# Rekurzija u nastavi informatike

**Wrigley-Pimley-McKerr, Karmela**

**Master's thesis / Diplomski rad**

**2023**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:217:159135

*Rights / Prava:* In copyright / Zaštićeno autorskim pravom.

*Download date / Datum preuzimanja:* **2024-12-08**

*Repository / Repozitorij:*

Repository of the Faculty of Science - University of Zagreb

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Sveučilište u Zagrebu
Prirodoslovno-matematički fakultet
Matematički odsjek

Karmela Wrigley-Pimley-McKerr

# Rekurzija u nastavi informatike

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Vedran Krčadinac

Zagreb, travanj 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____ , predsjednik
2. _____ , član
3. _____ , član

Povjerenstvo je rad ocijenilo ocjenom _____ .

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

# Sadržaj

*Posvećujem III., IV. i III., mojim suputnicima na ovom putu.*

# 1 Uvod

U ovom radu definirat ćemo rekurziju i dati primjere rekurzivnih algoritama pisanih u programskom jeziku Python. Dat ćemo rekurzivno rješenje za računanje potencije broja dva i računanje faktorijela od zadanog prirodnog broja $n$. Definirat ćemo Fibonaccijev niz te napisati algoritme koji rekurzivno i iterativno računaju Fibonaccijeve brojeve. Rekurzivno rješenje popravit ćemo memoizacijom te ćemo usporediti vremena izvršavanja tih algoritama. Dalje ćemo definirati binomne koeficijente, povezati ih s Pascalovim trokutom te ćemo dokazati Pascalovu formulu. Binomne koeficijente izračunat ćemo rekurzivno te ćemo rekurzivnu funkciju unaprijediti memoizacijom i na kraju ćemo napisati program koji ih računa bez rekurzije. U tom pristupu započet ćemo s osnovnim slučajevima i zatim graditi rješenje iterativno pomoću tablice. Na kraju ćemo usporediti vrijeme izvršavanja ta tri algoritma.

Prikazat ćemo rekurzivne algoritme za generiranje kombinatornih objekata. Generirat ćemo permutacije, kombinacije te sve podskupove danog skupa. Rekurzija se koristi u crtanju fraktalnih skupova. Navest ćemo fraktale koji se najčešće spominju u nastavi kao što su Kochova krivulja, binarno stablo i trokut Sierpińskog. Fraktale ćemo crtati u programskom jeziku Python služeći se kornjačinom grafikom.

Mnogi algoritmi za sortiranje su rekurzivne prirode. Opisat ćemo dva: mergesort i quicksort te pokazati njihovo djelovanje na primjeru. Testirat ćemo njihovu brzinu u usporedbi s ugrađenom funkcijom u programskom jeziku Python te analizirati njihovu vremensku složenost.

Poglavlje rekurzija u školi započet ćemo prolaskom kroz kurikulum nastavnog predmeta Informatike. Iz kurikuluma ćemo saznati u kojem razredu i koliko detaljno se obrađuje rekurzija te koje su preporuke za obradu gradiva rekurzije u nastavi. Prikazat ćemo kako udžbenici za osnovnu i srednju školu obrađuju tu temu te što oni smatraju bitnim. U udžbenicima za osnovnu školu susret ćemo se s pojmom računalno razmišljanje. Provest ćemo korake računalnog razmišljanja na problemu Hanojskih tornjeva te ćemo izračunati broj poteza potrebnih za rješenje problema Hanojskih tornjeva s $n$ diskova. Prisjetit ćemo se legende o šahu te izračunati broj zrna koji se dobije ako se na svako naredno polje šaha stavi dvostruko više zrna nego na prethodno polje. Iz udžbenika za srednju školu izdvojit ćemo zanimljive problemske zadatke koji se rješavaju rekurzivnom funkcijom kao što su problem dobro sparenih zagrada, crtanje Pitagorina stabla i Collatzova slutnja.

Na kraju ćemo dati osvrt na udžbenike te primjer obrade rekurzije u nastavi s obzirom na uzrast.

# 2 Primjeri rekurzivnih algoritama

Rekurzija je tehnika rješavanja problema u kojoj se problemi rješavaju reduciranjem na manje probleme istog oblika. Rekurziju kao tehniku susrećemo u radu s rekurzivnim funkcijama. Rekurzivna funkcija je definirana u terminima osnovnih slučajeva i rekurzivnih koraka. U osnovnom slučaju, izračunavamo rezultat odmah s obzirom na ulaze u poziv funkcije. U rekurzivnom koraku izračunavamo rezultat uz pomoć jednog ili više rekurzivnih poziva te iste funkcije, ali s unosima smanjene veličine ili složenosti, bliže osnovnom slučaju [2].

Rekurzija svoju najveću primjenu ima u računarstvu. Rekurzija rješava rekurzivne probleme korištenjem funkcija koje pozivaju same sebe iz vlastitog koda. Rekurzivna implementacija u programskom jeziku uvijek ima dva dijela. Prvi dio je osnovni slučaj, koji je najjednostavniji, najmanji primjer problema, koji se može direktno riješiti. Osnovni slučajevi često odgovaraju praznini – prazan niz, prazna lista, prazan skup, nula itd. Zanemarivanje pisanja osnovnog slučaja ili njegovo neispravno testiranje može uzrokovati beskonačnu petlju.

Drugi dio je rekurzivni korak, koji rastavlja veći problem na jedan ili više jednostavnijih ili manjih problema koje se mogu riješiti rekurzivnim pozivima, a zatim rekombinira rezultate tih potproblema kako bi proizveo rješenje izvornog problema. Za rekurzivni korak važno je transformirati početni problem u nešto manji problem, inače rekurzija možda nikada neće završiti. U pravilno dizajniranoj rekurziji sa svakim rekurzivnim pozivom, problem unosa mora biti pojednostavljen na takav način da se na kraju dolazi do osnovnog slučaja. Ako svaki rekurzivni korak smanjuje problem doći ćemo do osnovnog slučaja te je tada rekurzija konačna.

Rekurzivna implementacija može imati više od jednog osnovnog slučaja ili više od jednog rekurzivnog koraka. Na primjer, funkcija koja izračunava Fibonaccijev niz ima dva osnovna slučaja, $n = 0$ i $n = 1$.

Rekurzija nije prikladna za svaki problem, no veliki broj problema u matematici i informatici ima rekurzivna rješenja. U ovom dijelu diplomskog rada proučavat ćemo probleme iz matematike te dati njihova rekurzivna rješenja. Rješenja će biti prikazana u programskom jeziku Python.

## 2.1 Izračunavanje funkcija

Određene jednostavne matematičke probleme, koji imaju i svoje rekurzivno rješenje, riješit ćemo pomoću programskog jezika Python. Problemi će biti riješeni iterativno i rekurzivno te ćemo usporediti učinkovitost jednog i drugog pristupa.

### 2.1.1 Potencije broja 2

Potenciju $2^n$ možemo dobiti tako da $n$ puta pomnožimo broj 2 sa samim sobom. Rješenje bez rekurzije je:

```
n = int(input('Unesi n '))
baza = 2
rezultat = 1

for i in range (n):
    rezultat = rezultat * baza

print('Rezultat je ', rezultat)
```

No, znamo i da $2^n$ možemo zapisati kao $2^n = 2 \cdot 2^{n-1}$. Na taj smo način originalni problem razbili na jednostavniji problem. Znači, ako znamo izračunati $2^{n-1}$ tada bismo jednostavno tu vrijednost pomnožili s 2 i riješili originalni problem. Osnovni slučaj rekurzije je $n = 0$, tada je rješenje 1. Korak će biti da izračunamo $2^{n-1}$. Time smo problem rastavili na manji.

```
def potencija(n):
    if n == 0:
        return 1
    else:
        return 2 * potencija(n - 1)

n = int(input('Unesi n '))
p = potencija(n)
print(p)
```

### 2.1.2 Faktorijeli

Faktorijel prirodnog broja $n$ je umnožak svih prirodnih brojeva manjih ili jednakih $n$. Sljedeći kod upravo tako računa faktorijel od zadanog prirodnog broja $n$.

```
n = int(input('Unesi n '))
rezultat = 1
for i in range (1, n + 1):
    rezultat = rezultat * i
```

```
print('Faktorijel broja', n, 'iznosi', rezultat)
```

Faktorijele možemo računati i rekurzivno. Rekurzivnu funkciju koja računa faktorijele zapisat ćemo na sljedeći način:

$$n! = \begin{cases} 1, & \text{ako je } n = 0, \\ (n-1)! \cdot n, & \text{ako je } n > 0. \end{cases}$$

U rekurzivnoj implementaciji osnovni slučaj je $n = 0$, gdje izračunavamo i odmah vraćamo rezultat, $0! = 1$. Rekurzivni korak je za $n > 0$, gdje izračunavamo rezultat uz pomoć rekurzivnog poziva za dobivanje $(n-1)!$, zatim dovršavamo izračun množenjem s $n$.

```
def faktorijel(n):
    if n == 0:
        return 1
    else:
        return n * faktorijel(n - 1)


n = int(input('Unesi n '))
f = faktorijel(n)
print('Faktorijel broja', n, 'iznosi', f)
```

### 2.1.3 Fibonaccijevi brojevi

Fibonaccijev niz zadan je rekurzivno na sljedeći način:

$$F_n = \begin{cases} 0, & \text{ako je } n = 0, \\ 1, & \text{ako je } n = 1, \\ F_{n-1} + F_{n-2}, & \text{ako je } n > 1. \end{cases}$$

Dakle, nakon dvije početne vrijednosti, svaki sljedeći broj je zbroj dvaju prethodnika. Uočimo da je ova definicija rekurzivna. Fibonaccijeve brojeve koji čine Fibonaccijev niz možemo računati i preko Binetove formule. Binetova formula je izraz za računanje $n$-tog Fibonaccijevog broja. Ako s $\alpha, \beta$ označimo

$$\alpha = \frac{1+\sqrt{5}}{2}, \ \beta = \frac{1-\sqrt{5}}{2}$$

tada formula glasi

$$F_n = \frac{1}{\sqrt{5}}(\alpha^n - \beta^n).$$

Rekurzivni način definiranja prirodniji je od Binetove formule. Napišimo kod u Pythonu za rekurzivnu funkciju koja će vraćati $n$-ti element Fibonaccijeva niza.

```
from time import time

def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)

n = int(input())
start_time = time()
f = fib(n)
end_time = time()
print(f)
time_taken = end_time - start_time
print(time_taken)
```

Promotrimo tablicu 1 u kojoj je prikazan tijek računanja za 6. član Fibonaccijeva niza. Uočimo kako iste vrijednosti računamo više puta, a to

<table>
<tr><td colspan="8">$F_6 = F_5 + F_4$</td></tr>
<tr><td colspan="5">$F_5 = F_4 + F_3$</td><td colspan="3">$F_4 = F_3 + F_2$</td></tr>
<tr><td colspan="3">$F_4 = F_3 + F_2$</td><td colspan="2">$F_3 = F_1 + F_2$</td><td colspan="2">$F_3 = F_1 + F_2$</td><td></td></tr>
<tr><td colspan="2">$F_3 = F_1 + F_2$</td><td rowspan="3">$F_2 = 1$</td><td>$F_2 = 1$</td><td>$F_1 = 1$</td><td>$F_2 = 1$</td><td>$F_1 = 1$</td><td>$F_2 = 1$</td></tr>
<tr><td>$F_2 = 1$</td><td>$F_1 = 1$</td><td colspan="3" rowspan="2">2</td><td colspan="2">$F_2 = 1$</td></tr>
<tr><td colspan="2">2</td><td colspan="3" rowspan="3">3</td></tr>
<tr><td colspan="5">3</td></tr>
<tr><td colspan="5">5</td></tr>
<tr><td colspan="8">8</td></tr>
</table>

Tablica 1: Prikaz poziva za izračun 6. člana Fibonaccijeva niza.

je nepotrebno. Kod će biti puno učinkovitiji ako naša rekurzivna funkcija

pamti ranije izračunate vrijednosti. Popravak koji je potreban za uklanjanje suvišnih poziva rekurzivne funkcije naziva se memoizacija [10]. Ideja je zapamtiti sve vrijednosti koje smo izračunali, tako da sljedeći put kada se od nas zatraži da izračunamo vrijednost možemo je potražiti umjesto da je ponovno izračunamo. Vrijednosti koje smo ranije izračunali pohranit ćemo u jednodimenzionalni niz. Kad god se od nas traži da izračunamo vrijednost, počinjemo s pregledom niza. Tek ako vrijednost već nije izračunata, izračunamo te onda spremimo vrijednost u niz prije nego što vratimo rezultat.

```
from time import time

niz = [0, 1]

def fib(n):
    if n < len(niz):
        return niz[n]
    else:
        niz.append(fib(n - 1) + fib(n - 2))
        return niz[n]


n = int(input())
start_time = time()
f = fib(n)
end_time = time()
print(f)
time_taken = end_time - start_time
print(time_taken)
```

Fibonaccijeve brojeve možemo izračunati i iterativno, dovoljno je pamtiti zadnja dva člana niza jer njihovom sumom dobijemo sljedeći član niza.

```
from time import time

def fib(n):
    a = 0
    b = 1
```

```
    for i in range(n):
        suma = a + b
        a = b
        b = suma
    return a

n = int(input())
start_time = time()
f = fib(n)
end_time = time()
print(f)
time_taken = end_time - start_time
print(time_taken)
```

Tablica 2 prikazuje usporedbu vremena za sva tri načina. Pogledajmo vrijeme koje nam je potrebno za bismo rekurzivno izračunali 10. član Fibonaccijevog niza, pa 30. a 45. član nećemo uspjeti ni dočekati. Iako se ovaj problem može riješiti rekurzivno, čisto rekurzivno rješenje nije efikasno i treba izbjegavati takav način rješavanja.

| | Rekurzivno | S memoizacijom | Iterativno |
|---|---|---|---|
| 10 | 0.0 | 0.0 | 0.0 |
| 15 | 0.0 | 0.0 | 0.0 |
| 20 | 0.0 | 0.0 | 0.0 |
| 25 | 0.0312573 | 0.0 | 0.0 |
| 30 | 0.3751118 | 0.0 | 0.0 |
| 35 | 4.8072667 | 0.0 | 0.0 |
| 40 | 46.3696251 | 0.0 | 0.0 |
| 45 | / | 0.0 | 0.0 |
| 100 | / | 0.0 | 0.0 |

Tablica 2: Usporedba vremena - Fibonaccijev niz.

