Vežbe 1

Kompleksnost algoritama, algoritmi pretrage i sortiranja

01

Kompleksnost algoritama

Kako se definiše kompleksnost algoritma?

Koje su klase kompleksnosti algoritama?

Kako odrediti kompleksnost algoritma?

Šta je algoritam?

Algoritam je višekoračni postupak za izvršavanje nekog zadatka u ograničenom vremenskom periodu.

Ulazni podaci → **ALGORITAM** → **Izlazni podaci**

Poreklo reči: Abū ʿAbdallāh Muḥammad ibn Mūsā *al-Khwārizmī*

Šta je kompleksnost algoritma?

Kompleksnost (složenost) algoritma: *maksimalni broj operacija potrebnih za njegovo izvršavanje.*

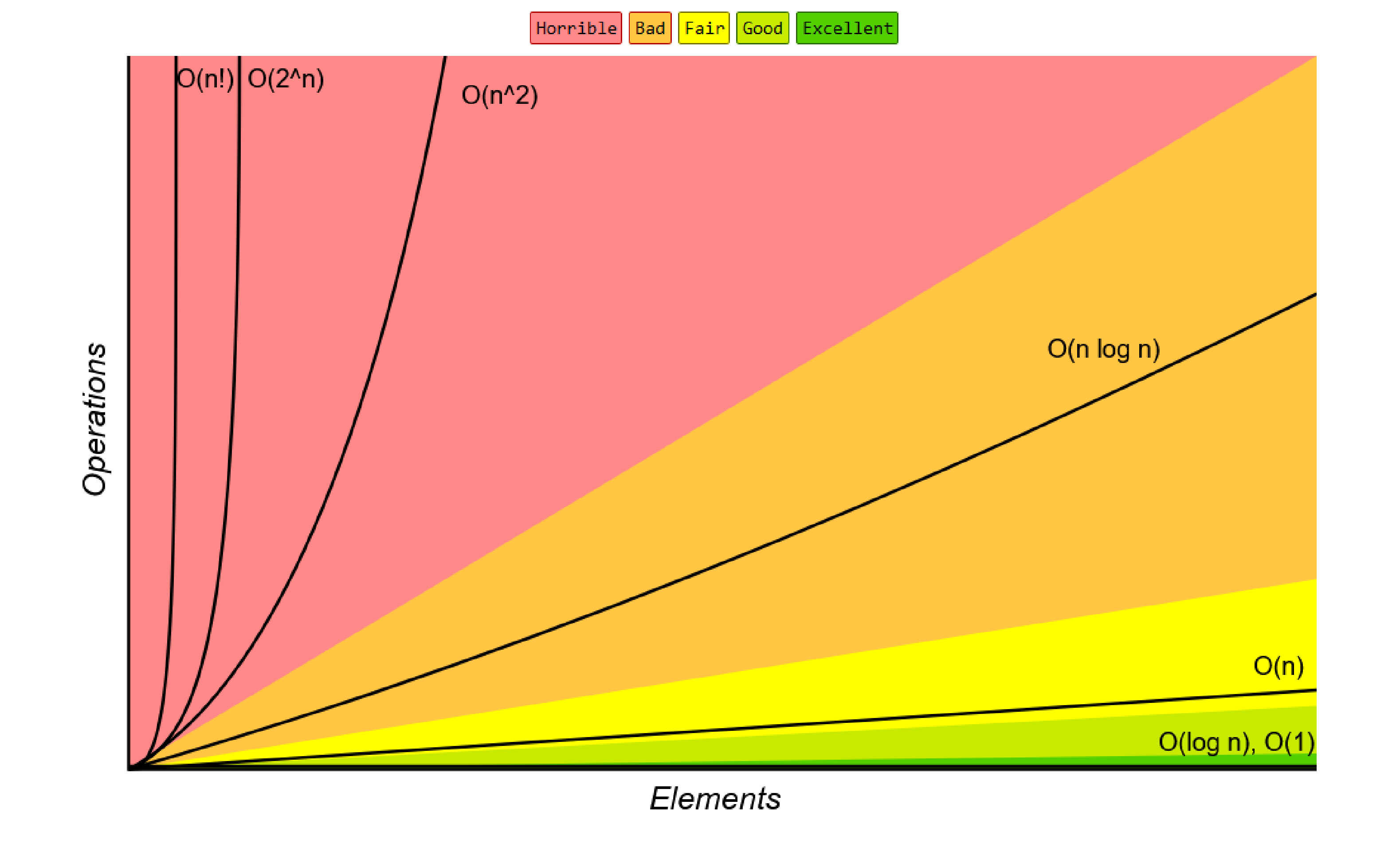
- **Zavisi od veličine ulaza (broja ulaznih podataka)**
- Koristimo **asimptotsku notaciju ->** procena vremena izvršavanja algoritma kada je ulaz veeeliko *n*
- Zanimaju nas **najgori slučajevi** -> oznaka *O(n)*

