjuju (korišćenjem isecanja *rec[1:-1]*), zagarantovano je da će se svakim rekurzivnim pozivom napredovati ka baznom slučaju i da neće doći do beskonačne rekurzije. Treba napomenuti da pozicije karaktera u stringu počinju da se broje od 0. U slučaju da string sadrži *n* karaktera, prvi karakter će zauzimati poziciju 0 a poslednji poziciju *n – 1*. Dakle, korišćenjem isecanja *rec[1:-1]*, prvi karakter sa pozicije 0 i poslednji karakter sa pozicije -1 se odstranjuju.

Funkcija *da_li_je_palindrom* može se testirati na sledeći način:

*Algoritam 11.* ***Testiranje funkcije da_li_je_palindrom***

```
print (da_li_je_palindrom("zdravo"))          #program ispisuje „False“
print (da_li_je_palindrom("radar"))           #program ispisuje „True“
print (da_li_je_palindrom("Teret"))           #program ispisuje „True“
print (da_li_je_palindrom("lampa"))           #program ispisuje „False“
```

Na slici ispod je prikazano izvršavanje funkcije *da_li_je_palindrom(“radar”)*.




*Slika 11. Upoređivanje karaktera*

**Primer 5.** Program za ispisivanje reči unazad.

*Algoritam 12.* ***Funkcija za ispisivanje reči unazad***

```
def okreni(rec):
    #bazni slučaj
    #string dužine 1 je palindrom
    if len(rec) <=  1:
        return rec

    #rekurzivni korak
    else:
        return okreni(rec[1:])  +  rec[0]
```

*Algoritam 13.* ***Poziv funkcije okreni(“zdravo”)***

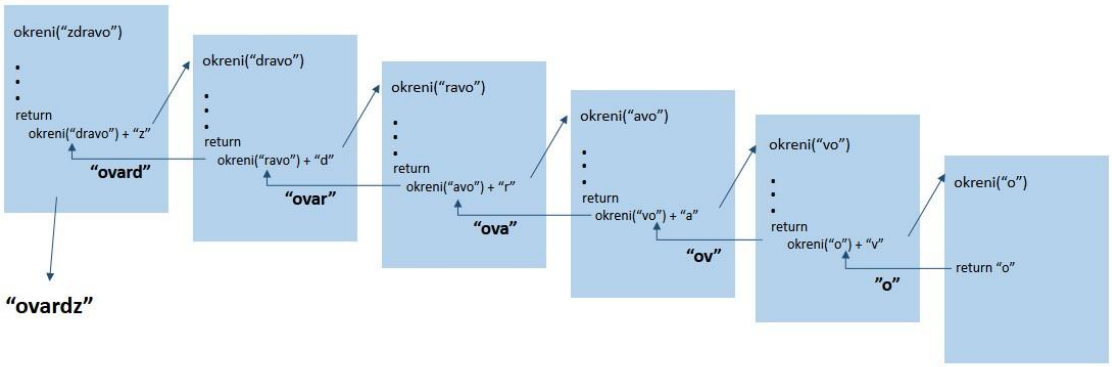
```
print (okreni("zdravo"))
```

Program radi sledeće:

```
okreni("zdravo")
= okreni("dravo") + z
= okreni("ravo") + d + z
= okreni("avo") + r + d + z
= okreni("vo") + a + r + d + z
= okreni("o") + v + a + r + d + z        #osnovi uslov kada je reč dužine 1
= o + v + a + r + d + z
= ovardz
```

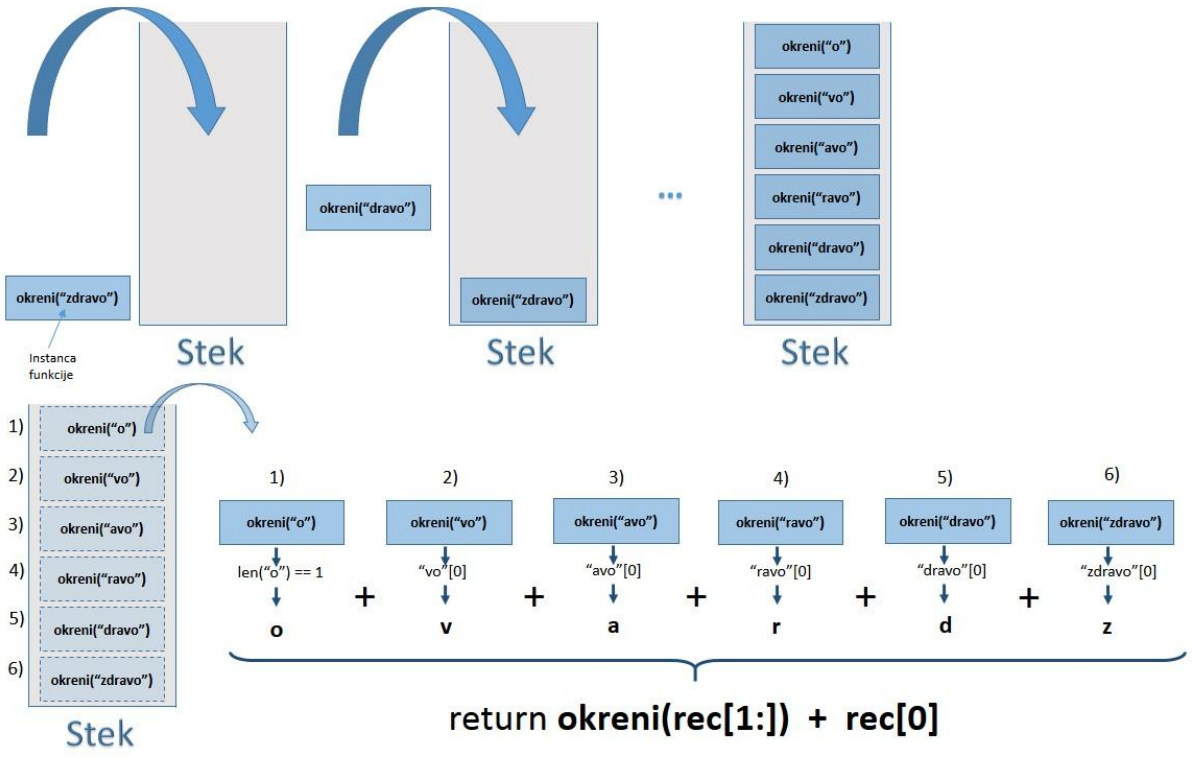
Prilikom pozivanja funkcije *okreni* kreira se instanca te funkcije koja u svom izvršavanju proverava da li je uneti string dužine manje ili jednako jedan. Ako je dužina stringa manja ili jednaka jedan, neće doći do kreiranja nove instance i funkcija će vratiti uneti string.