### 2.1.4 Binomni koeficijenti

Binomni koeficijent $\binom{n}{k}$ je broj $k$-članih podskupova skupa od $n$ elemenata. Pojavljuje se kao koeficijent u razvoju binoma $(x + y)^n$ te se zato naziva binomnim koeficijentom.

Kada binomne koeficijente poredamo u trokutastu formu dobijemo Pascalov trokut:

$$
\begin{array}{ccccccccccc}
 & & & & 1 & & & & & & \\
 & & & 1 & & 1 & & & & & \\
 & & 1 & & 2 & & 1 & & & & \\
 & 1 & & 3 & & 3 & & 1 & & & \\
1 & & 4 & & 6 & & 4 & & 1 & & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

Konstrukcija Pascalovog trokuta temelji se na Pascalovoj formuli.

**Propozicija 2.1.** *(Pascalova formula) Za $n, k \in \mathbf{N}, 1 \leq k \leq n-1$ vrijedi*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{1}$$

*Dokaz.* Neka je $S$ skup od $n$ elemenata, a $x$ jedan istaknuti element skupa $S$. Podskupove skupa $S$ koji imaju točno $k$ elemenata možemo rastaviti u dvije disjunktne klase prema tome sadrže li element $x$ ili ne. U klasi $A$ nalaze se oni podskupovi koji sadrže element $x$, a u klasi $B$ oni podskupovi koji ne sadrže element $x$. Budući da su $A$ i $B$ disjunktni, vrijedi

$$\binom{n}{k} = |A| + |B|.$$

Prebrojimo sad broj podskupova u klasi $A$. Budući da svaki od njih sadrži element $x$, tih podskupova ima onoliko na koliko načina možemo izabrati preostalih $k-1$ elemenata iz ostatka skupa $S$, jer je jedan element, $x$, već izabran. Zato je

$$|A| = \binom{n-1}{k-1}.$$

Skup $B$ sastoji se od onih podskupova veličine $k$ koji ne sadrže element $x$. To znači da elemente tih podskupova možemo izabrati između $n-1$ elementa skupa $S$. Zato je

$$|B| = \binom{n-1}{k}.$$

Prema tome, vrijedi:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

□

U Pascalovom trokutu novi element dobivamo zbrojivši dva elementa prethodnog retka koji su lijevo i desno od njega. Izuzetak su rubovi na kojima su uvijek jedinice.

Za binomni koeficijent $\binom{n}{k}$ vrijedi formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Po konvenciji je $0! = 1$ te imamo dva posebna slučaja:

$$\binom{n}{1} = \frac{n!}{1!(n-1)!} = \frac{n!}{(n-1)!} = n,$$
$$\binom{n}{0} = \frac{n!}{0!(n-0)!} = \frac{n!}{1 \cdot n!} = \frac{n!}{n!} = 1.$$

Svojstvo simetrije glasi

$$\binom{n}{k} = \binom{n}{n-k}.$$

Zbog svojstva simetrije vrijedi

$$\binom{n}{n} = \binom{n}{n-n} = \binom{n}{0} = 1$$

te

$$\binom{n}{n-1} = \binom{n}{n-(n-1)} = \binom{n}{1} = n.$$

Binomne koeficijente mogli bismo izračunati tako da izračunamo sve uključene faktorijele i zatim podijelimo brojnik s nazivnikom. No, takav način računanja nije praktičan čak ni za umjereno velike vrijednosti $n$ jer faktorijeli rastu vrlo brzo. Umjesto toga, možemo koristi rekurziju (1).

```
from time import time

def C(n, k):
  if k == 0 or k == n:
    return 1
  else:
    return C(n - 1, k - 1) + C(n - 1, k)

n = int(input('Unesi n '))
k = int(input('Unesi k '))
start_time = time()
rezultat = C(n,k)
end_time = time()
```

```
print(rezultat)
time_taken = end_time - start_time
print(time_taken)
```

Izračunata vrijednost je točna, ali ovakav način računanja nije efikasan. Izračunajmo binomni koeficijent $\binom{8}{5}$.

$$
\begin{gathered}
C(8,5) \text{ poziva } C(7,4) \text{ i } C(7,5) \\
C(7,4) \text{ poziva } C(6,3) \text{ i } C(6,4) \\
C(7,5) \text{ poziva } C(6,4) \text{ i } C(6,5) \\
C(6,3) \text{ poziva } C(5,2) \text{ i } C(5,3) \\
C(6,4) \text{ poziva } C(5,3) \text{ i } C(5,4) \\
C(6,4) \text{ poziva } C(5,3) \text{ i } C(5,4) \\
C(6,5) \text{ poziva } C(5,4) \text{ i } C(5,5) \\
\vdots
\end{gathered}
$$

Uočimo kako se iste vrijednosti računaju više puta. Ovaj učinak postaje toliko ozbiljan za velike $n$ i $k$ da prikazana jednostavna rekurzivna funkcija postaje neučinkovita. Rekurzivnu funkciju unaprijedit ćemo memoizacijom. Koristit ćemo tablicu, tj. dvodimenzionalno polje za čuvanje izračunatih vrijednosti funkcije. Kad god se od nas traži da izračunamo vrijednost, počinjemo s pregledom tablice. Vrijednost ćemo računati samo ako već nije izračunata i pohranjena u tablici, a kada ju izračunamo spremit ćemo je u tablicu.

```
from time import time

n = int(input('Unesi n '))
k = int(input('Unesi k '))
c = [[0 for i in range(n + 1)] for j in range(n + 1)]

def C(n, k):
  if c[n][k] == 0:
    if k == 0 or k == n:
      c[n][k] = 1
    else:
      c[n][k] = C(n - 1, k - 1) + C(n - 1, k)
  return c[n][k]
```

```
start_time = time()
rezultat = C(n, k)
end_time = time()
print(rezultat)
time_taken = end_time - start_time
print(time_taken)
```

Rješavanje problema broja kombinacija možemo riješiti i direktnim pristupom, ispunjavanjem tablice s vrijednostima. Ići ćemo redom kroz tablicu te računati i popunjavati tablicu koristeći već prethodno izračunate vrijednosti.

```
from time import time

n = int(input('Unesi n '))
k = int(input('Unesi k '))
c = [[0 for i in range(n + 1)] for j in range(n + 1)]

start_time = time()
for i in range(n + 1):
  for j in range(i + 1):
    if j == 0 or j == i:
      c[i][j] = 1
    else:
      c[i][j] = c[i - 1][j - 1] + c[i - 1][j]
rezultat = c[n][k]
end_time = time()

print(rezultat)
time_taken = end_time - start_time
print(time_taken)
```

Usporedba vremena izvršavanja ova tri koda dana je u tablici 3.

| $\binom{n}{k}$ | Rekurzivno | S memoizacijom | Nerekurzivno |
|---|---|---|---|
| $\binom{8}{5}$ | 0.0 | 0.0 | 0.0 |
| $\binom{12}{9}$ | 0.0 | 0.0 | 0.0 |
| $\binom{24}{20}$ | 0.0153649 | 0.0 | 0.0 |
| $\binom{26}{11}$ | 1.2233541 | 0.0 | 0.0 |
| $\binom{27}{9}$ | 0.5623958 | 0.0 | 0.0 |
| $\binom{28}{12}$ | 4.5369866 | 0.0 | 0.0 |
| $\binom{30}{14}$ | 25.0157127 | 0.0 | 0.0 |
| $\binom{36}{15}$ | / | 0.0 | 0.0 |

Tablica 3: Usporedba vremena - Binomni koeficijent.

## 2.2 Generiranje kombinatornih objekata

U ovom odjeljku bit će prikazani rekurzivni algoritmi za generiranje kombinatornih objekata, tj. kako pomoću rekurzije generirati sve podskupove danog skupa, permutacije te kombinacije. U prethodnoj cjelini problemi su uvijek imali jednostavno rješenje bez rekurzije, no za generiranje kombinatornih objekata zbog same njihove prirode rekurzivni pristup je bolji te čak jednostavniji.

### 2.2.1 Permutacije

Neka je $S = \{x_1, \ldots, x_n\}$ neki skup s $n$ članova. Permutacija skupa $S$ je uređena $n$-torka međusobno različitih elemenata iz $S$, a definiramo ju kao bijekciju $f : S \to S$. Odabirom uređaja elemenata skupa $S$, elementi $x_1, \ldots, x_n$ prelaze redom u $f(x_1), \ldots, f(x_n)$, dakle u jedan novi niz (uređenu $n$-torku) elemenata iz skupa $S$.

Upotrebom rekurzije ispisat ćemo sve moguće permutacije skupa $S$. Naš skup $S$ bit će implementiran pomoću liste $L = [x_1, x_2, \ldots, x_n]$. Pretpostavimo da imamo algoritam koji ispisuje sve permutacije liste od $n - 1$ elemenata. Fiksiramo prvi element i za svaki taj fiksni prvi element pozovemo algoritam

za ostale elemente liste:

$$\begin{aligned} &x_1|[x_2, x_3, \ldots, x_n] \\ &x_2|[x_1, x_3, \ldots, x_n] \\ &\quad\vdots \\ &x_n|[x_1, x_2, \ldots, x_{n-1}] \end{aligned}$$

Na taj način će algoritam raditi rekurzivno. Zadnji korak bit će kada fiksiramo predzadnji element jer nam tada samo preostaje lista od jednog elementa koje ima samo jednu permutaciju. Ulazni podatak za algoritam je lista $L$, a izlazne vrijednosti su sve permutacije elemenata liste $L$.

```
def permutacije(lista, pozicija):
    if (pozicija == len(lista) - 1):
        print(lista)
    else:
        for i in range(pozicija, len(lista), 1):
            lista[pozicija], lista[i] = lista[i], lista[pozicija]
            permutacije(lista, pozicija + 1)
            lista[pozicija], lista[i] = lista[i], lista[pozicija]
    return

L = eval(input("Unesite listu: "))
permutacije(L, 0)
```

Na primjer, ako korisnik unese listu $L = [1, 2, 3, 4]$, izlaz će biti:

```
[1, 2, 3, 4]
[1, 2, 4, 3]
[1, 3, 2, 4]
[1, 3, 4, 2]
[1, 4, 3, 2]
[1, 4, 2, 3]
[2, 1, 3, 4]
[2, 1, 4, 3]
[2, 3, 1, 4]
[2, 3, 4, 1]
[2, 4, 3, 1]
```

```
[2, 4, 1, 3]
[3, 2, 1, 4]
[3, 2, 4, 1]
[3, 1, 2, 4]
[3, 1, 4, 2]
[3, 4, 1, 2]
[3, 4, 2, 1]
[4, 2, 3, 1]
[4, 2, 1, 3]
[4, 3, 2, 1]
[4, 3, 1, 2]
[4, 1, 3, 2]
[4, 1, 2, 3]
```

### 2.2.2 Kombinacije

Kombinacija je izbor elemenata iz skupa, tako da redoslijed odabira nije bitan. Na primjer, za skup od tri voćke, recimo jabuku, naranču i krušku, postoje tri kombinacije od dva elementa koja se mogu izabrati iz ovog skupa: jabuka i kruška; jabuka i naranča; ili kruška i naranča. Formalnije, $k$-kombinacija skupa $S$ je podskup od $k$ elemenata od $S$. Ako skup ima $n$ elemenata, broj $k$-kombinacija je binomni koeficijent $\binom{n}{k}$.