Klase kompleksnosti algoritma

$O(1) < O(log_2n) < O(n) < O(nlog_2n) <$

$O(n^2) < O(2^n) < O(n!)$

Klase kompleksnosti algoritma



Klase kompleksnosti algoritma

$n \setminus f(n)$	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 *ms*
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 *ms*	77.1 *god*
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 *s*	8.4×10^{15} *god*
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 *min*	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 *dan*	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} *god*	
1,000	0.010 μs	1.0 μs	9.966 μs	1 *ms*		
10,000	0.013 μs	10 μs	130 μs	100 *ms*		
100,000	0.017 μs	0.10 μs	1.67 *ms*	10 *s*		
1,000,000	0.020 μs	1 *ms*	19.93 *ms*	16.7 *min*		
10,000,000	0.023 μs	0.01 *s*	0.23 *s*	1.16 *dan*		
100,000,000	0.027 μs	0.10 *s*	2.66 *s*	115.7 *dan*		
1,000,000,000	0.030 μs	1 *s*	29.9 *s*	31.7 *god*		

Klase kompleksnosti algoritma

O(1):

```
def prviElementNiza(niz):
        print("Prvi element niza je ", niz[0])
```

- Dodavanje i uklanjanje elementa sa stack-a ili queue-a, provera da li je broj paran......
- *O(1)* ne mora nužno da označava izvršavanje u jako kratkom vremenskom intervalu

Klase kompleksnosti algoritma

$O(log_2 n)$:

```
def deljenjeSaDva(broj):
        while broj > 1:
            broj /= 2
```

- Binarna pretraga, pojedini algoritmi sortiranja *(videćemo kasnije)*

Klase kompleksnosti algoritma

O(n):

```
def ispisNaKonzolu(n):
        for broj in range(1, n+1):
            print(broj)
```

- Linearna pretraga, “obilazak” niza, provera da li je reč palindrom...

Klase kompleksnosti algoritma

$O(nlog_2n)$:

```
def primerNLogN(n):
        for i in range(1, n+1):
            for j in range(1, int(math.log(n, 2)) + 1):
                print("Vrednost 1: ", i, "Vrednost 2: ", j)
```

- *„Divide and Conquer“* algoritmi sortiranja (Merge sort, Heap sort... – *videćemo kasnije*)

Klase kompleksnosti algoritma

$O(n^2)$:

```
def dvePetlje(n):
        for broj1 in range(1, n+1):
            for broj2 in range(1, n+1):
                print("Zbir brojeva je: " broj1 + broj2)
```

- Bubble sort, Selection sort, Insertion sort.... (*videćemo kasnije*)

Klase kompleksnosti algoritma

$O(2^n)$:

```
def fibonacci(n):
        if n <= 1:
          return n
        else:
          return(fibonacci(n-1) + fibonacci(n-2))
```

- Uglavnom kod rekurzivnih algoritama (Fibonači, Hanojske kule...)

Klase kompleksnosti algoritma

O(n!):

```
def faktorijelFunkcija(n):
        for broj in range(1, n+1):
            faktorijelFunkcija(n-1)
```

- „Brute force“ rešenje problema trgovačkog putnika, generisanje permutacija niza ili stringa...

Kako odrediti kompleksnost algoritma?

KORAK 1

DEFINISANJE VREMENA IZVRŠAVANJA NAREDBI

Za svaku naredbu odrediti koliko će se maksimalno puta izvršiti, zapisati kompleksnost njenog izvršavanja i sabrati sve dobijene vrednosti

KORAK 2

ZADRŽAVANJE NAJVEĆEG ČLANA

Zadržati samo najveći sabirak u dobijenom izrazu, jer on najviše utiče na složenost

KORAK 3

IGNORISANJE KONSTANTI

Ignorisati sve konstantne vrednosti u dobijenom izrazu, jer ne utiču na formiranje složenosti

Kako odrediti kompleksnost algoritma?

Primer 1 - Odrediti složenost sledećeg koda:

```
def zbirKvadrataElemenata(a, b, c):
        a2 = a * a
        b2 = b * b
        c2 = c * c
        zbir = a2 + b2 + c2
        print("Zbir kvadrata prosleđenih elementa je ", zbir)
        return zbir
```

$T(n) = c_1$ (const.)

Broj naredbi: $5 * c_1$
Kompleksnost algoritma: *O(1)*

Kako odrediti kompleksnost algoritma?

