Softverski algoritmi u sistemima automatskog upravljanja

# Kompleksnost Algoritama

Danilo Kaćanski, Ivan Radman

# Šta je algoritam?

*Algoritam je konačan i precizan skup koraka za rešavanje nekog problema*

- Svaki algoritam ima:
  - **Ulaz** - početne podatke
  - **Pravila (korake)** - precizno opisane instrukcije
  - **Izlaz** - rezultat obrade


- Primer:
  - **Ulaz**: brojevi 5 i 3
  - **Algoritam**: Sabiranje
  - **Izlaz**: 8

Algoritam **mora imati kraj** - završava se u **konačnom broju koraka**

# Šta je kompleksnost algoritma?

*Kompleksnost algoritma pokazuje koliko vremena i memorije je potrebno da se neki problem reši*

### ⏱️ Vremenska složenost

- **Broj operacija** koje algoritam izvršava
- Zavisi od veličine ulaza
- Fokus u ovom kursu

### 💾 Memorijska složenost

- **Količina memorije** potreba za izvršavanje
- Naročito važna kod rekurzije, grafova, dinamičkog programiranja

- Koristimo **asimptotske notacije** da opišemo ponašanje algoritma kada $n \to \infty$:
  - **Big O**: Gornja granica (*najgori slučaj*)
  - **Omega (Ω)**: Donja granica (*najbolji slučaj*)
  - **Theta (Θ)**: Tačna granica (*prosečan sluča, ako je poznat*)

# Klase kompleksnosti algoritama

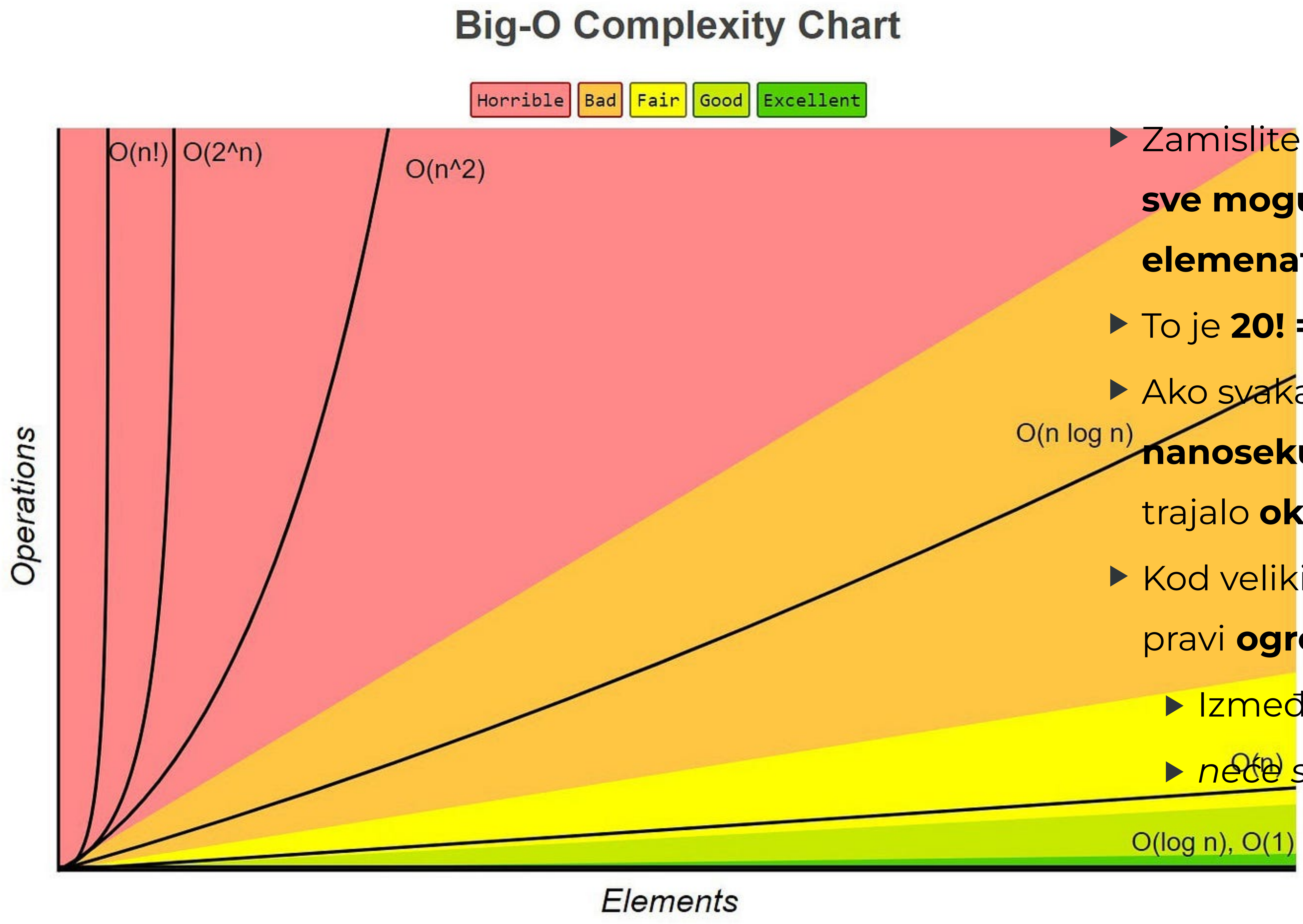
$$O(1) < O(\log n) < O(n) < O(n * \log n) <$$