Koristeći rekurziju generirat ćemo sve moguće $k$-kombinacije $n$-članog skupa $S = \{x_1, \ldots, x_n\}$. Ideja rekurzivnog algoritma temelji se na tome da pretpostavimo da znamo generirati sve $(k-1)$-kombinacije proizvoljnog skupa te nas sada zanima kako pomoću njih generirati sve $k$-kombinacije skupa $S$. Pozovemo rekurziju koja generira $(k-1)$-kombinacije. U svaku kombinaciju koju dobijemo tim pozivom rekurzije dodajmo član $x_1$. Dobili smo sve $k$-kombinacije koje sadrže $x_1$. Kako sad imamo sve $k$-kombinacije koje sadrže $x_1$, $x_1$ nam više ne treba. Preostaje generirati sve kombinacije skupa $S = \{x_2, \ldots, x_n\}$. To ćemo učiniti na isti način. Generiramo $(k-1)$-kombinacije skupa $S = \{x_3, \ldots, x_n\}$, u svaku kombinaciju skupa dodamo $x_2$ i dobijemo sve $k-$kombinacije koje sadrže $x_2$, a ne sadrže $x_1$. Postupak ponavljamo dok ne dođemo do kraja skupa $S$.

```
def kombinacije(lista, k, pozicija, podskup):
    if len(podskup) == k:
        print(podskup)
    else:
        for i in range(pozicija, len(lista)):
            kombinacije(lista, k, i + 1, podskup + [lista[i]])
```

```
    return

L = eval(input("Unesite listu: "))
k = int(input("Unesite k: "))
kombinacije(L, k, 0, [])
```

Na primjer, ako korisnik unese listu $L = [1, 2, 3, 4]$ i broj $k = 3$ ispis će biti:

```
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
```

Drugi pristup generiranju kombinacija temelji se na dokazu Pascalove formule: podijelimo sve $k$-člane podskupove na one koji ne sadrže prvi element skupa i one koji sadrže prvi element skupa, tj. uzimamo jedan po jedan element upisanog skupa koji je implementiran kao polje i ponavljamo dva slučaja. Prvi slučaj je da je element uključen u trenutnu kombinaciju, element stavljamo u pomoćni niz te povećamo broj članova u pomoćnom nizu jer smo upravo dodali jedan element. Drugi slučaj je da element ne želimo u trenutnoj kombinaciji, ne stavljamo ga u pomoćni niz i ne povećavamo broj članova u pomoćnom nizu. U oba slučaja nakon tog koraka promatramo sljedeći element skupa.

```
def kombinacije(lista, k, podskup):
    if len(lista) == k:
        print(podskup + lista)
    elif k == 0:
        print(podskup)
    else:
        kombinacije(lista[1:], k - 1, podskup + [lista[0]])
        kombinacije(lista[1:], k, podskup)
    return

L = eval(input("Unesite listu: "))
k = int(input("Unesite k: "))
kombinacije(L,k,[])
```

Usporedimo vrijeme izvršavanja ova dva koda.

| Testni primjeri: | Rekurzivno | Rekurzivno - Pascal |
|---|---|---|
| $n = 10$, $k = 5$ | 0.0 | 0.0 |
| $n = 12$, $k = 3$ | 0.0 | 0.0 |
| $n = 20$, $k = 5$ | 0.0308731 | 0.0156465 |
| $n = 24$, $k = 11$ | 7.6091408 | 4.8269689 |
| $n = 25$, $k = 12$ | 17.9147012 | 10.1257436 |
| $n = 27$, $k = 13$ | 72.4826410 | 26.2356181 |
| $n = 28$, $k = 14$ | 111.0931282 | 51.3513007 |
| $n = 30$, $k = 12$ | 135.2574759 | 110.4784949 |

Tablica 4: Usporedba vremena - Kombinacije.

Za primjere na kojima smo testirali algoritme, rekurzija koja se temelji na Pascalovoj formuli je učinkovitija.

### 2.2.3 Particije

Particija $n$-članog skupa je familija nepraznih disjunktnih podskupova koji u uniji daju cijeli skup. Pod $k$-particijom podrazumijevamo particiju kojoj je broj podskupova jednak $k$. Upotrebom rekurzije generirat ćemo sve particije skupa $S = \{x_1, \ldots, x_n\}$. Ideja rekurzivnog algoritma temelji se na tome da pretpostavimo da znamo generirati sve particije $(n-1)$-članog skupa te nas sada zanima kako iz njih generirati sve particije $n$-članog skupa. Uzmimo proizvoljnu particiju skupa $S \setminus \{x_n\}$ i označimo ju s $P$. U elemente skupa $P$, koji su podskupovi skupa $S \setminus \{x_n\}$, dodamo element $x_n$ i na taj način generiramo particije skupa $S$. Dodatno, generiramo još jednu particiju tako da dodamo skup $\{x_n\}$ u skup $P$. Ako taj postupak napravimo za svaku particiju skupa $S \setminus \{x_n\}$ generirat ćemo sve particije skupa $S$.

```
def particije(part, skup):
    if skup==[]:
        print(part)
    else:
        for i in range(len(part)):
            part[i].append(skup[0])
            particije(part, skup[1:])
            part[i].pop(-1)
        part.append([skup[0]])
        particije(part, skup[1:])
        part.pop(-1)
    return
```

```
L = eval(input("Unesite listu: "))
particije([], L)
```

Na primjer, ako korisnik unese listu $L = [1, 2, 3, 4]$, izlaz će biti:

```
[[1, 2, 3, 4]]
[[1, 2, 3], [4]]
[[1, 2, 4], [3]]
[[1, 2], [3, 4]]
[[1, 2], [3], [4]]
[[1, 3, 4], [2]]
[[1, 3], [2, 4]]
[[1, 3], [2], [4]]
[[1, 4], [2, 3]]
[[1], [2, 3, 4]]
[[1], [2, 3], [4]]
[[1, 4], [2], [3]]
[[1], [2, 4], [3]]
[[1], [2], [3, 4]]
[[1], [2], [3], [4]]
```

## 2.3 Crtanje fraktalnih skupova

Fraktal je geometrijski lik koji se sastoji od dijelova, a svaki dio sliči na cijeli fraktal. Za fraktale je karakteristično da se isti oblik stalno ponavlja, a konačna slika izgleda kao uvećan osnovni oblik od kojih se fraktal sastoji.

Fraktale ćemo crtati u programskom jeziku Python služeći se kornjačinom grafikom koju moramo uključiti na početku programa. Kako je za fraktale karakteristično da se isti oblik stalno ponavlja, a mijenja se samo njegova veličina, koristit ćemo rekurziju. Fraktal možemo crtati beskonačno, no da bismo mogli osmisliti algoritam moramo imati zaustavni trenutak, odnosno moramo definirati koliko detaljan fraktal crtamo. Korisnik će unijeti broj $n$ koji će predstavljati dubinu rekurzije, odnosno koliko detaljno želi nacrtati fraktal te duljinu osnovne stranice, odnosno duljinu segmenta nad kojim će se $n$ puta pozvati rekurzivna funkcija za crtanje određenog uzorka, ali svaki put smanjenog po nekom pravilu.

### 2.3.1 Kochova krivulja

Konstrukcija započinje crtanjem segmenta koji je podijeljen na tri jednaka dijela. Srednji dio segmenta zamijenimo s dvije stranice jednakostraničnog trokuta iste duljine. Dobijemo četiri sukladne dužine čija je duljina $\frac{1}{3}$ početne duljine segmenta. Postupak se nastavlja dalje na isti način. Svaku od četiri dužine podijelimo na tri jednaka dijela te na svakom dijelu ponovimo postupak, odnosno srednji dio zamijenimo stranicama jednakostraničnog trokuta bez baze. Uočavamo da se svaki put kada treba ponoviti postupak može pozvati funkcija. Kod je napisan tako da kornjača kada dođe do $\frac{1}{3}$ duljine segmenta odmah kreće u konstrukciju trokuta bez baze. Na tom dijelu poziva se rekurzija te kornjača na jednom dijelu crta potpunu sliku te tek kada završi osnovni slučaj nastavlja dalje na drugi dio segmenta.

Ulaz je varijabla $n$ koja označava dubinu rekurzije na kojoj primjenjujemo osnovni slučaj, varijabla $a$ koja označava širinu krivulje, a izlaz je slika Kochove krivulje.

```
from turtle import *

def krivulja(n,a):
    if n == 0:
        fd(a)
    else:
        krivulja(n-1,a/3); lt(60)
        krivulja(n-1,a/3); rt(120)
        krivulja(n-1,a/3); lt(60)
        krivulja(n-1,a/3)

a = float(input('Veličina '))
n = int(input('Dubina '))
krivulja(n,a)
```

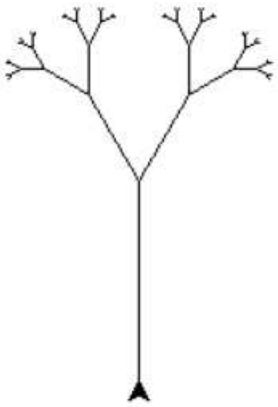
Slika 1: Kochova krivulja za $n = 1$, $n = 2$ i $n = 3$.

### 2.3.2 Binarno stablo

Fraktal binarno stablo je geometrijski lik koji se sastoji od linije koja predstavlja deblo te lijeve i desne grane. Svaka grana dalje sadrži svoju lijevu i desnu manju granu. Duljina svake sljedeće grane duplo je manja od prethodne.

Crtanje binarnog stabla započnemo tako da crtamo deblo, nakon toga crtamo lijevu granu stabla koja je u odnosu na deblo pod kutom od 30°, ali duljinu grane skratimo na pola, nakon toga crtamo desnu granu stabla pod kutom od 30° u odnosu na deblo, ali opet duljinu grane skratimo na pola. Nakon toga se vraćamo u korijen stabla. Stablo se crta dok je veličina grane veća od 1, vrijednosti manje od toga ne možemo uočiti na ekranu te je to zaustavna točka.

```
from turtle import*

def stablo (a):
    if a > 1:
        fd(a)
        lt(30)
        stablo(a // 2)
        rt(60)
        stablo( a // 2)
        lt(30)
        pu()
        bk(a)
        pd()
lt(90)
pu()
bk(200)
pd()
n = int(input('Veličina '))
stablo(n)
```



Slika 2: Binarno stablo.

U funkciji *stablo(n)* dva puta smo pozvali istu funkciju *stablo(n)*, ali s manjom ulaznom vrijednosti. Najprije se crta lijeva polovica stabla, pa zatim desna. Stablo koje smo nacrtali je simetrično. Dvije susjedne grane iste duljine uvijek su pod kutom od 60 stupnjeva. Svaka grana crta se na isti način kao čitavo stablo. Na kraju svakog poziva funkcije, kornjača se vraća u „korijen“ onoga što je tom funkcijom nacrtano.

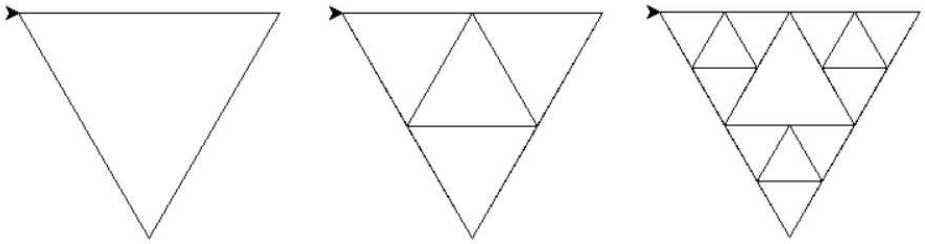
### 2.3.3 Trokut Sierpińskog

Trokut Sierpińskog je jednakostranični trokut koji se sastoji od više manjih jednakostraničnih trokuta nastalih spajanjem polovišta stranica trokuta.

Konstrukciju trokuta Sierpińskog započinjemo crtanjem jednakostraničnog trokuta. Nakon toga pronađemo polovišta svake stranice te ih spojimo. Spajanjem polovišta dobijemo četiri sukladna trokuta. Postupak možemo dalje ponavljati na tri rubna trokuta koliko puta želimo. Vidimo da nakon prvog spajanja polovišta dobijemo tri sukladna trokuta čije su stranice upola manje od početne.

```
from turtle import *

def sierpinski(n,a):
    if n == 0:
        return

    for i in range(3):
        sierpinski(n-1, a/2)
```

```
        fd(a); rt(120);

    return

a = int(input('Veličina '))
n = int(input('Dubina '))
sierpinski(n, a)
```



Slika 3: Trokut Siepińskog za $n = 1$, $n = 2$ i $n = 3$.

## 2.4 Sortiranje

### 2.4.1 Mergesort

Mergesort je algoritam za sortiranje niza. Princip njegovog rada je uzastopno dijeljenje niza na pola sve dok se ne može dalje dijeliti, sortiranje svake nastale polovice te spajanje sortiranih podnizova u sortirani niz početne veličine. Mergesort je rekurzivni algoritam jer proces podjele niza na pola te sortiranje novo nastalih podnizova ponavlja toliko dugo dok podniz koji nastaje podjelom ima samo jedan element, a niz s jednim elementom smatramo sortiranim. Nakon rekurzivnog poziva slijedi spajanje nizova. Spajanje se odvija prolaskom kroz dva podniza paralelno te uspoređivanjem podnizova, element po element. Manji od dva trenutna elementa stavljamo u sortirani niz i pomaknemo se na sljedeći element podniza u kojem se nalazio manji element.

Koraci mergesorta su: postavljanje lijevog indeksa, postavljanje desnog indeksa, provjera je li lijevi indeks niza manji od desnog indeksa, ako da, pronalazak središnjeg član niza da bi se niz mogao podijeliti na dva niza. Nakon što se niz podijelio na dvije polovice, poziva se ponovno mergesort koji radi na prvoj polovici, nakon toga poziva se mergesort koji radi na drugoj polovici. Na kraju se lijeva i desna polovica spajaju u sortirani niz.