Ukoliko je dužina stringa veća od jedan, dolazi do rekurzivnog poziva i kreiranja nove instance funkcije *okreni* sa parametrom, odnosno stringom iz kojeg je izbačen prvi karakter. Izbacivanje prvog karaktera se postiže isecanjem stringa koristeći naredbu rec[1:] kojom se iz stringa *rec* uzimaju karakteri koji se nalaze na pozicijama od 1 pa do kraja (napomena: pozicija prvog karaktera je 0). Rekurzivni pozivi i kreiranja instanci se nastavljaju sve dok se početni uneti string ne skrati na dužinu od jednog karaktera, odnosno dok se ne dođe do baznog slučaja. U poslednjoj instanci funkcije se vraća preostali karakter koji se sabira sa početnim karakterom stringa koji je bio parametar u prethodnoj instanci. Dakle, u prethodnom primeru je poslednji karakter „o“ i on se sabira sa prvim karakterom stringa „vo“ = „v“ i tako dalje. Postupak i redosled izvršavanja instanci se može videti na slici ispod.




*Slika 12. Izvršavanje funkcije okreni("zdravo")*

Prilikom poziva funkcije *okreni(„zdravo“)*, stek memorija će se popunjavati kao što je ilustrovano na slici 13.

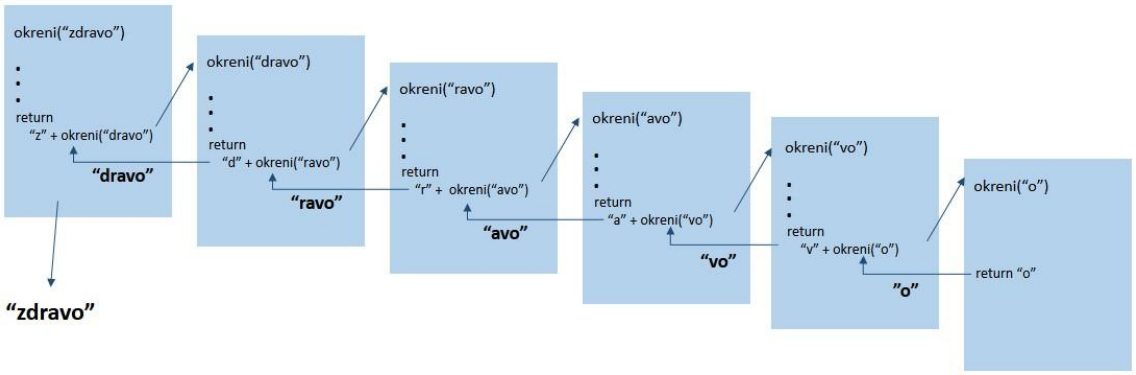



*Slika 13. Popunjavanje steka - string*

Napomena: Bitan je redosled poziva funkcije. Ako se u rekurzivnom koraku zameni redosled "okreni(rec[1:]) + rec[0]" sa "rec[0] + okreni(rec[1:])", program će se ponašati slično kao u primeru faktorijela:

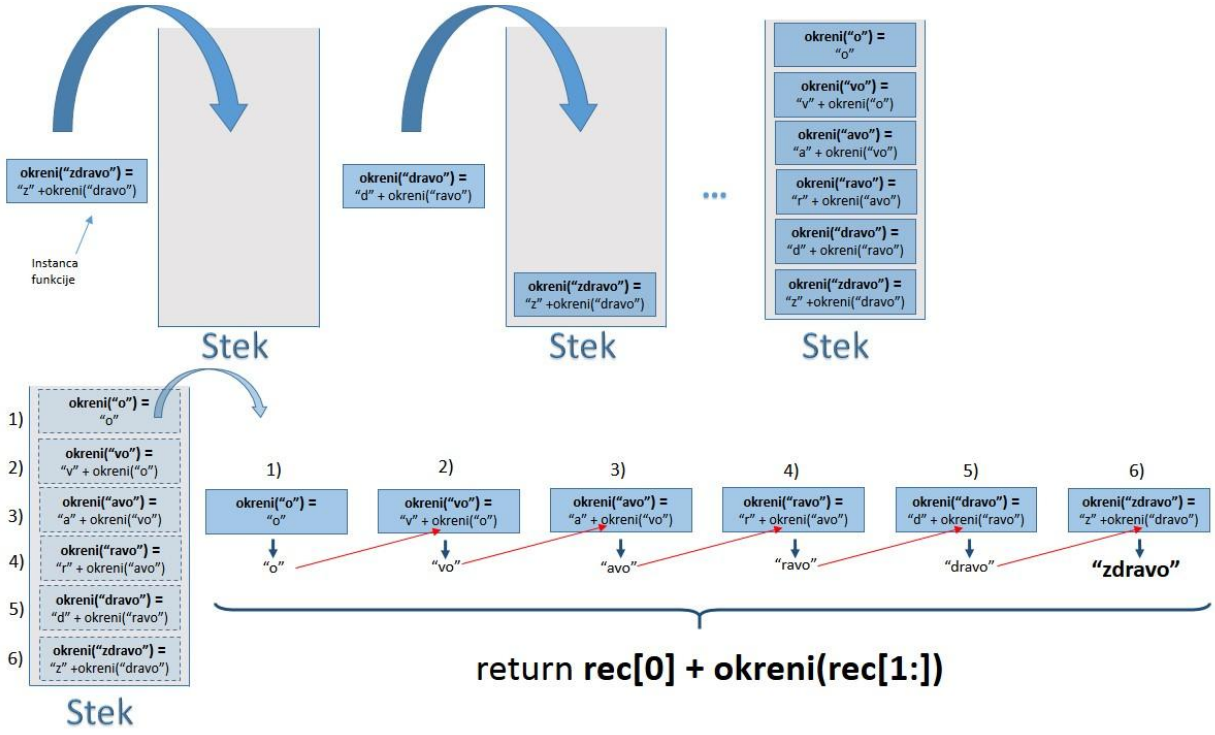
```
okreni("zdravo")
= z + okreni("dravo")
= z + d + okreni("ravo")
= z + d + r + okreni("avo")
= z + d + r +  a + okreni("vo")
= z + d + r +  a + v + okreni("o")      #došli smo do osnovnog uslova kada je reč dužine 1
= z + d + r +  a + v + o
= zdravo
```

Postupak i redosled izvršavanja instanci se može videti na slici 14.




*Slika 14. Izvršavanje funkcije okreni("zdravo") 2*

Prilikom poziva funkcije *okreni(„zdravo“)* sa obrnutim redosledom poziva, stek memorija će se popunjavati na način ilustrovan ispod (slika 15.):




*Slika 15. Obrnuti poziv*

# 5 Dobre i loše strane rekurzije

Rekurzija i iteracija izvršavaju iste vrste zadataka, što bi značilo da se rešava deo po deo komplikovanog zadatka. Suština iteracije je u ponavljanju zadatog koda dok se ne dođe do krajnjeg rešenja zadatka, dok je suština rekurzije u raščlanjivanju velikog problema na male probleme koji se mogu jednostavno rešiti [16].