Primer 2 - **Odrediti složenost sledećeg koda:**

```
a = b + c

for broj1 in range(1, n+1):
        broj1 *= 2

for broj2 in range(1, n+1):
        broj2 += 4
```

$T(n) = c_1$

$T(n) = c_2 * n$

$T(n) = c_3 * n$

Broj naredbi: $c_1 + c_2 n + c_3 n$
Kompleksnost algoritma: *$O(n)$*

Kako odrediti kompleksnost algoritma?

Primer 2 - Odrediti složenost sledećeg koda:

```
def ifElse(n, uslov):
        if uslov:
            for i in range(1, n+1):
                print(i)
        else:
            for j in range(1, n+1):
                for k in range(2, n+2):
                    print(j*k)
```

$T(n) = c_1 * n$

$T(n) = c_2 * n^2$

Broj naredbi: $c_1 n + c_2 n^2$

Kompleksnost algoritma: $O(n^2)$

02

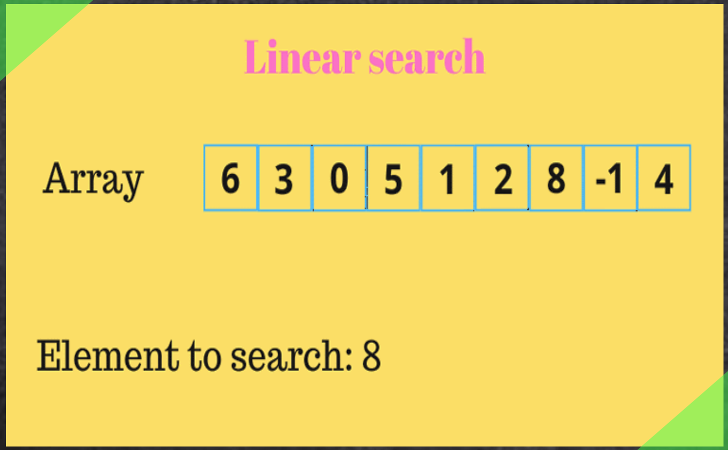
Algoritmi pretrage

- Linearna pretraga
- Binarna pretraga

Linearna pretraga

- **Najjednostavniji algoritam pretrage**
- **Proverava svaki element u nizu dok se:**
  - **ne pronađe željeni element**
  - **ne dođe do kraja niza**
- **Kao povratna vrednost vraća se indeks traženog elementa (pozicija u nizu)**
- **Implementacija u Python-u:**

```python
def linearSearch(niz, zeljeniBroj):
        for index in range(len(niz)):
            if niz[index] == zeljeniBroj:
                return index
        return -1
```



Kompleksnost:
O(n)

Linearna pretraga sa stražarom (Sentinel search)

- **Mana prethodnog algoritma: dve provere u svakoj iteraciji (da li se došlo do kraja niza, da li je element pronađen)**
- **Ovo se može ispraviti tako što će se na kraj niza (umesto poslednjeg elementa) dodati željeni element, čime se ostavlja samo jedna provera**

Linearna pretraga sa stražarom (Sentinel search)

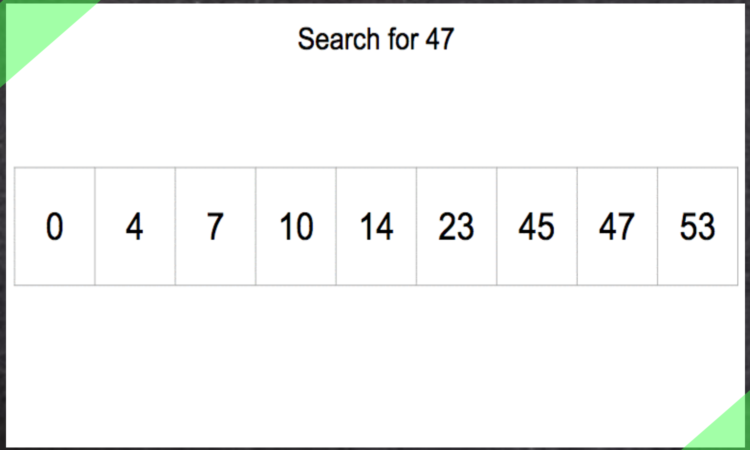
- **Implementacija u Python-u:**

```
def sentinelSearch(niz, zeljeniBroj):
        duzinaNiza = len(niz)
        poslednjiEl = niz[duzinaNiza - 1]
        niz[duzinaNiza - 1] = zeljeniBroj

        index = 0
        while (niz[index] != zeljeniBroj):
                index += 1

        niz[duzinaNiza - 1] = poslednjiEl

        if ((index < duzinaNiza – 1) or (niz[duzinaNiza - 1] == zeljeniBroj)):
          return index
        else:
          return -1
```