Spajanje nizova u konačni sortirani niz ostvaruje se kroz nekoliko prolaza po svim elementima. Prvi prolaz spaja nizove veličine jedan u nizove veličine dva, drugi prolaz spaja nizove veličine dva, a $i$-ti prolaz spaja nizove veličine $2^{i-1}$. Niz veličine $n$ dijelili smo na dva dijela toliko dugo dok ne dođemo do niza duljine 1, a broj takvih dijeljenja je $\log_2 n$. Zato je ukupan broj prolaza spajanja $\log_2 n$. Spajanje dva sortirana niza odvija se u linearnom vremenu, odnosno za spajanje treba $O(n)$ vremena. Ukupno vrijeme izvršavanja je $O(n \log_2 n)$.

```
def mergesort(niz):
    if len(niz) > 1:

        srednji = len(niz)//2
        L = niz[:srednji]
        R = niz[srednji:]
        mergesort(L)
        mergesort(R)
        i = 0
        j = 0
        k = 0

        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                niz[k] = L[i]
                i = i + 1
            else:
                niz[k] = R[j]
                j = j + 1
            k = k + 1

        while i < len(L):
            niz[k] = L[i]
            i = i + 1
            k = k + 1

        while j < len(R):
            niz[k] = R[j]
            j = j + 1
            k = k + 1
    return
```

```
niz = eval(input("Unesite niz: "))
mergesort(niz)
print(niz)
```

Promatrajmo niz [5,6,1,4,3,9,7]. Niz ima 7 elemenata te će se podijeliti na dva niza od kojih jedan ima 4 elementa, a drugi 3 elementa: [5,6,1,4] i [3,9,7]. Sada ponovno dijelimo svaki nastali niz na dva. Nizovi koji su nastali su: [5,6], [1,4], [3,9] i [7]. Nastali nizovi i dalje se dijele na pola sve dok daljnje dijeljenje nije moguće. Nizovi na kraju podijele su: [5],[6],[1],[4],[3],[9] i [7]. Nakon što smo podijelili niz na najmanje dijelove, spajat ćemo nizove u veći na temelju usporedbe veličina elemenata. Prolazimo kroz svaki niz paralelno te u novi niz stavljamo manji element. Time se postiže sortiranost. Nakon prvog spajanja imamo [5,6], [1,4], [3,9] i [7]. Nakon sljedećeg spajanja imamo: [1,4,5,6], [3,7,9] te konačno [1,3,4,5,6,7,9]. Rezultat mergesorta je [1,3,4,5,6,7,9].

### 2.4.2 Quicksort

Quicksort je još jedan rekurzivni algoritam za sortiranje niza. Ideja quicksorta je odabrati jedan element niza kojem nazivamo pivotni element ili stožerni element. Pivotni element može biti prvi, zadnji, srednji element niza ili jednostavno slučajno odabrani element niza. Cilj je pivotni element staviti na njegovu ispravnu poziciju u sortiranom nizu, a to postižemo tako da krenemo od krajnjeg lijevog elementa niza te se pomičemo u desno toliko dugo dok ne nađemo element koji je veći od pivotnog, također to radimo i s krajnje desne strane te se pomičemo u lijevo tako dugo dok ne nađemo element koji je manji od pivotnog. Kad smo pronašli element lijevo veći od pivotnog i element desno manji od pivotnog, zamijenimo ta dva elementa. Time će se postići da veći elementi dolaze iza pivotnog, a manji ispred. Zatim nastavljamo dalje, tražeći sljedeća dva broja koja treba zamijeniti. Postupak ide toliko dugo dok lijevi indeks ne postane veći od desnog. Kada lijevi indeks postane veći od desnog sljedeći korak je zamijeniti pivotni element s elementom koji je nakon preklapanja poprimio indeks desno, to je ujedno i najveći broj koji je manji od pivotnog. Njihovom zamjenom pivotni element dolazi na svoje mjesto. Osim što je pivotni element na svojoj ispravnoj poziciji imamo dio niza u kojem su svi elementi manji od pivotnog elementa te drugi dio niza u kojem se nalaze svi elementi koji su veći od pivotnog. Korak nakon toga je pozivanje rekurzivne funkcije quicksort na dio niza ispred pivotnog i na dio niza iza pivotnog. Ne dijelimo niz nego promijenimo vrijednosti krajnjeg lijevog i krajnjeg desnog indeksa koje ćemo proslijediti rekurzivnoj funkciji quicksort. Problem se rastavlja na manji uzastopnom

primjenom quicksorta, odnosno postupak ide toliko dugo dok niz nad kojim pozivamo rekurzivnu funkciju nije prazan ili duljine jedan. Prazan niz i niz duljine jedan smatramo sortiranim.

```
def quicksort(niz, lijevo, desno):
    if (lijevo < desno):

        pivot = lijevo
        i = lijevo
        j = desno

        while (i < j):
            while niz[i] <= niz[pivot] and i < desno:
                i = i + 1
            while niz[j] > niz[pivot]:
                j = j - 1

            if (i < j):
                niz[i], niz[j] = niz[j], niz[i]

        niz[pivot], niz[j] = niz[j], niz[pivot]
        quicksort(niz, lijevo, j - 1)
        quicksort(niz, j + 1, desno)


niz = eval(input("Unesite niz: "))
quicksort(niz, 0, len(niz) - 1)
print(niz)
```

Promatrajmo niz [5,6,1,4,3,9,7]. Za pivotni element uzimamo element prvi s lijeva, a to je 5. Element s lijeva nakon pivotnog, a s kojim krećemo je 6, a zdesna 7. Krećemo se po nizu s lijeva dok ne dođemo do elementa većeg od pivotnog i tu stanemo, a to je element 6. Također se krećemo zdesna po nizu dok ne dođemo do elementa manjeg od pivotnog, a to je 3. Mijenjamo 3 i 6 jer nisu u pravom poretku. Niz je sada: [5,3,1,4,6,9,7]. Dalje prolazimo po nizu, 1 i 4 su manji od 5 te indeks lijevo stane tek na elementu 6, a indeks desno stane na elementu 4. Indeksi su se preklopili, pivotni element se mijenja s elementom čiji je indeks desno, a to je 4. Niz sada izgleda: [4,3,1,5,6,9,7]. Uočimo da imamo dio niza u kojem su svi elementi manji od pivotnog elementa 5 te drugi dio niza u kojem se nalaze svi elementi koji

su veći od pivotnog. Sada pozivamo rekurzivnu funkciju quicksort na dijelu niza ispred pivotnog i na dijelu niza iza pivotnog. To postižemo tako da promijenimo vrijednosti krajnjeg lijevog i krajnjeg desnog indeksa te onda te vrijednosti proslijedimo rekurzivnoj funkciji quicksort. Nizovi nad kojima se ponavlja postupak su [4,3,1] i [6,9,7]. U prvom nizu pivotni element je 4. Ponavlja se postupak uspoređivanja brojeva s pivotnim elementom, indeks lijevo stane na 1 te i indeks desno stane na elementu 1 jer je to element manji od pivotnog. Indeksi su se preklopili, pivotni element se mijenja s elementom čiji je indeks desno, a to je 1. Niz sada izgleda ovako: [1,3,4]. Nakon toga se rekurzivna funkcija poziva na dijelovima [1,3] i []. U nizu [1,3] neće biti zamjene jer je poredak pravilan. Pozvat će se rekurzivna funkcija na [] i [3]. Postupak staje jer dolazimo do praznog niza i niza duljine 1, a oni su sortirani. U nizu [6,9,7] pivotni element je 6. Indeks lijevo odmah stane na broju 9, a indeks desno stane na broju 6 jer je to prvi element koji je nije veći od pivotnog. Pivotni se element mijenja sam sa sobom. Niz je [6,9,7]. Nakon toga se rekurzivna funkcija poziva na dijelovima [] i [9,7]. U nizu [9,7] dolazi do zamjene te je niz onda [7,9]. Dalje se poziva rekurzivna funkcija na [] i [7]. Postupak staje jer dolazimo do praznog niza i niza duljine 1, a oni su sortirani. Rezultat je sortirani niz koji izgleda ovako: [1,3,4,5,6,7,9].

Vrijeme potrebno quicksortu općenito se može zapisati na sljedeći način:

$$T(n) = T(i) + T(n - i - 1) + O(n),$$

gdje je $T(n)$ vremenska složenost quicksorta za $n$ cijelih brojeva. Analizirajmo to raščlanjivanjem vremenske složenosti svakog procesa. Prva dva izraza odnose se na dva rekurzivna poziva, a posljednji izraz je za proces podjele niza. Broj elemenata koji su manji od pivotnog označen je s $i$. Ovisno o ulaznom nizu te strategiji podjele kod quicksorta razlikujemo tri slučaja: najgori, prosječni i najbolji slučaj. Najgori slučaj događa se kada proces podjele uvijek odabire najveći ili najmanji element kao pivotni. U ovom slučaju, proces podjele bio bi vrlo neuravnotežen, tj. jedan podniz s $n - 1$ elemenata, a drugi s 0 elemenata. Ova situacija se događa kada je niz poredan u rastućem ili padajućem redoslijedu. Uvrstimo te vrijednosti u gornju formulu za $T(n)$. Neka je $i = n - 1$.

$$T(n) = T(n - 1) + T(0) + cn$$

$$T(n) = T(n - 1) + cn$$

Uvrstimo sve vrijednosti od $i = n - 1$ do 1.

$$\begin{aligned}
T(n) &= T(n-1) + cn \\
&= T(n-2) + c(n-1) + cn \\
&= T(n-3) + c(n-2) + c(n-1) + cn \\
&\vdots \\
&= T(1) + 2c + 3c + \cdots + c(n-3) + c(n-2) + c(n-1) + cn \\
&= c + 2c + 3c + \cdots + c(n-3) + c(n-2) + c(n-1) + cn \\
&= c(1 + 2 + 3 + \cdots + n - 3 + n - 2 + n - 1 + n)
\end{aligned}$$

Ovo je suma aritmetičkog niza te je $T(n) = c(n(n+1)/2) = O(n^2)$. Vremenska složenost quicksorta u najgorem slučaju je $O(n^2)$ [7]. Najbolji slučaj se događa kada proces podjele uvijek odabire srednji element kao pivotni. Takvu podjelu nazivamo uravnotežena podjela jer rekurzivnu funkciju quicksort pozivamo na nizovima jednake veličine, odnosno veličine $n/2$.

$$T(n) = 2T(n/2) + \theta(n).$$

Ovo ponavljanje je slično mergesortu za koje je rješenje $O(n \log n)$. Prosječni slučaj quicksorta ovisi o redoslijedu vrijednosti u unosu. Možemo pretpostaviti da su sve permutacije ulaza jednako vjerojatne. Kada pokrenemo algoritam na nasumičnim nizovima, malo je vjerojatno da će se podjela dogoditi na isti način na svakoj razini rekurzije. Stoga očekujemo da će neke od podjela biti dobro uravnotežene, a neke od podjela će biti neuravnotežene. U prosječnom slučaju imat ćemo kombinaciju dobrih (uravnoteženih) i loših (neuravnoteženih) podjela koje će biti nasumično raspoređene. Možemo pretpostaviti da dobra i loša podjela alterniraju. Pretpostavimo da na početku imamo dobru podjelu, a na sljedećoj razini poziva rekurzije imamo lošu podjelu. Vrijeme procesa podjele bit će $O(n)$ na obje razine. Dakle, vrijeme loše podjele nakon koje slijedi dobra podjela je $O(n)$. Ova situacija je ekvivalentna jednoj razini podjele koja izgleda slično slučaju uravnotežene podjele. Dakle, prosječno vrijeme izvođenja quicksorta puno je bliže najboljem slučaju. Prosječna vremenska složenost je $O(n \log n)$.

"Veliko O" i theta $\theta$ dvije su oznake koje se koriste u računalnoj znanosti za opisivanje gornje granice i izvedbe prosječnog slučaja algoritama. Zapis $O$ gornja je granica stope rasta vremena izvođenja algoritma, što znači da daje procjenu maksimalnog vremena koje će algoritmu biti potrebno za izvođenje. Ideja iza oznake $O$ je opisati kako vrijeme rada algoritma raste s povećanjem veličine ulaza. To nam daje ideju o najgorem slučaju vremenske složenosti algoritma. Theta notacija, s druge strane, daje analizu prosječnog slučaja vremena izvođenja algoritma za sve moguće ulaze zadane veličine.

Prosječna vremenska složenost algoritma mergesort i quicksort je $\theta(n \log n)$. Kako možemo usporediti dva $\theta(n \log n)$ algoritma da bismo odlučili koji je brži? Kada se suočimo s algoritmima iste vremenske složenosti, detalji implementacije i nedostaci sustava poput performansi predmemorije i veličine memorije mogu biti odlučujući faktor. Eksperimenti pokazuju da je dobro implementiran quicksort obično dva do tri puta brži od mergesorta [15].