Zajedničke osobine rekurzije i iteracije:

- Bazirane su na kontrolnoj strukturi (*control structure*);
  - Iteracija koristi ponavljačku strukturu (*repetition structure*);
  - Rekurzija koristi selekcionu strukturu (*selection structure*);
- Koriste ponavljanje u svom procesu;
  - Iteracija eksplicitno koristi ponavljačku strukturu (*repetition structure*);
  - Rekurzija postiže ponavljanje kroz ponovljene pozive metoda;
- Koriste terminacioni test;
  - Iteracija se prekida kada se ne zadovolje uslovi u petlji;
  - Rekurzija se prekida kada se dođe do baznog slučaja;
- Može se javiti beskonačno ponavljanje;
  - Beskonačna petlja se javlja u iteraciji ukoliko je uslov u petlji uvek tačan;
  - Beskonačna rekurzija se javlja kada rekurzivni korak ne uproštava problem ka baznom slučaju [17].

**Dobre strane rekurzije** podrazumevaju čitljiv i kratak kod koji je jednostavan za razumevanje. Dakle, može se naići na problem koji je teško rešiti iterativno i zahteva veliki programerski napor, dok je rekurzivno rešenje tog problema mnogo jednostavnije i ne zahteva veliki programerski napor. Primer ovakvog slučaja su „Kule Hanoja“ koje će kasnije biti obrađene.

**Loše strane rekurzije** se ogledaju u ceni poziva i suvišnog izračunavanja. Cena poziva podrazumeva da se prilikom svakog rekurzivnog poziva popunjava stek memorija što je vremenski i memorijski veoma zahtevno, ukoliko rekurzivnih poziva ima mnogo. Suvišna izračunavanja se javljaju ukoliko se problem razloži na manje potprobleme koji se preklapaju, jer to dovodi do višestrukih rekurzivnih poziva za iste potprobleme. Primer za nepotrebno korišćenje rekurzije predstavlja izračunavanje elemenata Fibonačijevog niza. Ukoliko se pažljivo ispiše rekurzivna funkcija za određeni problem, postoji velika mogućnost da program funkcioniše ispravno bez prevelikog razumevanja o radu same funkcije od strane programera. Međutim, ako se napravi i najmanja greška, njeno otkrivanje je uglavnom mnogo teže nego kod iterativnih funkcija.

Opšte pravilo je da, ako postoji mogućnost da se problem lako reši iterativno, onda treba da se radi iterativno. Ukoliko iterativno rešenje traži preveliki programerski napor, onda problem treba rešiti rekurzivno. Teorijski, svaki problem može da se rešava iterativno.

# 6 Fibonačijev niz

Fibonačijev niz je niz brojeva u kome su prva dva broja 0 i 1, a svaki sledeći je zbir prethodna dva broja, tako da Fibonačijev niz izgleda: 0, 1, 1, 2, 3, 5, 8, 13, 21, … Definicija Fibonačijevog niza ima oblik:

$$F(n) = \begin{cases} 0 & \text{ako je n} = 0; \\ 1 & \text{ako je n} = 1; \\ F(n-1) + F(n-2) & \text{ako je n} > 1. \end{cases}$$

Fibonačijev niz je dobio naziv po italijanskom matematičaru Leonardu Pizanu (Leonardo Pisano, Fibonacci, 1180 – 1240) koji je rođen u Pizi. Nakon majčine smrti nazvan je Fibonači, po nadimku svog oca Bonača (Bonaccio). Osnovno obrazovanje stiče na severu Afrike a putujući kroz mnoge zemlje Evrope i Bliskog istoka stiče dodatno matematičko znanje. Nakon svojih putovanja, publikovao je dve knjige: *Knjiga o abakusu – Liber abaci* (1202) i *Praktična geometrija* (1220).

Fibonači je predstavio niz preko problema o idealizovanom rastu populacije zečeva; pitanje je bilo koliko će pari zečeva reprodukovati jedan par za godinu dana pod uslovom da svakog meseca jedan par rodi par zečeva (jednog muškog i jednog ženskog zeca) koji će za dva meseca postati reproduktivan [18, 19].


*Slika 16. Fibonačijevi brojevi*

Brojevi koji čine Fibonačijev niz se nazivaju Fibonačijevi brojevi (slika 16.). Primer 6. predstavlja program za izračunavanje Fibonačijevog broja korišćenjem rekurzije.

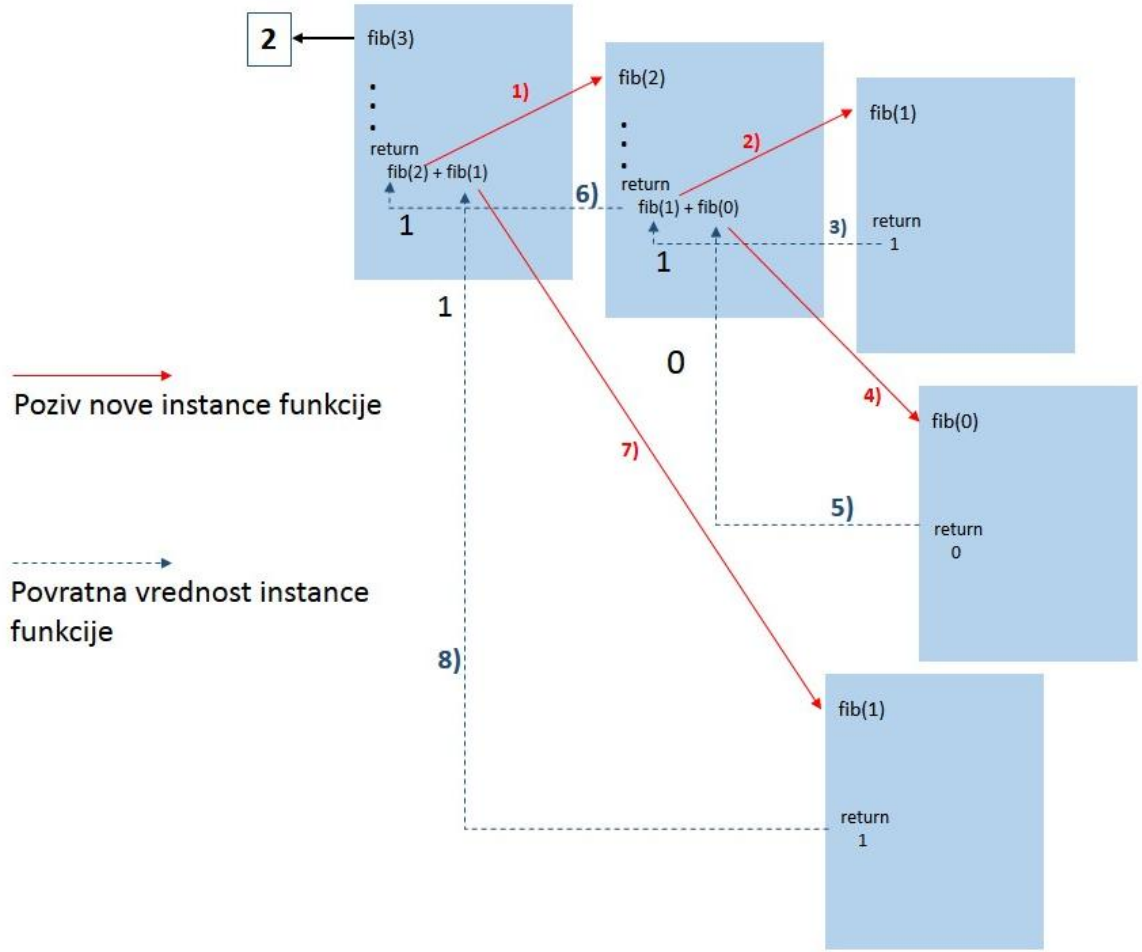
**Primer 6.** Fibonačijevi brojevi.