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

- Mali ulazi? **Svi algoritmi rade prihvatljivo**
- Ali kada ulaz poraste, **razlike psotaju drastične**
- Na primer, za ulaz veličine **n=1000**:
  - ***O(n)*** *→ 1,000 koraka*
  - ***O(n²)*** *→ 1,000,000 koraka*
  - ***O(n!)*** *→ više koraka nego što ima atoma u svemiru*

*Zato analiziramo složenost **pre nego što napišemo i jednu liniju koda***

# Kako različite kompleksnosti rastu?




- Zamislite algoritam koji ispisuje **sve moguće permutacije 20 elemenata**
- To je **20! = 2.43 × $10^{18}$** kombinacija
- Ako svaka operacija traje **1 nanosekudnu**, izvršavanje bi trajalo **oko 77 godina**
- Kod velikih ulaza, izbor algoritma pravi **ogromnu razliku**:
  - Između *radi odmah* i
  - *neće se izvršiti ni za života*

# Konstantna vremenska složenost

$O(1)$ :

```python
def prvi_element(niz):
    return niz[0]
```

Operacija koja se **uvek izvršava u jednom koraku, bez obzira na veličinu ulaza.**

**Primeri:**

- Pristup elementu po **indeksu**
- Dodavanje/uklanjanje sa **stack/queue**
- Provera da li je broj **paran**

⚠️ **Napomena:**

- $O(1)$ ne znaci da je **trenunto**, već da se **ne povećava sa rastom *n***

# Logaritmska vremenska složenost

$$O(\log n):$$

```python
def deljenje_sa_dva(n):
    while n > 1:
        n = n // 2
```

Broj koraka ne raste linearno, već se povećava **mnogo sporije — logaritamski.**

**Primeri:**

- **Binarna pretraga**
- Algoritmi koji **redukuju veličinu problema na pola** u svakoj iteraciji

⚠️ **Napomena:**

- Tipicno kod **divide & conquer** pristupa

# Linearna vremenska složenost

$$O(n):$$

```python
def ispis(n):
    for i in range(n):
        print(i)
```

Broj izvršenih operacija raste **linearnom brzinom** u odnosu na veličinu ulaza. Ako se ulaz **udvostruči**, duplo više operacija se izvršava.

**Primeri:**

- **Linearna pretraga**
- Provera da li je niz **palindrom**
- **Sumiranje/iteracija** kroz listu

⚠️ **Napomena:**

- Često se javlja kod algoritama koji **moraju obraditi svaki element** bar jednom

# Linearno-logaritamska vremenska složenost

$$O(n * log\, n)\ :$$

```python
def primer_nlogn(n):
    for i in range(n):
        j = 1
        while j < n:
            j *= 2
```

Svaka od *n* iteracija spoljašnje petlje sadrži unutrašnju petlju koja se ponaša **logaritamski**. Ovo je tipična složenost kod mnogih **efikasnih algoritama sortiranja**.

**Primeri:**

- Merge sort
- Heap sort
- Quick sort

⚠️ **Napomena:**

- **Teorijski donja granica** efikasnosti za sve algoritme sortiranja koji porede elemente

# Kvadratna vremenska složenost

$O(n^2)$ :

```python
def kvadratna(n):
    for i in range(n):
        for j in range(n):
            print(i + j)
```

Broj operacija raste **kvadratno** sa brojem ulaza.

Nije pogodno za **veće ulaze**.