### 2.4.3 Usporedba vremena izvršavanja

U ovoj cjelini uspoređujemo vremena izvršavanja tri algoritma za sortiranje: mergesort, quicksort i ugrađene funkcije sort u programskom jeziku Python. Slijedi dio koda koji se odnosi na mjerenje vremena koje je potrebno pojedinom algoritmu za sortiranje niza određene duljine:

```
from time import time
import random
import matplotlib.pyplot as plt
import matplotlib
import numpy as np

def crtaj(merge_lista):
    x1 = [100,200,300,400,500,600,700,800,900,1000,2000,3000,
          4000,5000]
    plt.scatter(x1, merge_lista, label = "Mergesort")

    a1 = merge_lista[13]/5000
    a2 = merge_lista[13]/(5000 * np.log(5000))

    x = np.array(range(1,5000))
    y = a1 * x
    plt.plot(x, y, markersize=1, marker='.', color='red')

    x = np.array(range(1,5000))
    y = a2 * x * np.log(x)
    plt.plot(x, y, markersize=1, marker='.', color='blue')

    plt.yscale('log')

    plt.gca().yaxis.set_major_locator(plt.LogLocator(
        base=10.0, subs=(1, 2, 5)))
    plt.gca().yaxis.set_major_formatter(plt.FormatStrFormatter(
```

```
        '%.1f'))
    plt.ylim(bottom=0.1)
    plt.ylim(top=merge_lista[13]+0.5)


    ax = plt.gca()
    ax.xaxis.set_minor_locator(plt.MultipleLocator(200))

    plt.show()


merge_lista = []
lista_n=[100000,200000,300000,400000,500000,600000,700000,
         800000,900000,1000000,2000000,3000000,4000000,5000000]

for i in lista_n:
    print("\n")
    for j in range(10):
        s1 = 0
        s2 = 0
        s3 = 0
        n1 = list(range(i))
        random.shuffle(n1)
        n2 = n1.copy()
        n3 = n1.copy()

        start_time1 = time()
        mergesort(n1)
        end_time1 = time()
        time_taken1 = end_time1 - start_time1
        s1 = s1 + time_taken1

        start_time2 = time()
        quicksort(n2, 0, len(n2) - 1)
        end_time2 = time()
        time_taken2 = end_time2 - start_time2
        s2 = s2 + time_taken2


        start_time3 = time()
        n3.sort()
```

| Duljina | Mergesort | Quicksort | Ugrađena funkcija |
|---|---|---|---|
| 100 000 | 0.065158557891846 | 0.037633609771729 | 0.002001404762268 |
| 200 000 | 0.143125605583191 | 0.080759787559509 | 0.004824709892273 |
| 300 000 | 0.225535988807678 | 0.125772070884705 | 0.008098983764648 |
| 400 000 | 0.310421705245972 | 0.180858898162842 | 0.011372089385986 |
| 500 000 | 0.397456216812134 | 0.225444006919861 | 0.014647102355957 |
| 600 000 | 0.495147109031677 | 0.283408617973328 | 0.017889881134033 |
| 700 000 | 0.581924700737000 | 0.325863385200501 | 0.022578787803650 |
| 800 000 | 0.673564887046814 | 0.377324104309082 | 0.025591778755188 |
| 900 000 | 0.769825983047485 | 0.438009715080261 | 0.030015397071838 |
| 1 000 000 | 0.862856888771057 | 0.486750793457031 | 0.034714603424072 |
| 2 000 000 | 1.885594272613525 | 1.133363199234009 | 0.082199716567993 |
| 3 000 000 | 3.218360877037048 | 1.893784976005554 | 0.134744000434876 |
| 4 000 000 | 4.304028224945069 | 2.539829397201538 | 0.193134570121765 |
| 5 000 000 | 5.459983706474304 | 3.271336913108826 | 0.254327082633972 |

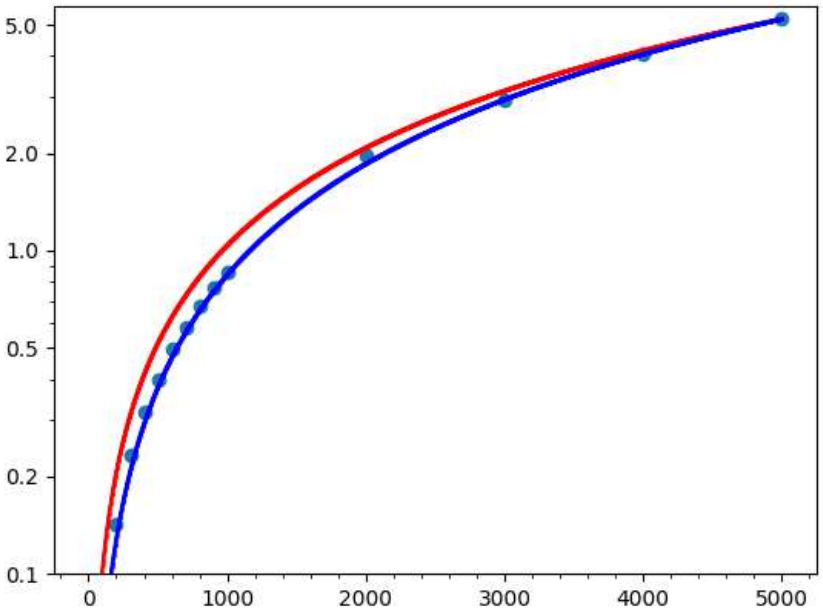
Tablica 5: Usporedba prosječnih vremena izvođenja.

```
        end_time3 = time()
        time_taken3 = end_time3 - start_time3
        s3 = s3 + time_taken3


    p1 = s1/10
    p2 = s2/10
    p3 = s3/10

    merge_lista.append(p1)
    print(i," ",p1," ",p2," ",p3)

crtaj(merge_lista)
```

U listi su pohranjene vrijednosti koje predstavljaju duljinu nizu kojeg želimo sortirati te uz pomoć naredbe *random.shuffle* generiramo niz nad kojim se pozivaju sve tri funkcije za sortiranje te se mjeri vrijeme. Postupak se ponavlja 10 puta na raznim nizovima, ali jednake duljine da bismo dobili prosječno vrijeme. Duljine niza te prosječna vremena izvršavanja za sva tri algoritma sortiranja dana su u tablici 5.

Za nizove velike duljine, ugrađena funkcija je daleko brža. U testiranju se pokazala više od 21 puta brža od mergesorta te više od 12 puta brža od quicksorta za primjer kada je duljina niza 5 000 000.

Graf daje usporedbu vremena mergesort algoritma s dvije funkcije $f(x) = a_1 \cdot x$, označena crvenom bojom i $g(x) = a_2 \cdot x \log(x)$, označena plavom bojom, gdje je $a_1$ konstanta takva da vrijedi: $a_1 = mergesort(5000)/5000$, a $a_2$ konstanta takva da vrijedi: $a_2 = mergesort(5000)/(5000 \cdot \log(5000))$. Oznaka $mergesort(5000)$ odnosi se na prosječno vrijeme izvršavanja sortiranja niza mergesort algoritmom, a broj 5000 odnosi se na skaliranu duljinu niza prikladnu za crtanje.




Slika 4: Usporedba mergsorta s $f(x)$ i $g(x)$.

# 3 Rekurzija u školi

## 3.1 Rekurzija u kurikulumu

Ciljevi predmeta Informatike u kurikulumu [12] razvrstani su u četiri domene:

- e-društvo,
- digitalna pismenost i komunikacija,
- računalno razmišljanje i programiranje,
- informacije i digitalna tehnologija.

Rekurzivni način razmišljanja i programiranje korištenjem rekurzija pripada domeni Računalno razmišljanje i programiranje. Koncept rekurzije prvi put se spominje u 8. razredu, ishod B.8.3: *prepoznaje i opisuje mogućnost primjene rekurzivnih postupaka pri rješavanju odabranih problema te istražuje daljnje mogućnosti primjene rekurzije.* Dalje u 3. razredu prirodoslovno-matematičke gimnazije sa 105 sati godišnje ishod B.3.3 glasi: *nakon treće godine učenja predmeta Informatika u domeni Računalno razmišljanje i programiranje učenik rješava problem primjenjujući rekurzivnu funkciju.* Te u 3. razredu prirodoslovno-matematičke gimnazije i 4. razredu opće gimnazije obje sa 70 sati informatike godišnje ishod je jednak B.3.3/B.4.1 te glasi: *nakon treće/četvrte godine učenja predmeta Informatika u srednjoj školi u domeni Računalno razmišljanje i programiranje učenik rješava problem primjenjujući rekurzivnu funkciju.*

Kurikulum daje preporuke za ostvarivanje odgojno-obrazovnih ishoda. Za 8. razred preporučuje da se rekurzija uvodi na grafičkim primjerima (npr. trokut Sierpinskog, Kochova pahuljica) te da se diskutira o njihovim obilježjima. U radu se treba služiti konkretnim modelima kao što su matrjoške (ruske lutke), tornjevi Hanoja, primjeri iz stvarnoga života (npr. dijeljenje stanica).

Za 3. razred srednje škole sa 105 sati godišnje te za 3. i 4. razred srednje škole sa 70 sati godišnje preporučuje da učenici sami pronađu primjere vizualnih rekurzija poput zrcala koja se ogledaju jedno u drugome, da odrede rekurzivnu relaciju na jednostavnim problemima kao što je zbroj prvih $n$ članova reda $1–2 + 3–4 + \ldots$, te da analiziraju jednostavne primjere poput Fibonaccijevih brojeva. Učenicima treba skrenuti pozornost da u nekim problemima kao što je taj s Fibonaccijevim brojevima, rekurzivni postupak nije učinkovit. Dalje je preporuka da se primjenjuje kornjačina grafika za crtanje rekurzivnih crteža.

## 3.2 Obrada rekurzije u udžbenicima

### 3.2.1 Osmi razred

Na početku prolazimo kroz aktivnosti koje su zajedničke udžbenicima za osmi razred osnovne škole, a nakon toga ćemo za svaki udžbenik istaknuti nekoliko posebnosti. Rekurzija se obrađuje kao nastavna jedinica unutar cjeline Programiranje u Pythonu. Aktivnosti koje se provode u raznim udžbenicima za osmi razred su:

- ispis prvih $n$ prirodnih brojeva,

```
def ispis(n):
    if n==1:
        print(1, end=' ')
    else:
        ispis(n-1)
        print(n, end=' ')

n = int(input())
ispis(n)
```

- zbroj prvih $n$ prirodnih brojeva,

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

n = int(input())
z = zbroj(n)
print(z)
```

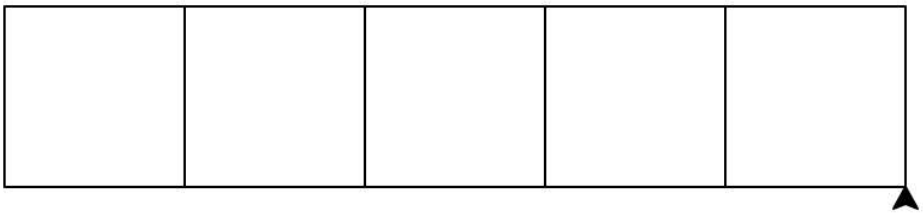
- umnožak prvih $n$ prirodnih brojeva,

```
def umnozak(n):
    if n ==1:
        return 1
    else:
        return n * umnozak(n - 1)
```

```
n = int(input())
u = umnozak(n)
print(u)
```

- crtanje niza kvadrata pomoću rekurzije u kornjačinoj grafici.

```
from turtle import *
def kvadrat(n, a):
    if n > 0:
        for k in range(4):
            fd(a); rt(90)
        rt(90); fd(a); lt(90)
        kvadrat(n - 1, a)


lt(90)
a = textinput('Duljina stranice','a = ')
a = int(a)
n = textinput('Broj kvadrata','n = ')
n = int(n)
kvadrat(n, a)
```

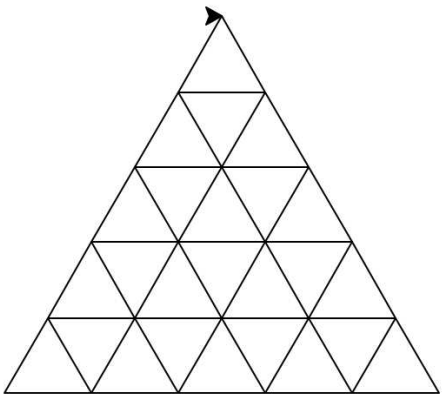


Slika 5: Kvadrati za $a = 75$ i $n = 5$.

Iz udžbenika [3] izdvajamo dvije zanimljive aktivnosti koje se ne spominju u ostalim udžbenicima. Prva je crtanje piramide od $n$ jednakostraničnih trokuta duljine stranice $a$ primjenom rekurzivne funkcije.

```
from turtle import*
def trokut(a):
```

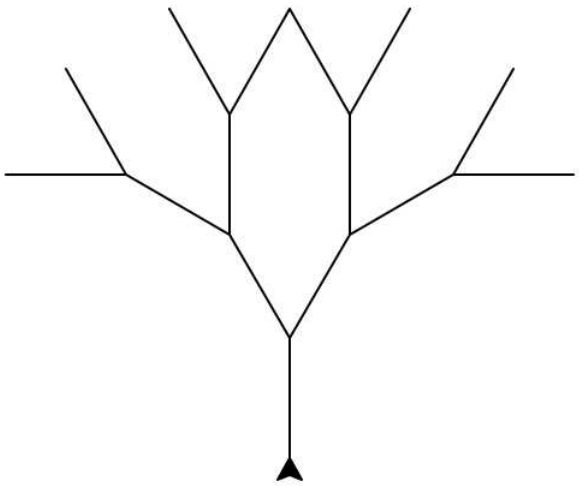
```
    for i in range (3):
        fd(a)
        lt(120)
def niz(a, n):
    for i in range (n):
        trokut(a)
        fd(a)
def postavi(a, n):
    bk(n * a)
    lt(60); fd(a); rt(60)

def piramida(a, n):
    if n < 1:
        return
    else:
        niz(a, n)
        postavi(a, n)
        piramida(a, n - 1)

n = int(input())
a = int(input())
piramida(a, n)
```



Slika 6: Piramida za $a = 50$ i $n = 5$.

Druga aktivnost iz udžbenika [3] je crtanje simetričnog stablo u kojem svaka grana ima točno dvije podgrane u kornjačinoj grafici.