*Algoritam 14.* ***Funkcija za rekurzivno izračunavanje fibonačijevog broja***

```
#Pretpostavlja se da je n pozitivan ceo broj
def fib(n):

    #bazni slučaj
    if n == 0 or n == 1:
        return n

    #rekurzivni korak
    else:
        return fib(n-1) + fib(n-2)
```

Može se primetiti da je ovo primer rekurzivne funkcije sa više baznih slučajeva, tačnije sa dva bazna slučaja. Ukoliko je uneti parametar 0 ili 1, funkcija *fib* će napraviti samo jednu instancu koja će vratiti 0 ili 1 (u zavisnosti od unetog parametra).

Na slici 17. prikazan je redosled izvršavanja instanci funkcije *fib* ukoliko je uneti parametar broj 3.

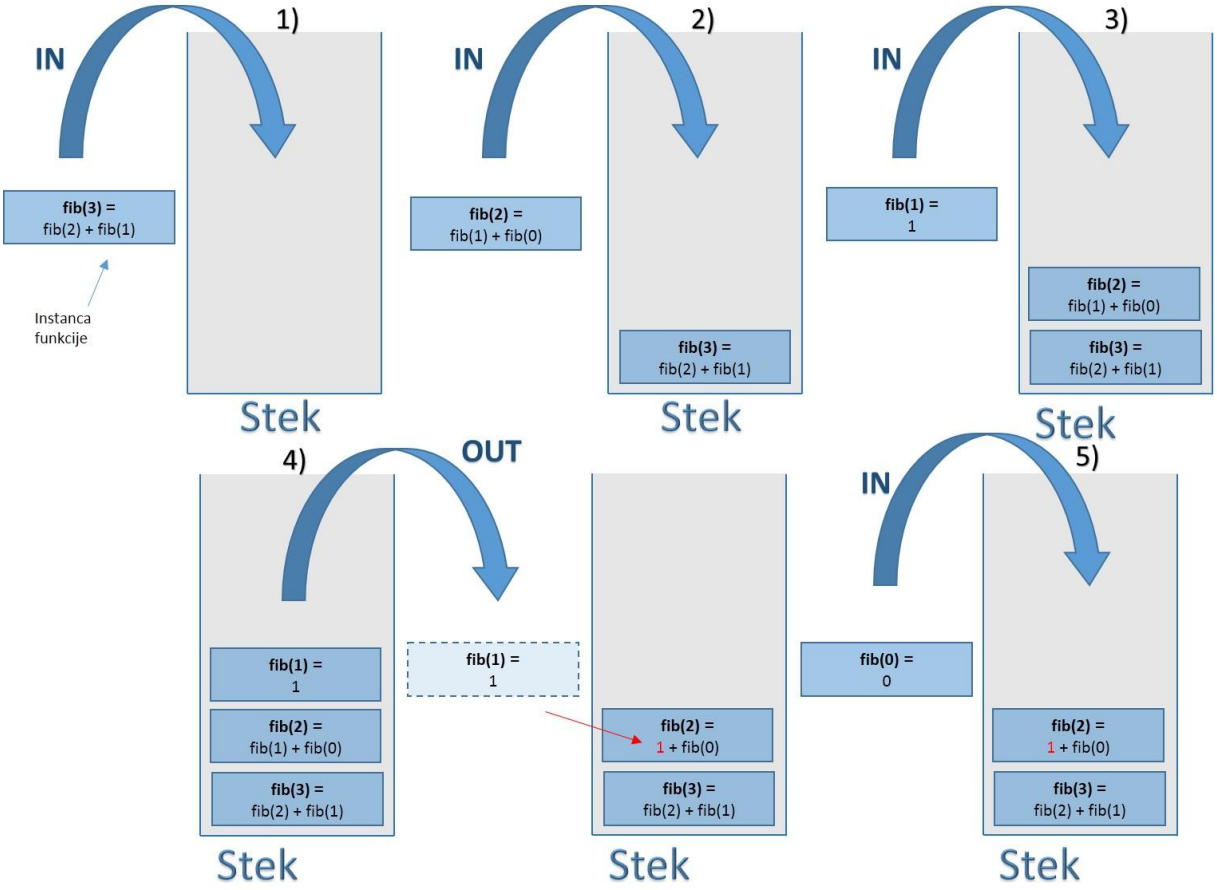



*Slika 17. Redosled izvršavanja instanci*

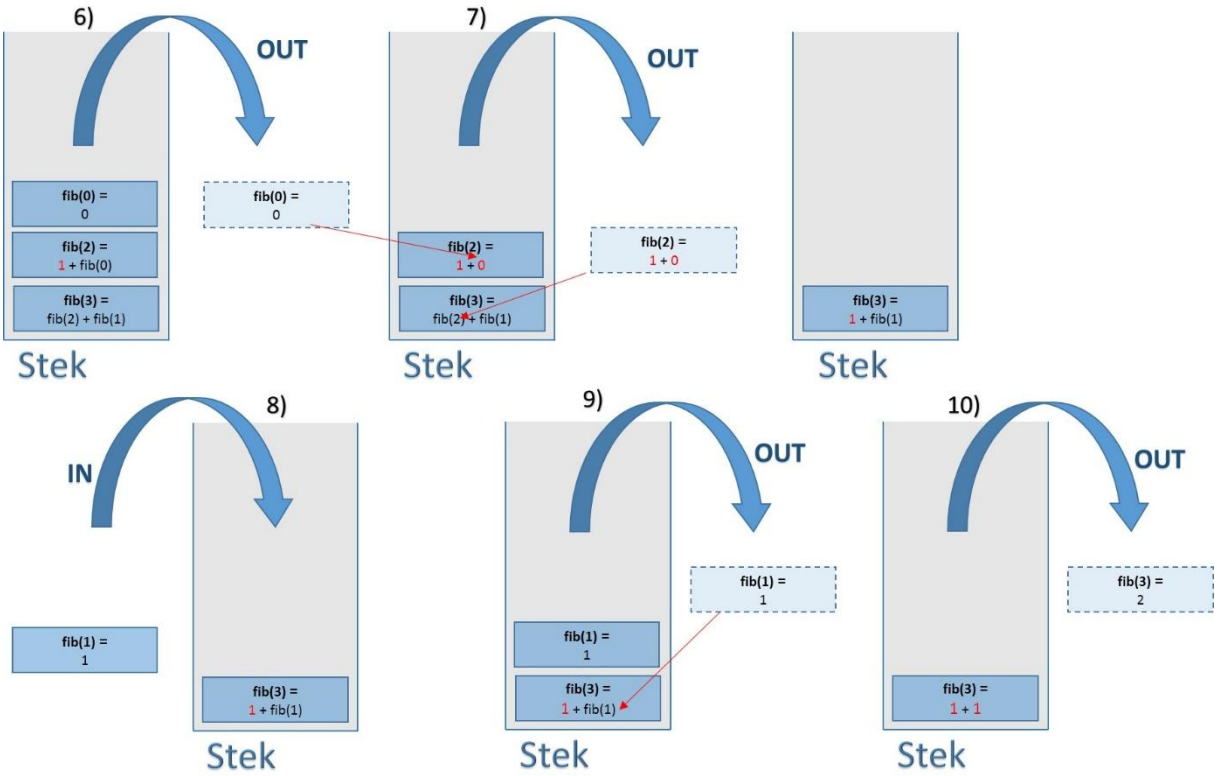
Prilikom pozivanja funkcije *fib(3)*, dolazi do rekurzivnog poziva i kreiranja instance *fib(2)* (*korak 1*), zatim iz nove instance *fib(2)* dolazi do rekurzivnog poziva i kreiranja instance *fib(1)* (*korak 2*) koja će na kraju svog izvršavanja vratiti vrednost 1 (*korak 3*). Nakon vraćanja ove vrednosti, dolazi do drugog rekurzivnog poziva u okviru instance *fib(2)* i pri tom se kreira instanca *fib(0)* (*korak 4*) koja predstavlja drugi sabirak u rekurzivnom slučaju *fib(1) + fib(0)*. Kada instanca *fib(0)* vrati vrednost 0 (*korak 5*), računa se povratna vrednost *fib(1) + fib(0)* = 1 + 0 = 1, koja predstavlja povratnu vrednost instance *fib(2)* (*korak 6*).