Binarna pretraga

- **Zahteva sortiran niz!**
- **Ideja algoritma:**

1. **Pronaći element u sredini niza**
2. **Uporediti ga sa traženim elementom**
   - **Ako su jednaki, pronašli smo naš element**
   - **Ako je traženi element veći od elementa u sredini, pretražujemo desnu polovinu niza**
   - **U suprotnom, pretražujemo levu polovinu niza**



Binarna pretraga

- **Implementacija u Python-u:**

```
def binarySearch(niz, zeljeniBroj):
        duzinaNiza = len(niz)
        donjaGranica = 0
        gornjaGranica = duzinaNiza - 1
        sredina = 0

        while donjaGranica <= gornjaGranica:
            sredina = (donjaGranica + gornjaGranica) // 2

            if niz[sredina] < zeljeniBroj:
                donjaGranica = sredina + 1
            elif niz[sredina] > zeljeniBroj :
                gornjaGranica = sredina - 1
            else:
                return sredina

        return -1
```

Kompleksnost:
$O(\log_2 n)$

Binarna pretraga

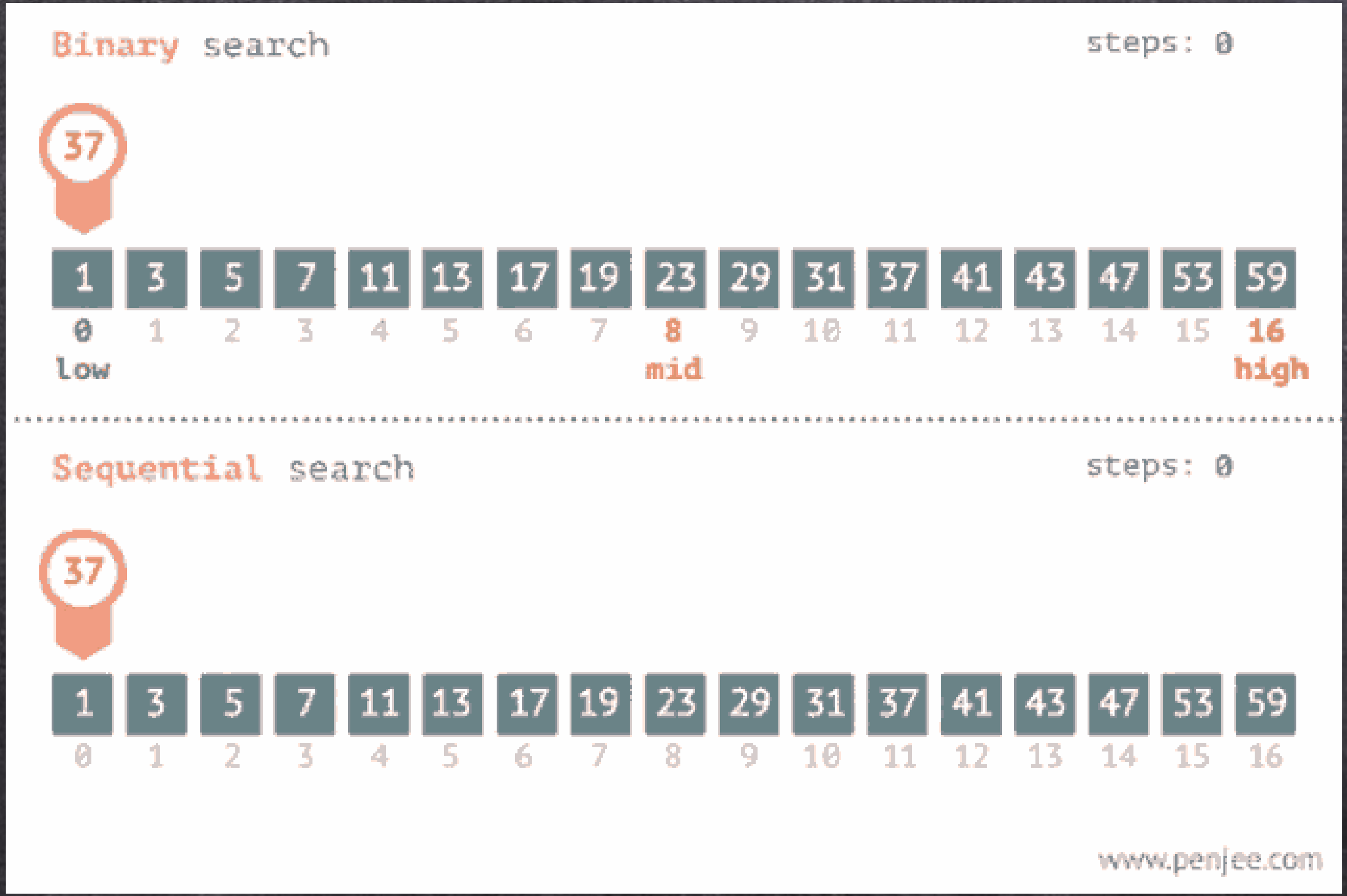
- **Implementacija u Python-u (verzija 2):**