```
from turtle import*
lt(90)
def stablo(a, n, kut):
    if n < 1:
        return
    else:
        fd(a)
        lt(kut)
        stablo(a, n - 1, kut)
        rt(2 * kut)
        stablo(a, n - 1, kut)
        lt(kut)
        bk(a)

a = int(input())
n = int(input())
kut = int(input())
stablo(a, n, kut)
```



Slika 7: Stablo za $a = 50$, $n = 4$ i kut=30.

Rekurzija se u udžbeniku [14] pojavljuje unutra tri poglavlja. Prvi puta se pojavljuje unutar poglavlja Računalno razmišljanje gdje se općenito objašnjava

koncept rekurzije. Drugi put se pojavljuje unutar poglavlja Scratch, a treći puta unutar poglavlja Python. Zanimljiv je pristup koji je dan u poglavlju Računalno razmišljanje. Računalno razmišljanje odnosno, *computational thinking* puno je širi i dublji pojam od samog programiranja. Prije nego što krenemo s rješavanjem problema, potrebno je razumjeti sam problem i načine na koje se on može riješiti. Računalno razmišljanje nam daje uputu kako pristupiti rješavanju bilo kojeg problema i kako razviti moguća rješenja i to na način prikladan za implementaciju na računalu. Računalno razmišljanje provodi se kroz četiri faze: razumijevanje problema, stvaranje plana rješavanja problema, izvršavanje osmišljenog plana i osvrt na rješenje i metodu rješavanja. Tek nakon provedbe tih koraka koristimo računalo kao pomoć u rješavanju problema.

Navodimo zadatak iz udžbenika [14] nad kojim je provedeno računalno razmišljanje. Nora je voditeljica plesne skupine i nakon održane audicije mora izabrati $n$ plesača/plesačica koji će sudjelovati na natjecanju. Napiši algoritam za rješavanje tog problema.

U koraku stvaranja plana rješavanja problema udžbenik navodi: u početku treba odabrati $n$ plesača. Nakon što odaberemo prvog plesača i dalje imamo problem - odabrati plesače, ali sada ih ne treba više odabrati $n$ nego $n-1$. Nakon toga Nora ima isti problem - odabrati plesače, ali sada treba odabrati još jednog plesača manje te je sve bliže cilju. Problem je riješen kad odabere svih $n$ plesača.

Iako su u danom zadatku u udžbeniku provedeni svi koraci te je dano rješenje, zadatak ipak nije najbolji primjer. Odabirom jednog plesača problem se smanjuje na manji, ali se ne objašnjava po kojem kriteriju biramo tog jednog plesača. U ovom primjeru autori su kao glavni cilj imali naglasak na rekurzivni pristup, odnosno da problem treba smanjiti dok ne dođemo do osnovnog problema kojeg znamo riješiti.

Pokazat ćemo kako bi primjena računalnog razmišljanja izgledala na problemu Hanojskih tornjeva. Problem Hanojskih tornjeva pretpostavlja da postoje tri štapa, označeni s A, B, odnosno C i niz diskova različitih veličina koji se mogu staviti na bilo koji štap. Diskovi su složeni jedan na drugi u rastućem poretku tako da je najmanji na vrhu, a najveći na dnu. Cilj je premjestiti sve diskove s prvog štapa, štapa A na treći štap, štap C. U tome procesu treba poštovati sljedeća pravila: istovremeno se može premjestiti samo jedan disk, svaki potez sastoji se od uzimanja diska s vrha jedne hrpe i stavljanja na drugu hrpu ili na prazan štap te se veći disk ne smije stavljati na manji disk.

- Razumijevanje problema: iako u početku ne znamo riješiti problem s $n$ diskova, da bismo razumjeli problem i mogli ga onda riješiti krenut

ćemo s osnovnim slučajem. Ako imamo samo jedan disk, rješenje postaje očito, premjestit ćemo taj disk sa štapa A na štap C. Ako imamo dva diska, prvo ćemo premjestiti prvi disk s A na B, onda ćemo donji veći disk premjestiti s A na C te onda disk s B premjestiti na disk C. Već polako uočavamo da je najbitnije najveći disk premjestiti na odredišni da bi onda na njega mogli slagati manje diskove. Ako imamo tri diska to će izgledati tako da prvo u tri poteza pomaknemo prva dva diska na štap B onda treći, najveći disk premjestimo na štap C te onda diskove sa štapa B opet u tri koraka premjestimo na štap C. Potezi premještanja za tri diska izgledaju ovako: prvi disk s A na C, drugi disk s A na B, prvi disk s C na B, treći disk s A na C, prvi disk s B na A, drugi disk s B na C te konačno prvi disk s A na C.

- Stvaranje plana rješavanja problema: sada ćemo naše opažanje primijeniti na $n$ diskova. Prvi korak je pomicanje $n-1$ diskova s A na B. Drugi korak je pomicanje preostalog diska s A na C. Na kraju treba premjestiti $n-1$ diskova s B na C. U svakom trenutku može se premjestiti samo jedan disk te se veći disk ne smije staviti na manji pa kada premještamo diskove s A na B ili s A na C ne možemo to učiniti direktno nego koristimo štap B i onda prebacujemo diskove s B na C. Do osnovnog koraka dođemo kada je broj diskova jednak 1. U tom slučaju disk s A prebacujemo na C.

- Izvršavanje osmišljenog plana: algoritam glasi:

```
def hanoi(n, A, B, C):
    if n > 0:
        hanoi(n - 1, A, C, B)
        move(A, C)
        hanoi(n - 1, B, A, C)

def move(a,c):
    print('premjesti disk s',a,'na',c)

n = int(input())
hanoi(n,'A','B','C')
```

Ispis za $n = 4$ će biti:

```
premjesti disk s A na B
premjesti disk s A na C
premjesti disk s B na C
```

```
premjesti disk s A na B
premjesti disk s C na A
premjesti disk s C na B
premjesti disk s A na B
premjesti disk s A na C
premjesti disk s B na C
premjesti disk s B na A
premjesti disk s C na A
premjesti disk s B na C
premjesti disk s A na B
premjesti disk s A na C
premjesti disk s B na C
```

- Osvrt na rješenje i metodu rješavanja: problem rješavamo tako da najveći disk zanemarimo i rješavamo problem kao da imamo jedan disk manje. Nakon toga opet ponavljamo algoritam s još jednim diskom manje, opet zanemarujemo najveći promatrani disk i rješavamo problem s još jednim diskom manje. I tako razmišljamo dalje dok ne dođemo do problema samo s jednim diskom kojeg znamo riješiti. Ovaj algoritam primjer je podjeli pa vladaj algoritma. Početni problem je podijeljen na nekoliko problema manjeg opsega, ali jednake vrste [13].

Izračunajmo broj poteza za problem s $n$ diskova. Broj poteza potrebnih za pomicanje $n$ diskova može se izraziti kao broj poteza potrebnih za pomicanje $n-1$ diskova, plus broj poteza potrebnih za pomicanje donjeg $n$-tog diska te plus broj poteza potrebnih za ponovno pomicanje $n-1$ diskova s pomoćnog diska na odredišni disk. Broj poteza možemo zapisati: $h(n) = h(n-1) + 1 + h(n-1)$, gdje je $h(n)$ broj poteza potrebnih za pomicanje $n$ diskova. Koristeći ovu formulu rekurzivno, možemo izvesti broj pomaka potrebnih za bilo koju vrijednost $n$:

$$\begin{aligned} h(n) &= h(n-1) + 1 + h(n-1) \\ &= 2 \cdot h(n-1) + 1 \end{aligned} \tag{2}$$

Izračunajmo $h(1)$. Pomicanje jednog diska zahtijeva jedan potez te je $h(1) = 1$. Dalje možemo koristiti formulu 2 za izračunavanje $h(2)$, $h(3)$, $h(4)$ i tako

dalje.

$$h(2) = 2 \cdot h(1) + 1 = 2 \cdot 1 + 1 = 3$$
$$h(3) = 2 \cdot h(2) + 1 = 2 \cdot 3 + 1 = 7$$
$$h(4) = 2 \cdot h(3) + 1 = 2 \cdot 7 + 1 = 15$$
$$h(5) = 2 \cdot h(4) + 1 = 2 \cdot 15 + 1 = 31$$

Uočavamo da broj potrebnih poteza eksponencijalno raste s $n$ i može se izraziti kao $2^n - 1$. Stoga je potrebno $2^n - 1$ poteza za pomicanje $n$ diskova.

U udžbeniku [11] obrađuju se rekurzivni potprogrami unutar cjeline Rješavanje problema programiranjem. Rekurzivni potprogrami pojavljuju se dva puta, prvo samo kao rekurzivni potprogrami pisani pseudojezikom, a kasnije unutar lekcije Programski jezik Python. Zadaci koje se pojavljuju u te dvije lekcije su identični te ćemo se osvrnuti samo na rješenja u programskom jeziku Python. Dva zanimljiva problema su problem Hanojski tornjevi te problem vezan za legendu o šahu. Legenda o šahu priča je o mudracu i kralju koji je tlačio narod. Jedan je mudrac htio poučiti kralja kako bi više trebao cijeniti svoj narod te je izumio igru šah i pokazao kako i najslabija figura - pješak - može biti ključna za ishod bitke na 64 polja i donijeti pobjedu. Kralj je bio oduševljen igrom te je mudracu ponudio nagradu kakvu god želi. On je zatražio od kralja prividno skromnu nagradu: da mu isporuči onoliko žita koliko se dobije kada se na prvo polje šahovske ploče stavi jedno zrno, a na svako naredno polje dvostruko više nego na prethodno polje. Kralj je pomislio da se radi o beznačajnoj količini, pa je predložio mudracu da zatraži nešto vrjednije. No, mudrac je ostao pri svojoj odluci. Kralj je naredio svojim slugama da mu daju njegovu nagradu, no oni su mu odgovorili da u cijelom njegovom kraljevstvu ne mogu naći toliko žita. Postavlja se pitanje: "Koliko je zrna pšenice zatražio mudrac?" Programski zadatak vezan za ovaj problem glasio bi: napišite program koji rekurzivnom funkcijom računa ukupan broj zrna pšenice na prvih $n$ polja šahovske ploče ako na prvo polje stavljamo jedno zrno, na drugo polje dva zrna i na svako sljedeće polje dvostruko više zrna nego na prethodno polje.

Prvo polje ima jedno zrno, drugo polje dva zrna, treće polje ima 4 zrna ili $2^2$, na četvrtom polju ima 8 zrna ili $2^3$, na petom polju ima 16 zrna ili $2^4$, a na $n$-tom polju ima $2^{n-1}$ zrna pšenice. Koristit ćemo ugrađenu funkciju za računanje potencije: $pow(2, n)$.

```
def sah(n):
    if n == 1:
        return 1
```

```
    else:
        return pow(2, n - 1) + sah(n - 1)

n = int(input())
zrna = sah(n)
print(zrna)
```

Za unos $n = 64$ odgovor je 18446744073709551615.

### 3.2.2 Treći razred srednje škole

Udžbenik [17] započinje poglavlje o rekurziji s primjerom u kojem za učitani broj $n$ treba napisati algoritam za umnožak prirodnih brojeva do $n$. Rješenje je raspisano kroz korake računalnog razmišljanja. Udžbenik nakog toga objašnjava rekurzivni način razmišljanja i uvodi novi pojam - stog. Osobitost rekurzivnog načina razmišljanja je postojanje strukture podataka koja se naziva stog na koji se pohranjuju varijable rekurzivne funkcije i koji funkcionira na LIFO (Last in First Out) načinu rada. To znači da podatak koji je zadnji ušao prvi izlazi. U trenutku kad unutar rekurzivne funkcije dođemo do rekurzivnog poziva s izmijenjenom vrijednošću argumenata, privremeno se prekida izvršavanje tekuće funkcije i vrijednost varijable sprema se na stog. Prvi element stoga postavlja se na dno, zatim se na njega postavlja drugi element. Tim se postupkom povećava, raste sadržaj stoga. Postupak se ponavlja sve dok se ne dođe do osnovnog slučaja, čime se završavaju rekurzivni pozivi. Nakon izvršavanja naredbe koja označava osnovni slučaj nastavlja se proces koji je bio prekinut. Učitavaju se vrijednosti koje su zapisane u stogu i to obrnutim redoslijedom nego što su postavljene.

Udžbenik [9] donosi sljedeći problema, a to je problem sparivanje zagrada. Pogledajmo aritmetički izraz $((5 + 3) \cdot (6 - (2 + 6))) \cdot ((2 + 3) \cdot 4)$. Kada iz izraza obrišemo sve osim zagrada dobijemo $(()(()))(())$. Da smo krenuli od nekog drugog aritmetičkog izraza dobili bismo neki drugi niz zagrada. Ako imamo $n$ otvorenih i $n$ zatvorenih zagrada koliko različitih nizova *dobro sparenih* zagrada možemo napraviti? Niz zagrada je *dobro sparen* ako je, čitajući slijeva nadesno, u svakom trenutku broj otvorenih zagrada veći ili jednak broju zatvorenih zagrada i ukupan broj otvorenih i zatvorenih zagrada je jednak. Na primjer niz (() (()))))(() nema *dobro sparene* zagrade. Za $n = 3$, odnosno kada imamo 3 para zagrada svi mogući nizovi su: ()()(), ()(()),(())(),((())), (()()). Ima ih pet. Neka je $f(n)$ ukupan broj nizova dobro sparenih zagrada koje se sastoje od $n$ otvorenih i $n$ zatvorenih zagrada. Za $n = 1$ jedini ispravan niz je () pa je $f(1) = 1$. Za $n = 2$ imamo dva ispravna niza, a to su ()() i (()) pa je $f(2) = 2$. Brzo uočavamo da za veći $n$ nije lako ispisati, odnosno naći sve nizove.

Problem ćemo riješiti za $n$, odnosno napisat ćemo program koji će ovisno o $n$ ispisati broj *dobro sparenih* zagrada. Svaki niz od $n$ otvorenih i $n$ zatvorenih zagrada mora početi s otvorenom zagradom, također svaka otvorena zagrada se mora zatvoriti. Svaki niz možemo zapisati kao $(X)Y$, pri čemu su $X$ i $Y$ neki nizovi dobro sparenih zagrada. Ako pretpostavimo da se u $X$ nalazi $k$ otvorenih i $k$ zatvorenih zagrada onda se u $Y$ mora nalaziti $n-k-1$ otvorenih i $n-k-1$ zatvorenih zagrada. Takav $X$ onda možemo napraviti na $f(k)$ načina, a takav $Y$ na $f(n-k-1)$ načina. Ukupno izraz $(X)Y$ možemo napraviti na $f(k) \cdot f(n-k-1)$ načina. Primijetimo da $k$ može biti bilo koji od brojeva $0, 1, \cdots, n-1$. Kada je $k = 0$ onda niz izgleda ovako $()Y$, a kada je $k = n-1$ onda niz izgleda ovako $(X)$. Ako stavimo $f(0) = 1$ broj nizova $f(k) \cdot f(n-k-1)$ se ne mijenja. Dakle, formula za ukupan broj izraza glasi:

$$f(n) = f(0)f(n-1) + f(1)f(n-2) + \cdots + f(n-2)f(1) + f(n-1)f(0),$$

uz osnovne slučaje: $f(0) = 1, f(1) = 1$.