Nakon dobijanja vrednosti za *fib(2)*, dolazi do drugog rekurzivnog poziva u instanci *fib(3)* i pri tom se kreira instanca *fib(1)* (*korak 7*). Zatim, instanca *fib(1)* vraća vrednost 1 (*korak 8*) nakon čega se može izračunati zbir *fib(2) + fib(1)* = 1 + 1 = 2, a vrednost 2 je upravo povratna vrednost pozvane funkcije *fib(3)*.

Prilikom poziva funkcije *fib(3)*, stek memorija će se popunjavati na način ilustrovan na slici 18.

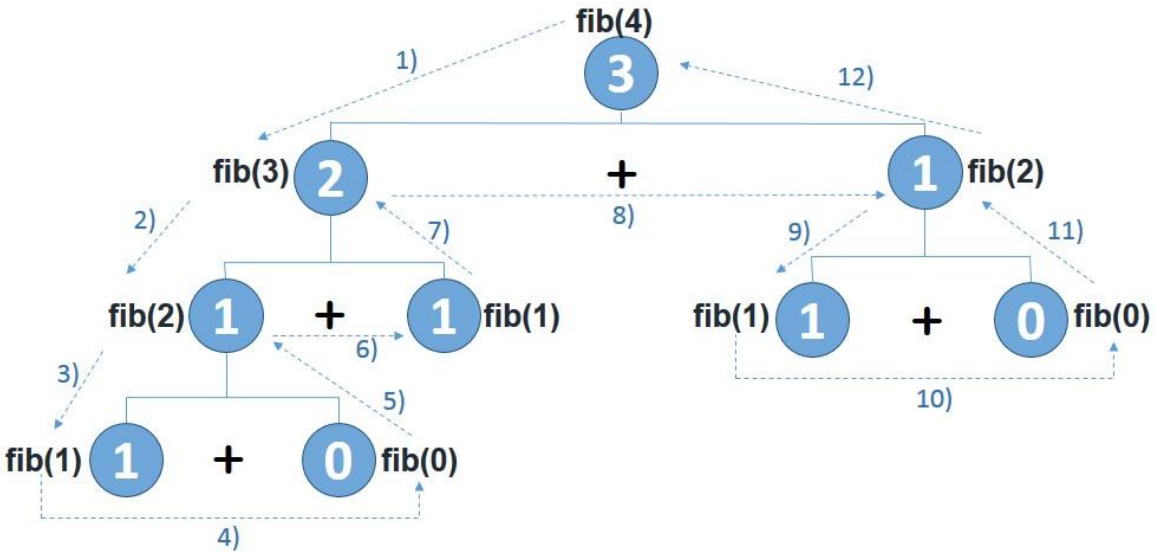



*Slika 18. Popunjavanje steka*




*Slika 19. Popunjavanje steka (nastavak)*

Izračunavanje funkcije *fib(4)* se može ilustrovati na sledeći način:




*Slika 20. Redosled izvršavanja*

## 6.1 Iterativni i rekurzivni način za izračunavanje Fibonačijevog broja

Kao što je napomenuto u poglavlju „Dobre i loše strane rekurzije“, izračunavanje Fibonačijevog broja preko rekurzivnog algoritma je vremenski i memorijski dosta zahtevnije u odnosu na iterativni način izračunavanja, iako je definicija sama po sebi rekurzivna. Na primer, za izračunavanje *fib(20)* je potrebno 21 891 poziva funkcije *fib()*, dok se za računanje *fib(30)* funkcija *fib()* poziva 2 692 537 puta [20].

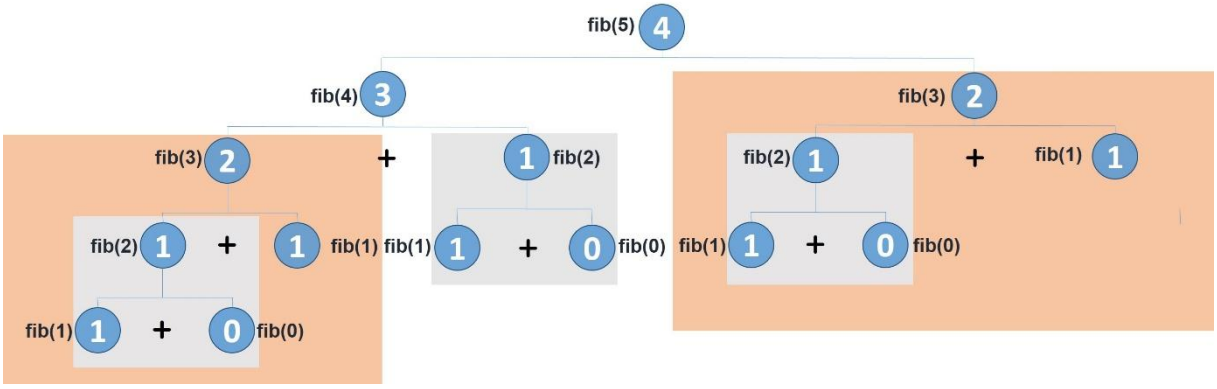
*Algoritam 15.* ***Rekurzivna funkcija***

```
def fib(n):
    #Bazni slučaj
    if n == 0 or n == 1:
        return n

    #Rekurzivni korak
    else:
        return fib(n-1) + fib(n-2)
```