Binarna pretraga

- **Implementacija u Python-u (verzija 2):**

```
def binarySearchRecursive(niz, donjaGranica, gornjaGranica, zeljeniBroj):
        if gornjaGranica >= donjaGranica:
            sredina = (donjaGranica + gornjaGranica) // 2

            if niz[sredina] == zeljeniBroj:
                return sredina
            elif niz[sredina] > zeljeniBroj :
                return binarySearchRecursive(niz, donjaGranica, sredina – 1, zeljeniBroj)
            else:
                return binarySearchRecursive(niz, sredina + 1, gornjaGranica, zeljeniBroj)

        else:
            return -1
```

Linearna pretraga vs Binarna pretraga



03

Algoritmi sortiranja

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Bubble sort

Kompleksnost:
$O(n^2)$

8 5 3 1 4 7 9

- **Najjednostavniji algoritam sortiranja**
- **Poredi svaka dva susedna elementa i menja im mesto po potrebi**
- **Postupak se ponavlja sve dok se ne dobije sortiran niz**
- **Ne koristi se zbog prevelike kompleksnosti**

- **Implementacija u Python-u:**

```
def bubbleSort(niz):
    for ind1 in range(len(niz)):
        for ind2 in range(0, len(niz) – ind1 - 1):
            if niz[ind2] > niz[ind2 + 1]:
                niz[ind2], niz[ind2 + 1] = niz[ind2 + 1], niz[ind2]
```

Bubble sort

- **Mana prethodnog rešenja: šta ako je niz u startu sortiran?**

```
def bubbleSortOptimized(niz):
    for ind1 in range(len(niz)):

        izvrsenaIzmena = False

        for ind2 in range(0, len(niz) – ind1 - 1):
            if niz[ind2] > niz[ind2 + 1]:
                niz[ind2], niz[ind2 + 1] = niz[ind2 + 1], niz[ind2]
                izvrsenaIzmena = True

        if not izvrsenaIzmena:
            break
```

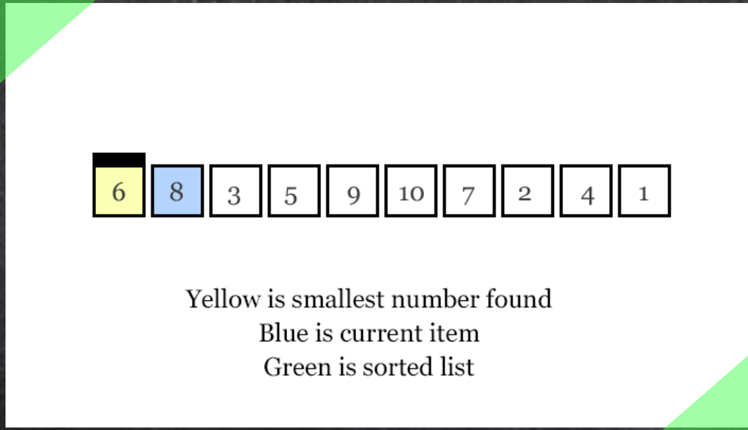
Ako u nekom prolasku kroz niz vrednost ostane false, znači da su svi elementi već sortirani i da ne treba nastavljati dalje

Selection sort

Kompleksnost:
$O(n^2)$

- Pronalazi se najmanji element u nizu i smešta na prvo mesto, potom se traži sledeći najmanji element i smešta na drugo mesto...
- Postupak se ponavlja sve dok se ne dobije sortiran niz
- Jednostavan, ali se ne koristi zbog prevelike kompleksnosti
- Najgori slučaj: najmanji element na poslednjem mestu, svi ostali pravilno sortirani
- Implementacija u Python-u:

```python
def selectionSort(niz):
    for index in range(len(niz)):
        min_index = index

        for j in range(index + 1, len(niz)):
            if niz[j] < niz[min_index]:
                min_index = j

        niz[index], niz[min_index] = niz[min_index], niz[index]
```



Insertion sort

- **Postupak koji podseća naređanje karata u ruci**
- **Uzima se element iz nesortiranog dela niza i postavlja se na odgovarajuće mesto u sortiranom delu niza**
- **Postupak se ponavlja sve dok se ne dobije sortiran niz**
- **Najgori slučaj: elementi sortirani u opadajućem redosledu**