```
def f(n):
    if n == 0 or n == 1:
        return 1
    else:
        ukupno = 0
        for k in range (0, n):
            ukupno = ukupno + f(k) * f(n - k - 1)
        return ukupno

n = int(input())
print('f({}) = {}'.format(n, f(n)))
```

Ispis za $n = 4$ je:

```
f(4) = 14
```

Proširit ćemo sadržaj iz udžbenika s rekurzivnim programom koji ispisuje sve moguće nizove od $n$ dobro sparenih zagrada.

```
def zagrade(n, otvorene, zatvorene, niz = ''):

    if len(niz) == 2 * n:
        print(niz)
        return

    if otvorene < n:
```

```
        zagrade(n, otvorene + 1, zatvorene, niz + '(')

    if zatvorene < otvorene:
        zagrade(n, otvorene, zatvorene + 1, niz + ')')

n = int(input())
zagrade(n,0,0)
```

Ispis za $n = 4$ glasi:

```
(((())))
(((()())))
(((())()))
(((()))())
((()(())))
((()()()))
((()())())
((())(()))
((())()())
(()((())))
(()(()()))
(()(())())
(()()(()))
(()()()())
```

Udžbenik [16] rekurziju obrađuje kroz primjer Fibonaccijeva niza i fraktala. Dani primjeri iz fraktala su Kochova krivulja, Kochova pahuljica te Pitagorino stablo. Pitagorino stablo spominje se i u udžbeniku [11], ali kao prošireni sadržaj. Pitagorino stablo je u udžbeniku [16] zadano preko slike, no bolji način je dan u udžbeniku [11] gdje učenik sam treba shvatiti kako izgleda slika. U tom slučaju riječ je o problemskom zadatku postavljenom na sljedeći način.

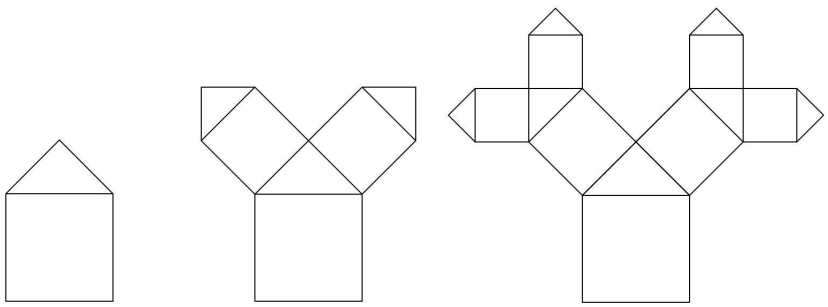
Preko zimskih praznika učenici su bili na zimovanju. Nakon igranja na snijegu učitelj je smislio igru crtanja. On je prvi započeo igru. Nacrtao je kvadrat i nad njim jednakokračni pravokutni trokut. Papir je dao prvom učeniku koji je trebao nacrtati još jedan kvadrat i po jedan pravokutni trokut nad jednom katetom pravokutnog trokuta te dati crtež sljedećem učeniku koji je onda ponovio to isto. Napišite program koji će nacrtati sliku ovisno o broju učenika $k$. Crtež je simetričan. Ako se nastavi crtati, koliko najmanje djece mora crtati kvadrate i trokute da lik i dalje bude simetričan?

Osnovna struktura Pitagorina stabla je kvadrat. Nad gornjim rubovima kvadrata crtamo jednakokračan pravokutan trokuta. Hipotenuza trokuta je

ujedno stranica kvadrata. Nad katetama crtamo nove kvadrate čija je duljina stranice jednaka duljini stranice katete, no da bismo prvo mogli nacrtati katete jednakokračnog pravokutnog trokuta nad stranicom kvadrata duljine $a$ moramo primijeniti Pitagorin poučak. Duljina katete, a ujedno i stranice novog kvadrata iznosi:

$$\begin{aligned} x^2 + x^2 &= a^2 \\ 2x^2 &= a^2 \\ x^2 &= \frac{a^2}{2} \\ x &= \frac{a}{\sqrt{2}} \end{aligned}$$

Uočimo da se crtanjem novih kvadrata nad krakom pravokutnog trokuta duljina stranice kvadrata smanjuje, a broj novonastalnih elemenata se povećava. Što je više učenika, odnosno što je veći $k$ to je više detalja na Pitagorinom stablu. Prvi korak za rješavanje je ovisno o broju učenika $k$ izračunati dubinu rekurzije. Jedan učenik samo crta jednu granu, odnosno jednu stranu. Da bi slika bila simetrična broj učenika mora biti $k = 2^n - 2$, gdje je $n$ dubina rekurzije. Odnosno, uočimo da rekurziju ne možemo zadati preko broja učenika, nego, ako želimo da slika bude simetrična ovisno o dubini $n$ možemo izračunati broj učenika.



Slika 8: Pitagorino stablo za $n = 1$, $n = 2$ i $n = 3$.

```
from turtle import *
from math import *
def kvadrat(a):
    for k in range(4):
        fd(a); rt(90)
```

```
def pitagora(a, n):
    if n >= 0:
        kvadrat(a);
        fd(a); lt(45);
        pitagora(a / sqrt(2), n - 1)
        rt(90);
        fd(a / sqrt(2));
        pitagora(a / sqrt(2), n - 1)
        rt(90);
        fd(a / sqrt(2));
        bk(a / sqrt(2));
        lt(90);
        bk(a / sqrt(2));
        lt(45);
        bk(a)

a = int(input('Duljina stranice '))
n = int(input('Dubina rekurzije '))
lt(90); ht();
pitagora(a, n)
```

### 3.2.3 Četvrti razred srednje škole

Udžbenik [18] za četvrti razred srednje škole spominje Collatzovu slutnju ili $3n + 1$ problem. Njemački matematičar Lothar Collatz 1937. godine formulirao je zadatak koji nazivamo Collatzovom slutnjom. Krenemo od nekog broja $n$ i formiramo niz brojeva prema sljedećem pravilu: ako je broj parni, podijelimo ga s 2; ako je broj neparni, pomnožimo ga s 3 i dodamo mu 1. Neriješeni matematički problem, odnosno hipoteza tvrdi da će svi prirodni brojevi nakon konačnog broja puta primjenjivanja navedenog algoritma doći do broja 1. Kada dođemo do broja 1 ponavljat će se vrijednosti 4, 2 i 1. Broj 1 je neparan broj i pomnožit ćemo ga s 3 i dodati mu broj 1, dobit ćemo paran broj 4 koji ćemo zatim podijeliti s 2 i u sljedećem koraku prvo dobiti 2 pa onda opet 1. Uočavamo da će se postupak stalno ponavljati.

Udžbenik daje zadatak za samostalni rad. Zadatak kaže da treba zapisati Collatzov problem matematički, napisati program koji će za uneseni broj ispisati niz brojeva dobivenih Collatzovim zadatkom, ispisati broj koraka potrebnih da se dođe do broja 1 te ispisati najveći član tog niza. Donosimo rješenje tog zadatka.

Matematički zapis pravila po kojem formiramo Collatzov niz:

$$f(n) = \begin{cases} \frac{n}{2}, & \text{ako je } n \equiv 0 \pmod 2, \\ 3n+1, & \text{ako je } n \equiv 1 \pmod 2. \end{cases}$$

Rekurzivno rješenje glasi:

```
def collatz(n):
    if n == 1:
        return [1]
    elif n % 2 == 0:
        return [n] + collatz(n // 2)
    else:
        return [n] + collatz(3 * n + 1)

n = int(input())
l = collatz(n)
m = max(l)
print(l)
print('Broj koraka je',len(l))
print('Najveći element je',m)
```

Ispis za $n = 6$ je:

```
[6, 3, 10, 5, 16, 8, 4, 2, 1]
Broj koraka je 9
Najveći element je 16
```

Funkcija *collatz* uzima početni broj $n$ i vraća listu s elementima Collatzova niza za taj broj. Funkcija je definirana rekurzivno: ako je $n$ jednak 1, vraća listu koja sadrži samo 1; inače, ako je $n$ paran, vraća listu koja se sastoji od $n$ iza kojeg slijedi Collatzov niz za $\frac{n}{2}$; a ako je $n$ neparan, vraća listu koja se sastoji od $n$ nakon čega slijedi Collatzov niz za $3n + 1$.

Udžbenik [17] dobro sažima prednosti i nedostatke rekurzije. Rekurzivno razmišljanje nije intuitivno. Primjenom rekurzija dobijemo kraće i jednostavnije algoritme, a neke probleme bi bilo vrlo teško riješiti na neki drugi način. Uz relativno jednostavne algoritme mogu se stvarati složene strukture kao što su fraktali. No rekurzivni programi su pri izvršavanju zahtjevniji i troše više memorije od iterativnih programa. Sporiji su u izvršavanju zato što moraju stalno spremati podatke na stog i čitati podatke s njega.

## 3.3 Zaključak i primjer obrade u nastavi

Nakon prolaska kroz udžbenike možemo zaključiti da razni udžbenici rekurziji pristupaju na isti ili sličan način. Prvo se kreće s algoritom za ispis prvih $n$ prirodnih brojeva pa se onda analogno radi takav algoritam za sumu i umnožak prvih $n$ brojeva. Udžbenici nakon toga opisuju fraktale i rekurzijom crtaju Kochovu krivulju, Kochovu pahuljicu i trokut Sierpinskog. Nakon toga neki udžbenici donose pravi problemski zadatak kao što je problem Hanojskih tornjeva i legendu o šahu i tu najčešće završava poglavlje o rekurziji. Broj sati koji se posvećuju obradi rekurzije nije velik te se puno toga nastoji obraditi u malo vremena, a rekurzija brzo postaje zahtjevna. Zato udžbenici sadrže točne te iste primjere i probleme nastojeći dati ono najvažnije i barem približiti osnovna svojstva rekurzije. Nakon takvog pristupa teško je ocijeniti usvojenost gradiva. Teško je u redovnoj nastavi dati neki drugi zadatak koji učenici nisu već vidjeli i očekivati da će sami doći do odgovora. Najviša razina znanja je da učenik za zadani problem sam piše rekurzivno rješenje, npr. kao što je rješenje zadatka o broju svih mogućih nizova dobro sparenih zagrada. Ne možemo očekivati da će svi to moći. Kada provjeravamo usvojenost znanja iz gradiva rekurzije moramo imati tipove zadataka gdje i učenici koji ne znaju sami riješiti problem, odnosno ne znaju sami doći do rekurzivnog algoritma, mogu pokazati svoje razumijevanja na drugačiji način. To ćemo napraviti tako da većina zadataka na ispitu nisu zadaci otvorenog tipa, odnosno nisu problemski zadaci.

Slijedi primjer obrade rekurzije u osnovnoj i srednjoj školi s prilagodbom s obzirom na uzrast.

- Uvođenje rekurzije: u osmom razredu osnovne škole u nastavi matematike učenici uče potencije i svojstvo $a^m \cdot a^n = a^{m+n}$. Rekurziju ćemo uvesti na pojmu potencije. Znamo da $2^n$ možemo izračunati tako da $n$ puta pomnožimo 2 sa samim sobom. Isto tako znamo da $2^n$ možemo zapisati kao $2^n = 2^1 \cdot 2^{n-1}$. Time smo problem sveli na jednostavniji jer se eksponent potencije smanjio. Kada bismo znali koliko je $2^{n-1}$ tada bismo tu vrijednost pomnožili s 2 i problem bi bio riješen. Rekurzivni način razmišljanja svodi se na rješavanje jednostavnijeg problema. Jednostavniji problem se dalje pojednostavlja dok se ne dođe do osnovnog problema čije je rješenje očito i može se jednostavno odrediti. Promotrimo sljedeći primjer: Nikola treba izračunati vrijednost potencije $2^4$. Zna da ako otkrije vrijednost potencije $2^3$ samo tu vrijednost treba pomnožiti s 2 i riješit će problem. Zato je pitao Ljiljanu zna li ona koliko je $2^3$. Ljiljana nije odmah znala odgovor na to pitanje, ali se dosjetila da ako sazna koliko je $2^2$ da će onda to pomnožiti s 2 i dobiti odgovor. Zato je pitala Zvonimira koliko je $2^2$. Zvonimir nije odmah znao, ali se

dosjetio da ako sazna koliko je $2^1$ da će onda lako taj broj pomnožiti s 2 i dobiti odgovor. Zato je pitao Sanju zna li ona koliko je $2^1$. Sanja je odmah znala odgovor jer je riječ o trivijalnom slučaju. Potencija broja čiji je eksponent jedan upravo je taj isti broj. Svoj odgovor proslijedila je Zvonimiru. Zvonimir je dobiveni odgovor pomnožio s 2, dobio 4 i svoj odgovor proslijedio Ljiljani. Ljiljana je dobiveni odgovor pomnožila s 2, dobila 8 i svoj odgovor proslijedila Nikoli. Nikola je dobiveni odgovor pomnožio s 2 i dobio rješenje originalnog problema, a to je 16. Temelj rješenja ovog problema bio je naći pravilo (relaciju) po kojoj se problem sveo na jednostavniji. Uz to trebamo i imati osnovni slučaj do kojeg ćemo doći uzastopnim svođenjem problema na jednostavniji. Takvo rješenje problema zove se rekurzivno rješenje, a postupak kojim smo došli do njega je rekurzivni način razmišljanja.

- Rekurzivna relacija: matematičkim jezikom zapišemo pravilo koje smo našli.

$$2^n = \begin{cases} 2, & \text{ako je } n = 1, \\ 2 \cdot 2^{n-1}, & \text{ako je } n > 1. \end{cases}$$

- Rekurzivna funkcija: kada smo došli do rekurzivne relacije onda pišemo rekurzivnu funkciju u programskom jeziku. Rekurzivna funkcija za računanje $2^n$ iz prethodnog primjera glasi:

```
def potencija(n):
    if n == 1:
        return 2
    else:
        return 2 * potencija(n - 1)

n = int(input('Unesi n '))
p = potencija(n)
print(p)
```

- Izvršavanje gotovih rekurzivnih relacija: ovaj dio služi usvajanju pisanja rekurzivnih funkcija. Proći ćemo kroz već napisane rekurzivne relacije koje ćemo implementirati. Primjer za osmi razred: neka je zadana rekurzivna relacija $f$. Odredi vrijednost rekurzivne relacije $f$ za parametar $x = 4$ te napiši rekurzivnu funkciju i odredi vrijednost $f(100)$.

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

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

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

Rekurzivna funkcija glasi:

```
def f(x):
    if x > 0:
        return f(x - 3) - 2

    else:
        return 1 - x

x = int(input())
print('f({}) = {}'.format(x, f(x)))
```

Ispis za $x = 100$ je:

```
f(100) = -65
```

Primjer za srednju školu: neka je zadana rekurzivna relacija $f$. Odredi vrijednost rekurzivne relacije $f$ za parametar $x = 12$ te napiši rekurzivnu funkciju i odredi vrijednost $f(42)$.

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

Rješenje:

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

Rekurzivna funkcija glasi:

```
def f(x):
    if x > 3:
        return f(x - 3) + 3

    elif x == 3:
        return 2 * x + 1

    else:
        return x * x + 2

x = int(input())
print('f({}) = {}'.format(x, f(x)))
```

Ispis za $x = 16$ je:

```
f(42) = 46
```

- Pisanje rekurzivne relacije: pisanje rekurzivne relacije najteži je dio. Sami trebamo osmisliti rekurzivno rješenje. Da bismo odredili rekurzivno rješenje nekog konkretnog problema zamislimo da znamo riješiti sve probleme koji su manji od njega. Tada je potrebno taj zadani problem riješiti uporabom gotovih rješenja manjih problema. Primjer: Marko živi na prvom katu zgrade, a od ulaza u zgradu do njegovog stana vodi $n$ stuba. Svakim korakom Marko se može popeti za po jednu ili za po dvije stube. Na koliko se različitih načina Marko može popeti od ulaza u zgradu do svojeg stana? Napišimo funkciju $f$ koja za broj stuba $n$ vraća broj različitih načina penjanja. Kada imamo jednu stubu onda znamo da se može popeti na tu jednu stubu na jedan način tako da napravi jedan mali korak. Ako imamo dvije stube, Marko se može popeti s pomoću dva mala koraka penjući se stubu po stubu ili s jednim velikim, penjući se po dvije stube. Broja načina je dva. Znači, za $n = 1$ i $n = 2$ znamo riješiti problem. Sada samo trebamo vidjeti da li se za svaki sljedeći $n > 2$ problem može svesti na te slučajeve koje znamo riješiti. Za $n$ stuba, ako se Marko prvim korakom popne za jednu stubu, morat će se nakon tog popeti za još $n - 1$ stuba. Ako je $f(n)$ funkcija koja vraća broj traženih načina penjanja na $n$ stuba, onda se Marko na $n - 1$ stuba može popeti na $f(n - 1)$ načina. Ako se Marko prvim korakom popne za dvije stube, morat će se nakon toga popeti za još $n - 2$ stube. To može napraviti na $f(n - 2)$ načina. $f(n)$ svodi se na $f(n - 1)$ i $f(n - 2)$ pa se onda analogno $f(n - 1)$ svodi na $f(n - 2)$ i $f(n - 3)$, $f(n - 2)$ svodi se na $f(n - 3)$ i $f(n - 4)$, $\cdots$, $f(3)$

svodi se na $f(2)$ i $f(1)$, a $f(2)$ i $f(1)$ znamo riješiti. Da bismo našli ukupan broj načina penjanja na $n$ stuba zbrojit ćemo sve načine ako se prvo popeo za jednu stubu i sve načine ako se prvo popeo za dvije stube, odnosno $f(n) = f(n-1) + f(n-2)$.

Gotova rekurzivna relacija glasi:

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

Jednom kada imamo rekurzivnu relaciju primjenjujemo znanje koje smo stekli izvršavanjem gotovih rekurzivnih relacija te pišemo rekurzivnu funkciju. Rekurzivna funkcija glasi:

```
def f(n):

    if n == 1:
        return 1

    if n == 2:
        return 2

    else:
        return f(n - 1) + f(n - 2)

n = int(input())
print(f(n))
```

Uočimo da je broj načina penjanja na $n$ stuba jednak $n$-tom Fibonaccijevom broju.

Cilj ovakvog pristupa obradi rekurzije je da učenik nakon svih provedenih dijelova može samostalno riješiti problemski zadatak. No, također rekurzija poprima širi značaj te u provjeru znanja možemo uključiti i zadatke u kojima iz dane rekurzivne relacije učenik treba napisati rekurzivnu funkciju ili pak zadatke u kojima učenik treba odrediti $f(x)$ za dani $x$. U obradi rekurzije naglasak treba biti na formiranju rekurzivnog načina razmišljanja i razvoju sposobnosti rješavanja problema, a ne memoriziranju poznatih rekurzivnih algoritama.

# Literatura


[1] I. Anderson, *A First Course in Discrete Mathematics*, Springer Verlag, 2001.

[2] S. Amarasinghe i dr., *Recursion*, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 2015. `https://web.mit.edu/6.005/www/fa15/classes/10-recursion/` (kolovoz 2022)

[3] M. Babić i dr., *# mojportal 8. Udžbenik informatike u osmom razredu osnovne škole*, Školska knjiga, 2021.

[4] L. Budin i dr., *Napredno rješavanje problema programiranjem u Pythonu*, Element, 2018.

[5] L. Budin i dr., *Rješavanje problema programiranjem u Pythonu*, Element, 2014.

[6] P. J. Cameron, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, 1994.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *Introduction To Algorithms. Third Edition.* The MIT Press Cambridge, Massachusetts, USA, 2009.

[8] S. Deljac i dr., *Informatika 8. Udžbenik u osmom razredu osnovne škole*, Profil Klett, 2021.

[9] N. Dmitrović i dr., *Informatika 3. Udžbenik iz informatike za 3. razred prirodoslovno-matematičkih gimnazija*, Udžbenik.hr d.o.o., 2020.

[10] J. N. Gregg, *More on recursion*, Lawrence University, Appleton, Wisconsin, 2021. `http://www2.lawrence.edu/fast/GREGGJ/CMSC210/algorithms/recursion.html` (kolovoz 2022.)

[11] I. Kniewald i dr., *Informatika+ 8. Udžbenik iz informatike za 8. razred osnovne škole*, Udžbenik.hr d.o.o., 2021.

[12] Kurikulum nastavnoga predmeta Informatika za osnovne i srednje škole, Narodne Novine 22/2018. `https://narodne-novine.nn.hr/clanci/sluzbeni/2018_03_22_436.html` (kolovoz 2022.)

[13] X. Ren i D. Yang, *Analysis of Recursive Teaching of Computational Thinking*, Advances in Social Science, Education and Humanities Research, vol. 480, (2020), 661-665.

[14] B. Rihter i dr., *Like IT 8. Udžbenik iz informatike za osmi razred osnovne škole*, Alfa, 2021.

[15] S. S. Skiena, *The Algorithm Design Manual. Second Edition.*, Springer-Verlag, 2008.

[16] D. Šafar Đerki i dr., *Svijet informatike 3. Udžbenik informatike s dodatnim digitalnim sadržajima u trećem razredu gimnazija*, Školska knjiga, 2020.

[17] I. Tomičić i dr., *Think IT 3. Udžbenik iz informatike za treći razred gimnazije*, Alfa, 2020.

[18] T. Volarić i dr., *Think IT 4. Udžbenik iz informatike za četvrti razred gimnazije*, Alfa, 2021.

[19] D. Walter, *Recursive Functions*, The Stanford Encyclopedia of Philosophy (Winter 2021 Edition), Edward N. Zalta (ed.), 2021. `https://plato.stanford.edu/archives/win2021/entries/recursive-functions/` (kolovoz 2022)

# Sažetak

U ovom diplomskom radu definirali smo rekurziju i dali primjere rekurzivnih algoritama u programskom jeziku Python. Fibonaccijeve brojeve i binomne koeficijente računali smo rekurzivno te smo usporedili učinkovitost rekurzivnog rješenja s iterativnim rješenjem mjereći vrijeme izvršavanja algoritama. Rekurzivne algoritme unaprijedili smo memoizacijom. Dalje smo generirali kombinatorne objekte: permutacije, kombinacije te sve podskupove danog skupa. Kornjačinom grafikom crtali smo fraktalne skupove: Kochovu krivulju, binarno stablo i trokut Sierpiński. Opisali smo dva rekurzivna algoritma za sortiranje: mergesort i quicksort te pokazali njihovo djelovanje na primjeru. Testirali smo njihovu brzinu u usporedbi s ugrađenom funkcijom u programskom jeziku Python te analizirali njihovu vremensku složenost. U poglavlju rekurzija u školi promatrali smo kako udžbenici za osnovnu i srednju školu obrađuju rekurziju. Na kraju smo dali osvrt na udžbenike i primjer kako obraditi rekurziju u nastavi.

# Summary

In this thesis, we defined recursion and gave examples of recursive algorithms in the Python programming language. Fibonacci numbers and binomial coefficients were calculated recursively and we compared the efficiency of the recursive solution with the iterative solution by measuring the execution time of the algorithms. We improved the recursive algorithms by memoisation. We generated combinatorial objects: permutations, combinations and all subsets of the given set. With the turtle graphics, we drew fractal sets: the Koch curve, the binary tree and the Sierpiński triangle. We described two recursive algorithms for sorting: mergesort and quicksort and showed their operation on an example. We tested their speed in comparison with a built-in function in the Python programming language and analysed their time complexity. In the chapter on recursion at school, we observed how textbooks for primary and secondary schools deal with recursion. At the end, we gave an overview of textbooks and an example of how to teach recursion in class.

# Životopis

Rođena sam u Zagrebu 1996. godine. Osnovnu školu Matije Gupca pohađala sam u Područnoj školi Dobri Zdenci te u matičnoj školi u Gornjoj Stubici. U Zaboku upisujem Gimnaziju Antuna Gustava Matoša, prirodoslovno-matematički smjer te 2015. godine upisujem Prirodoslovno-matematički fakultet, preddiplomski studij matematike; smjer nastavnički kojeg završavam u veljači 2019. godine. U rujnu iste godine upisujem Diplomski studij matematike i informatike; smjer: nastavnički. Tijekom studija radila sam u softverskoj tvrtki Photomath d.o.o. te kao učiteljica matematike u Osnovnoj školi Tina Ujevića u Zagrebu. Udana sam i majka dvoje djece.