*Algoritam 16.* ***Iterativna funkcija***

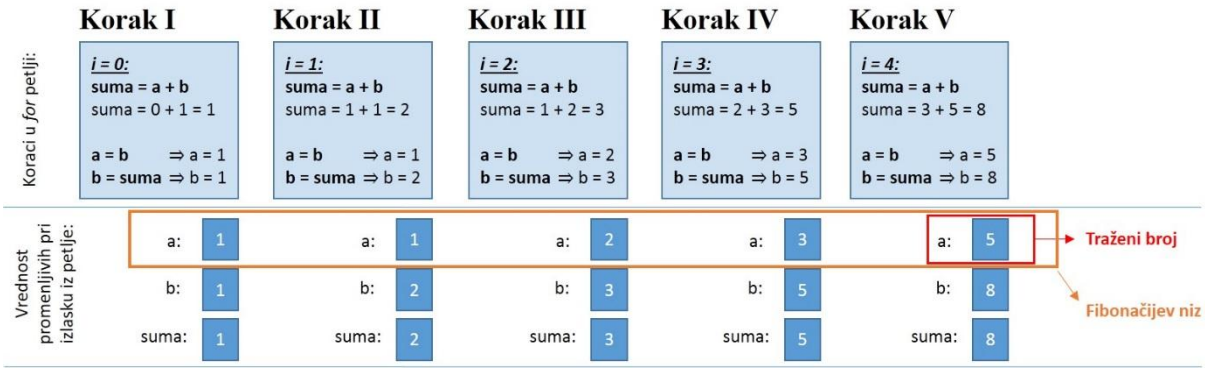
```
def fib(n):
    #Definisanje prvog i drugog
    #elementa Fibonačijevog niza
    a = 0
    b = 1

    #Iteracija preko for petlje
    for i in range(n):
        suma = a + b
        a = b
        b = suma
    return a
```

Iako rekurzivna funkcija ima kraći i elegantniji zapis, na sledećoj slici se može uvideti mana rekurzivnog načina izračunavanja Fibonačijevog broja. Prilikom razlaganja problema na manje potprobleme dolazi do preklapanja potproblema i do višestrukih poziva istih funkcija što je memorijski i vremenski dosta zahtevno.




*Slika 21. Rekurzivni način izračunavanja Fibonačijevog broja 5*

Pozivom iterativne funkcije *fib(5)* kreira se samo jedna instanca funkcije u kojoj se *for* petlja koristi za izračunavanje Fibonačijevog broja. Početne vrednosti promenljivih *a* i *b* su 0 i 1, respektivno, i one predstavljaju vrednosti prvog i drugog člana Fibonačijevog niza. Koraci u *for* petlji su šematski prikazani na slici 21.



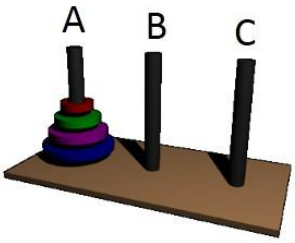

*Slika 22. Koraci u for petlji*

# 7 Kule Hanoja

Jedan od najpoznatijih primera rekurzije u kompjuterskoj literaturi je problem kule Hanoja. Ovaj problem je opisao francuski matematičar De Parvil 1884. godine. Problem predstavljaju tri štapa A, B i C (slika 22.) koja su zabodena na ravnoj podlozi. Na jednom štapu nalazi se *n* kolutova različitih prečnika poređanih po veličini, na dnu se nalazi kolut najvećeg prečnika, a na vrhu kolut najmanjeg prečnika. Cilj je preneti sve kolutove sa jednog na drugi štap uz poštovanje sledećih pravila:

- Dozvoljeno je premestiti samo jedan kolut u toku jednog poteza;
- Nije dozvoljeno staviti veći kolut preko manjeg;
- Svaki od štapova može se koristiti za privremeno smeštanje kolutova.

Cilj je odrediti najmanji broj prenosa potreban da se svih n kolutova prenese s prvog na drugi štap [21].




*Slika 23. Primer kule Hanoja*

## 7.1 Legenda o kuli Hanoja

*„Legenda kaže da je pre mnogo godina, negde u okolini grada Hanoj, živeo car koji je tražio novog dvorskog mudraca. Pošto je i sam bio mudar, želeo je da pronađe što boljeg mudraca, pa je rešio da uzme onoga koji da najbolje rešenje postavljenog problema (zagonetke): Dato je tri štapa i n diskova različitog prečnika. Svi diskovi su postavljeni na prvom štapu tako da se uvek manji disk nalazi iznad većeg. Potrebno je prebaciti sve diskove sa izvornog štapa na ciljni, premeštajući jedan po jedan disk, koristeći treći štap kao pomoćni, ali da se ni u jednom momentu disk većeg prečnika ne nađe na disku manjeg prečnika. Mnogi mudraci iz cele zemlje dolazili su pred cara sa raznim rešenjima, ali su rešenja bila ili nerazumljiva ili predugačka. „Mora da postoji jednostavniji način“, razmišljao je car. Jednog dana je pred cara stigao Buda, i rekao da je problem toliko jednostavan da se rešava sam od sebe. Svoje rešenje Buda je izložio na sledeći način: 1. Ako postoji samo jedan disk, pomeramo ga sa izvornog na ciljni štap, i to je toliko jednostavan posao da ga može uraditi svaka seoska luda. 2. Ako pak ima više od jednog diska postupak je sledeći: premestimo prvo n-1 diskova sa izvornog na pomoćni štap, koristeći ciljni kao pomoćni. Pošto je n-1 diskova na pomoćnom štapu, a najveći je i dalje ostao na izvornom, problem se svodi na tačku 1), odnosno treba*

*prebaciti taj jedan disk sa izvornog na ciljni štap. Potom treba n-1 diskova, sa pomoćnog štapa prebaciti na ciljni po istom postupku (sada koristeći izvorni štap kao pomoćni). Kada je Buda završio sa pričom, car ga je pitao kada će konačno reći svoje rešenje. Buda se samo nasmešio i otišao"* - preuzeto iz skripte „Uvod u programiranje" PMF Novi Sad [22].

Iterativno rešenje ovog problema je veoma kompleksno, a rekurzivno prilično jednostavno. Ukoliko je n = 1, prebaciti disk sa polaznog na dolazni štap. Ukoliko nije, prebaciti (rekurzivno) n-1 diskova sa polaznog na pomoćni štap korišćenjem dolaznog štapa kao pomoćnog (*korak 1*), prebaciti najveći disk sa polaznog na dolazni štap (*korak 2*) i na kraju prebaciti (rekurzivno) n-1 diskova sa pomoćnog na dolazni štap korišćenjem polaznog štapa kao pomoćnog (*korak 3*) (slika 23.) [23].