**Primeri:**

- Bubble sort
- Insertion sort
- Matrica x matrica

⚠️ **Napomena:**

- Efikasno **samo za male ulaze** ili u kombinaciji sa **heuristikama**

# Eksponencijalna vremenska složenost

$O(2^n)$ :

```python
def fib(n):
    if n ≤ 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Broj operacija rast **eksponencijalno** — svaki ulaz duplira posao.

**Primeri:**

- Naivan rekurzivni Fibonaci
- Hanojske kule
- Kombinatorni problemi (npr. svi podskupovi)

⚠️ **Napomena:**

- Potrebna optimizacija pomoću **memoizacija** ili dinamičkog programiranja

# Faktorska vremenska složenost

$$O(n!) :$$

```
import itertools

def permutacije(n):
    elementi = list(range(n))
    for p in itertools.permutations(elementi):
        print(p)
```

**Raste brže od eksponencijalnog** — broj kombinacija **eksplodira** sa porastom *n*.

**Primeri:**

- **Permutacije** svih elemenata
- **Putujući trgovac** (brute-force)
- **Raspoređivanje uloga** među n osoba

⚠️ **Napomena:**

- Praktično **neupotrebljivo bez heuristika ili ograničenja problema**

# Kako odrediti vremensku složenost algoritma?

1️⃣ **Izračunavanje složenosti svake naredbe**

🔍 Za svaku naredbu procenjujemo **broj izvršavanja** i **njenu složenost**

2️⃣ **Zadržavanje dominantnog člana**

📈 U izrazu zadržavamo samo **najveći rastući deo**, jer on **najviše utiče na ponašanje**

3️⃣ **Ignorisanje konstanti**

🚫 Konstantni faktori (poput *5*n* postaje *n*) **ne utiču na asimptotsko ponašanje**

# Kako odrediti vremensku složenost algoritma?

## Primer 1 — Odrediti slozenost sledeceg koda:

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

$c_1$
$c_1$
$c_1$
$c_1$
$c_1$

$$T(n) = \quad + \quad + \quad + \quad + \quad = 5 * c_1$$

🔢 Broj naredbi: $5 * c_1$

🧠 Kompleksnost algoritma: $O(1)$

# Kako odrediti vremensku složenost algoritma?

**Primer 2 — Odrediti slozenost sledeceg koda:**

```python
a = b + c

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

for j in range(n):
    j += 4
```

$c_1$

$c_2 * n$

$c_3 * n$

$T(n) = \quad + \quad +$

🔢 Broj naredbi: $c_1 + c_2 n + c_3 n$

🧠 Kompleksnost algoritma: $O(n)$

# Kako odrediti vremensku složenost algoritma?

**Primer 3 — Odrediti slozenost sledeceg koda:**

```
def if_else(n, uslov):
    if uslov:
        for i in range(n):
            print(i)
    else:
        for j in range(n):
            for k in range(n):
                print(j * k)
```

$c_1 * n$

$c_2 * n^2$

$T(n) = \quad +$

🔢Broj naredbi: $c_1 n + c_2 n^2$

🧠Kompleksnost algoritma: $O(n^2)$

# Odrediti vremensku složenost algoritma:

```
def mystery(n):
    for i in range(n):
        for j in range(n):
            if j * j > n:
                break
```

🧠Kompleksnost algoritma: $O(n \cdot \sqrt{n})$

```
def list_growth(n):
    a = []
    for i in range(n):
        a.append(i)
```

🧠Kompleksnost algoritma: $O(n)$

```
def log_nested(n):
    i = 1
    while i < n:
        j = n
        while j > 0:
            j //= 2
        i *= 2
```

🧠Kompleksnost algoritma: $O(n \cdot \sqrt{n})$