Kompleksnost:
$O(n^2)$

6 5 3 1 8 7 2 4

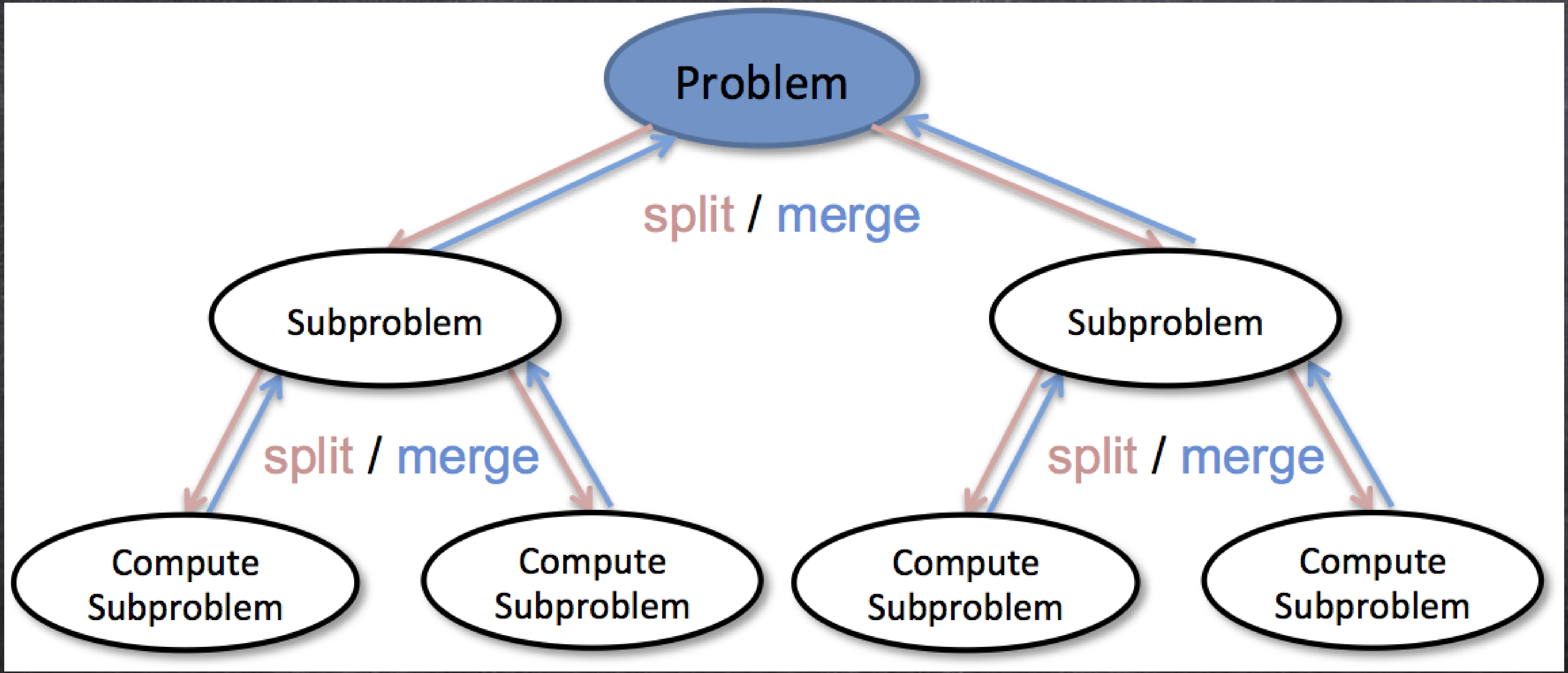
- **Implementacija u Python-u:**

```
def insertionSort(niz):
    for index in range(1, len(niz)):
        trenutniElement = niz[index]
        j = index - 1

        while j >= 0 and trenutniElement < niz[j]:
            niz[j + 1] = niz[j]
            j -= 1

        niz[j + 1] = trenutniElement
```

Podeli i osvoji (Divide-and-conquer)