*Slika 24. Koraci za primer kule Hanoja*

U programskom jeziku Pajton, primer kule Hanoja je predstavljen na sledeći način:

*Algoritam 17.* ***Kule Hanoja***

```
#f-ja Kule za argumente uzima broj diskova (n) i nazive štapova
#(polazni, dolazni i pomocni)

def Kule(n, polazni, dolazni, pomocni):

    #bazni slučaj. Ako postoji samo jedan disk, izvršiti prebacivanje
    if n == 1:
        print ("Pomeri sa " + polazni + " na " + dolazni)

    #rekurzivni korak
    else:
        #prvo se prebacuje n-1 diskova sa polaznog na pomoćni štap
        Kule(n-1, polazni, pomocni, dolazni)

        #najveći disk se prebaci sa polaznog na dolazni štap
        Kule(1, polazni, dolazni, pomocni)

        #na kraju, n-1 diskova se prebacuje sa pomoćnog na dolazni štap
        Kule(n-1, pomocni, dolazni, polazni)

    #poziv funkcije za ispisivanje rešenja
Kule(3, "A", "C", "B")
```

Prilikom poziva rekurzivne funkcije Kule prvo se proverava da li je uneti argument n jednak jedinici i ukoliko jeste, program ispisuje rešenje problema (sa kog polaznog štapa na koji dolazni štap treba prebaciti disk).

Ukoliko je uneti argument n veći od jedan, rekurzivnim pozivom funkcije Kule se prvo prebaci n-1 diskova sa polaznog štapa na pomoćni, zatim pomoću rekurzivne funkcije sa argumentom n = 1 se izvrši prebacivanje najvećeg diska sa polaznog na dolazni štap i na kraju se poslednjom rekurzivnom funkcijom sa argumentom n-1 prebaci preostali broj diskova sa pomoćnog na dolazni štap. Nakon završetka funkcije Kule dobiće se ispisano rešenje za prebacivanje n diskova sa jednog štapa uz pomoć trećeg štapa.

Program će ispisati sledeće:

```
Pomeri sa A na C
Pomeri sa A na B
Pomeri sa C na B
Pomeri sa A na C
Pomeri sa B na A
Pomeri sa B na C
Pomeri sa A na C
```

# 8 Dodatni primeri

**Primer 7.** Rekurzivni program za računanje proizvoda dva broja.

Množenje dva prirodna broja preko rekurzivnog algoritma se može predstaviti na sledeći način:

$$a \cdot b = a + \underbrace{a + a + \cdots + a}_{b-1}$$
$$= a + a + \underbrace{a + \cdots + a}_{b-2}$$
$$= \cdots$$

**Rešenje:**

*Algoritam 18.* ***Rekurzivno množenje***

```
def rekurzivno_mnozenje(a, b):
    if b  == 1:
        return a
    else:
        return a +  rekurzivno_mnozenje(a, b-1)
```

*Algoritam 19.* ***Poziv funkcije za rekurzivno množenje***

```
print rekurzivno_mnozenje(2, 3)
```

Program će raditi sledeće:

```
rekurzivno_mnozenje(2, 3)
= 2 +  rekurzivno_mnozenje(2, 2)
= 2 + 2 + rekurzivno_mnozenje(2, 1)
= 2 + 2 + 2
= 2 + 4
= 6
```

**Primer 8.** Program koji računa sumu svih elemenata iz liste

Algoritam koristi isecanje liste kako bi došao do baznog slučaja. Bazni slučaj je lista dužine jedan.

**Rešenje:**

*Algoritam 20.* ***Suma elemenata iz liste***

```
def suma_liste(lista):
    if len(lista) == 1:
        return lista[0]
    else:
        return lista[0] + suma_liste(lista[1:])
```

*Algoritam 21.* ***Poziv funkcije za sumu liste***

```
print suma_liste([1,3,5,7,9])
```

Program radi sledeće:

```
suma_liste([1,3,5,7,9])
= 1 +  suma_liste([3,5,7,9])
= 1 + 3 + suma_liste([5,7,9])
= 1 + 3 + 5 + suma_liste([7,9])
= 1 + 3 + 5 + 7 + suma_liste([9])
= 1 + 3 + 5 + 7 + 9
= 1 + 3 + 5 + 16
= 1 + 3 + 21
= 1 + 24
= 25
```

**Primer 9.** Program za stepenovanje dva prirodna broja.

Postoje dva rekurzivna algoritma za stepenovanje dva prirodna broja koja se razlikuju u brzini kompajliranja.

**Rešenje:**

*Algoritam 22.* ***Sporo stepenovanje dva prirodna broja***

```
def rstepen_sporo(x, n):
    if n == 0:
        return 1
    else:
        return x * rstepen_sporo(x, n-1)
```

*Algoritam 23.* ***Poziv funkcije za sporo stepenovanje dva prirodna broja***

```
print rstepen_sporo(2, 4)
```

Program će raditi sledeće:

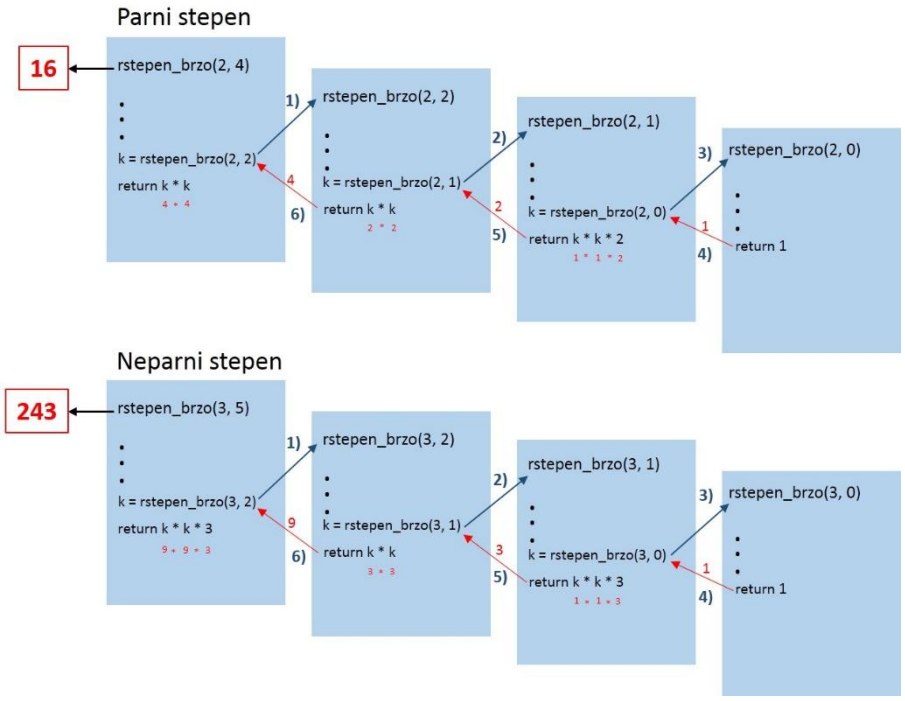
```
rstepen_sporo(2, 4)
= 2 * rstepen_sporo(2, 3)
= 2 * 2 * rstepen_sporo(2, 2)
= 2 * 2 * 2 * rstepen_sporo(2, 1)
= 2 * 2 * 2 * 2 * rstepen_sporo(2, 0)
= 2 * 2 * 2 * 2 * 1
= 2 * 2 * 2 * 2
= 2 * 2 * 4
= 2 * 8
= 16
```

*Algoritam 24.* ***Brzo stepenovanje dva prirodna broja***

```
def rstepen_brzo(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        k = rstepen_brzo(x, n/2)
        return k * k
    elif n % 2 == 1:
        k = rstepen_brzo(x, n/2)
        return k * k * x
```

*Algoritam 25.* ***Poziv funkcije za brzo stepenovanje dva prirodna broja***

```
print rstepen_brzo(2, 4)
print rstepen_brzo(3, 5)
```




*Slika 25. Brzo stepenovanje dva prirodna broja*

# 9 Zaključak

U ovom radu kroz primere rekurzivnih funkcija korišćenjem programskog jezika Pajton prikazan je nov kreativan primer udžbenika za učenike srednjih škola i generalno početnika u programiranju. Svi primeri su dostupni i mogu se rešavati na veb adresi http://edusoft.math.rs/python/. Putem ove interakcije sam materijal postaje interesantniji i lakši za savladavanje. Prikaz programskog jezika Pajton kroz elektronski kurs bi trebalo da privuče početnike koji su na samom početku svoje programerske karijere, da ga koriste kao prvi programski jezik na kome će naučiti sve osnove programiranja. U okviru elektronskog kursa obuhvaćeni su osnovni koncepti i načini razmišljanja koji su neophodni za prve programerske korake. Slikovitim i šematskim prikazivanjem određenih funkcija učenicima i početnim programerima je omogućeno lakše razumevanje samog rada programa.

Pored isticanja dobrih strana programskog jezika Pajton, jedan od ciljeva je bio savladavanje rekurzivnog razmišljanja, odnosno determinisanja problema koji su pogodni za rešavanje putem rekurzivnih funkcija. Navedeni su primeri od kojih neki predstavljaju dobar i konstruktivan način korišćenja rekurzije (Kule Hanoja), a neki loš i nefunkcionalan (Fibonačijev niz).

Razvoj tehnologije i interneta omogućio je korišćenje ovog i njemu sličnih kurseva „od kuće“, uglavnom korišćenjem besplatnih programa ili direktnom pristupu internetu čime se podstiče samostalno učenje i učenje otkrivanjem kao jednim od najboljih vidova učenja.

# Literatura


[1] Vasiljević N, *Zašto je programski jezik Pajton dobar za učenje programiranja*, INFOteka, 2013.

[2] http://viewsource.rs/uvod-u-python-part-1/ (pristupljeno 15. novembra 2014.)

[3] http://www.it-modul.rs/09/2012/python-programski-jezik/ (pristupljeno 15. novembra 2014.)

[4] Pilgrim M, *Dive Into Python*, Apress, 2004.

[5] Dawson M, *Python programming for the Absolute Beginner, 3rd Edition*, Course Technology, 2010.

[6] Swaroop C H, *A Byte of Python*, 2014. (http://www.swaroopch.com/notes/python/; pristupljeno 15. novembra 2014.)

[7] http://c2.com/cgi/wiki?PythonProblems (pristupljeno 15. novembra 2014.)

[8] Essert M, *Digitalni udžbenik Python osnove*, Osijek, Odjel za matematiku Sveučilišta Josipa Jurja Strossmayera, 2007.

[9] Marić F, Janičić P, Programiranje II Beleške za predavanja. Smer Informatika, Matematički fakultet, Beograd, 2011. (http://poincare.matf.bg.ac.rs/~filip/p2i/p2.pdf; pristupljeno 15. novembra 2014.)

[10] Kraus L, *Programski jezik C sa rešenim zadacima*, Beograd, Akademska misao, 2001.

[11] Mateljan I, *Materijal sa predavanja OOP sa C++ jezikom*, Split, 2007. (http://marjan.fesb.hr/~mateljan/cpp/slides13-rekurzija.pdf; pristupljeno 15. novembra 2014.)

[12] Dierbach C, *Introduction to Computer Science Using Python: A Computational Problem-Solving Focus*. John Wiley & Sons, Inc., 2013

[13] http://jelenagavanski.wordpress.com/category/racunari/ (pristupljeno 15. novembra 2014.)

[14] Miller B, Ranum D, *How to Think Like a Computer Scientist Learning with Python: Interactive Edition 2.0*, 2014. (http://interactivepython.org/courselib/static/thinkcspy/Recursion/TheThreeLawsofRecursion.html; pristupljeno 15. novembra 2014.)

[15] Trifunović D, Kratka priča o faktorijelu, *Nastava Matematike*, 1999, XLIV_1-2, 54-57 (http://elib.mi.sanu.ac.rs/files/journals/nm/214/nm441210.pdf; pristupljeno 15. novembra 2014.)

[16] Materijal sa predavanja, Kornel Univerzitet, 1998. (http://www.cs.cornell.edu/info/courses/spring-98/cs211/lecturenotes/07-recursion.pdf; pristupljeno 15. novembra 2014.)

[17] Materijal sa predavanja, Havaji Univerzitet (http://www2.hawaii.edu/~tp_200/lectureNotes/recursion.htm; pristupljeno 15. novembra 2014.)

[18] Lučić Z, *Ogledi iz istorije antičke geometrije*, JP Službeni glasnik, 2009.

[19] http://www.python-course.eu/recursive_functions.php (pristupljeno 15. novembra 2014.)

[20] Materijal sa predavanja Programiranje II, Univerzitet u Beogradu (http://www.alas.matf.bg.ac.rs/~mn06066/Predavanja%20II/PII4.ppt; pristupljeno 15. novembra 2014.)

[21] Oreški I, *Rekurzije*, Završni rad, Osijek, 2011. (http://www.mathos.unios.hr/~ioreski/Seminari/Rekurzije.pdf; pristupljeno 15. novembra 2014.)
[22] Tošić S, *Skripta: Uvod u programiranje – Glava 6 Procedure*, Univerzitet u Novom Sadu, 2013. (http://perun.pmf.uns.ac.rs/tosic/UUP06v13.pdf; pristupljeno 15. novembra 2014.)
[23] Online kurs *MITx: 6.00.1x Introduction to Computer Science and Programming* (https://courses.edx.org/courses/MITx/6.00.1x/3T2013/courseware/sp13_Week_3/videosequence:Lecture_5/; pristupljeno 15. novembra 2014.)