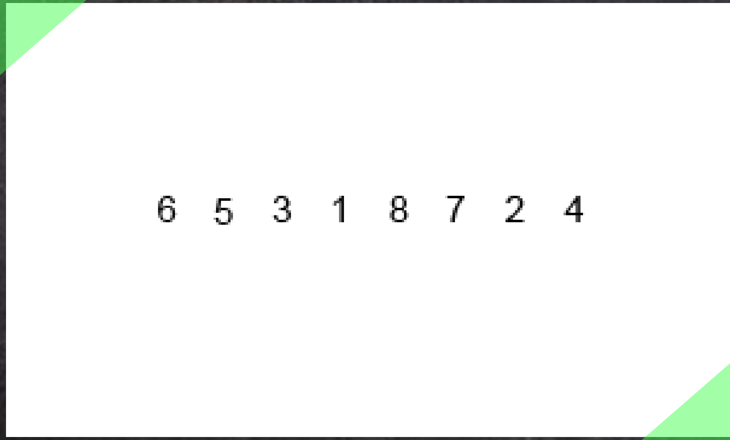
- **Paradigma za dizajniranje algoritama koja se zasniva na podeli problema na potprobleme**
- **Postupak podele problema na potprobleme se ponavlja dok potproblemi ne postanu dovoljno mali da mogu da se reše bez problema**
- **Rešenja potproblema se potom kombinuju kako bi se došlo do konačnog rešenja početnog problema**
- **Primeri upotrebe: Merge sort, Quick sort**

Podeli i osvoji (Divide-and-conquer)



Merge sort

Kompleksnost:
$O(n\log_2 n)$



- Ideja: rekurzivno razdvojiti niz na podnizove, tako da svaki sadrži 1 element, potom ih spojiti tako da se dobije sortiran niz
  - ➢ Pogodan za paralelizaciju
- Često se koristi u praksi
- Postoji više podvrsta
  - Top-down merge sort
  - Bottom-up merge sort
  - Natural merge sort
  - Ping-pong merge sort
  - …

- **Implementacija u Python-u (malo duža ☺):**

```
def mergeSort(niz):
    if len(niz) > 1:
        sredina = len(niz) // 2
        L = niz[:sredina]
        R = niz[sredina:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

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

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

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

Vrši se deljenje na levi i desni podniz

Rekurzivno se ponavlja proces za svaki podniz, dok se ne dobije da svaki podniz ima samo 1 element

Porede se elementi iz levog i desnog podniza i postavljaju na odgovarajuću poziciju

Kada se u sortirani niz uvrste svi elementi iz nekog od podnizova, treba uvrstiti i preostale elemente iz drugog podniza

Quick sort

Kompleksnost:
Najgori slučaj: $O(n^2)$
Prosečan slučaj: $O(nlog_2n)$

6 5 3 1 8 7 2 4

- Ideja: odabrati „pivot“ element i podeliti niz u 2 podniza: prvi, u kom su svi elementi manji od „pivota“ i drugi, u kom su svi elementi veći od „pivota“
  - Postupak se ponavlja rekurzivno za svaki podniz
- Nekoliko različitih strategija odabira „pivot“ elementa
  - Prvi element u nizu
  - Poslednji element u nizu
  - Random element
  - „Median-of-three“ pravilo
- Kompleksnost u najgorem slučaju $O(n^2)$, što ga čini lošijim od Merge sort-a, ali u prosečnom slučaju kompleksnost ova dva algoritma je jednaka
  - Quick sort je brži kada se podaci već nalaze u memoriji
  - Merge sort je brži kada je potrebno učitavati podatke
  - Quick sort ima manju memorijsku zahtevnost, ali je komplikovaniji za razumevanje i implementaciju

- **Implementacija u Python-u:**

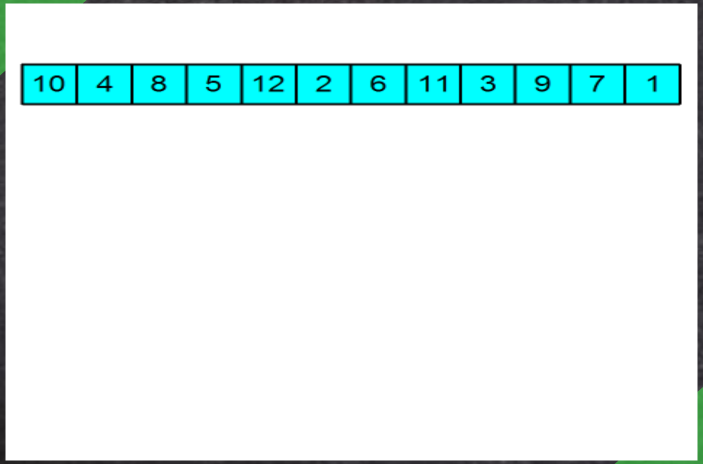
```
def partition(niz, donjaGranica, gornjaGranica):

    pivot = niz[gornjaGranica]

    i = donjaGranica – 1

    for j in range(donjaGranica, gornjaGranica):
        if niz[j] <= pivot:
            i += 1
            (niz[i], niz[j]) = (niz[j], niz[i])

    (niz[i + 1], niz[gornjaGranica]) = (niz[gornjaGranica], niz[i + 1])

    return i + 1


def quickSort(niz, donjaGranica, gornjaGranica):

    if donjaGranica < gornjaGranica:

        pi = partition(niz, donjaGranica, gornjaGranica)
        quickSort(niz, donjaGranica, pi - 1)
        quickSort(niz, pi + 1, gornjaGranica)
```

Poslednji element u nizu je "pivot"

Pri prvom pozivu funkcije, donja granica je 0, a gornja granica je broj elemenata u nizu

Heap sort

Kompleksnost:
$O(nlog_2n)$



- Koristi se heap, struktura podataka koja liči na binarno stablo

- Algoritam se sastoji od 2 faze:
  - Kreiranje heap-a
  - Sortiranje elemenata iz heap-a

- Kompleksnost ista kao kod Quick sorta, ali Quick sort brži u prosečnom slučaju
  - Prednost Heap sort-a: garantovana kompleksnost

- **Implementacija u Python-u:**

```
def heapify(niz, n, index):
    najveci = index
    levi = 2 * index + 1
    desni = 2 * index + 2

    if levi < n and niz[index] < niz[levi]:
        najveci = levi

    if desni < n and niz[najveci] < niz[desni]:
        najveci = desni

    if najveci != index:
        (niz[index], niz[najveci]) = (niz[najveci], niz[index])
        heapify(niz, n, najveci)


def heapSort(niz):
    n = len(niz)

    for i in range(n//2 - 1, -1, -1):
        heapify(niz, n, i)

    for j in range(n-1, 0, -1):
        (niz[j], niz[0]) = (niz[0], niz[j])
        heapify(niz, j, 0)
```

Priority queue

- *Struktura podataka koja se sastoji od uređenih parova (prioritet, vrednost)*
- Elementi se dodaju u red jedan po jedan, a iz reda se prvo izvlači element sa najvećim prioritetom
- Obično se implementira korišćenjem heap-a
- Primena:
  - U operativnim sistemima, za odabir procesa
  - U algoritmima veštačke inteligencije (npr. A* search)
  - U radu sa grafovima, npr. Dijkstra algoritam za pronalaženje najkraćeg puta

Sledeći termin ☺

- **Implementacija u Python-u:**

```
class priorityQueue(object):
    def __init__(self):
        self.queue = []

    def isEmpty(self):
        return len(self.queue) == 0

    def insert(self, data):
        self.queue.append(data)

    def delete(self):
        try:
            max_vrednost = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max_vrednost]:
                    max_vrednost = i
            element = self.queue[max_vrednost]
            del self.queue[max_vrednost]
            return element
        except:
            exit()
```

- **Upotreba u Python-u ☺:**

```
from queue import PriorityQueue

q = PriorityQueue()

q.put((4, ‘Čitanje’))
q.put((2, ‘Učenje’))
q.put((5, ‘Pisanje’))
q.put((1, ‘Spavanje’))
q.put((3, ‘Kuvanje’))

while not q.empty():
    next_item = q.get()
    print(next_item)
```

Ručna implementacija:
Veći broj → veći prioritet (*max-priority*)

Gotova implementacija:
Manji broj → veći prioritet (*min-priority*)

04

Zadaci za vežbu

1

Napisati algoritam koji u **strogo rastućem** sortiranom nizu $\mathbb{Z}^{0+}$ brojeva pronalazi najmanji element koji nije član tog niza. Optimizovati algoritam tako da ima najmanju moguću složenost.

Primer:

[0, 1, 2, 4] - najmanji broj koji se ne nalazi u nizu je broj 3
[1, 2, 3, 4] - najmanji broj koji se ne nalazi u nizu je broj 0
[0, 1, 2, 3, 4] - najmanji broj koji se ne nalazi u nizu je broj 5

2

Napisati algoritam koji u **strogo rastućem** sortiranom nizu prirodnih brojeva proverava da li postoji broj koji se nalazi na *n*-toj poziciji u nizu i ima vrednost *n*. Optimizovati algoritam tako da ima najmanju moguću složenost.

3

Napisati algoritam koji pronalazi najveći mogući proizvod dva broja iz nekog niza. Optimizovati algoritam tako da ima najmanju moguću složenost.

4

Napisati algoritam koji u nekom nizu pronalazi sve uređene trojke takve da im je zbir <= 100. Odrediti njegovu složenost.

5

Napisati algoritam koji pronalazi najmanji prirodni broj takav da se ne može predstaviti kao zbir nekih elemenata datog niza. Svaki element niza može najviše jednom da učestvuje kao sabirak. Odrediti složenost algoritma.