```
def count_pairs(n):
    count = 0
    for i in range(n):
        j = i
        while j < n:
            count += 1
            j += 1
    return count
```

🧠Kompleksnost algoritma: $O(n^2)$

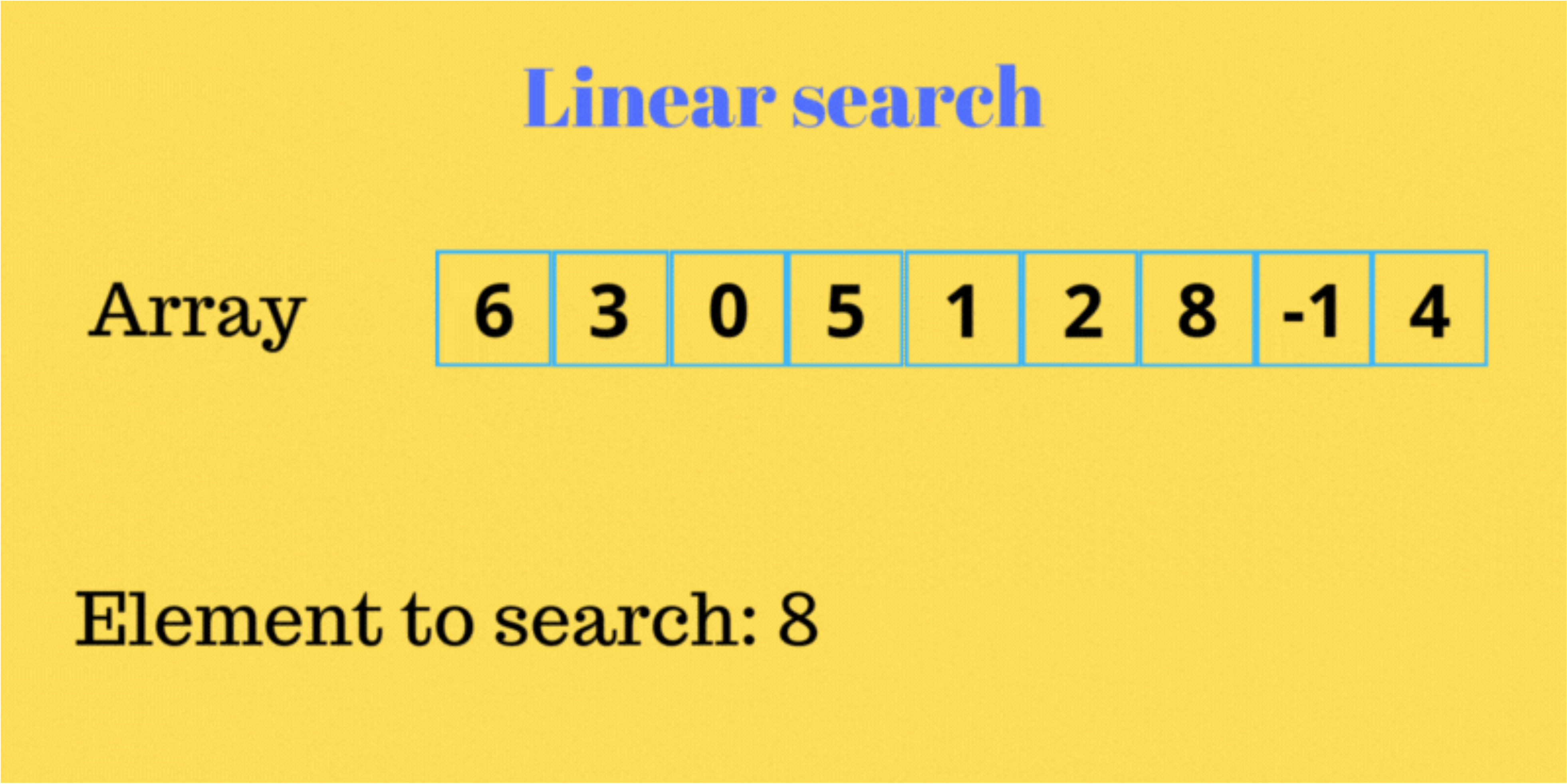
Softverski algoritmi u sistemima automatskog upravljanja

# Algoritmi pretrage

Danilo Kaćanski, Ivan Radman

# Linearna pretraga

- **Najjednostavniji** algoritam pretrage
- Prelazi se niz **od početka do kraja**
- Zastavljanje kada se:
  - Nađe element ili
  - Stigne do kraja
- **Bez posebnih zahteva** — radi i na nesortiranim nizovima




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

- Broj koraak u najgorem slučaju: $n$
- Kompleksnost: $O(n)$
- Jednostavno, ali **neefikasno za velike nizove**

# Stražar (Sentinel) optimizacija

- Umesto **2 provere** u svakoj interaciji:
  - Da li je `niz[i] == x`?
  - Da li je **kraj niza**?
- Sentinel **eliminiše proveru kraja** dodavanjem traženog elemnta na kraj
- **Samo jedna provera** u while petlji
- Takođe, ali brže u praksi
- **Priveremeno** menja poslednji element (vrati nazad!)

```python
def sentinelSearch(niz, x):
    n = len(niz)
    last = niz[n-1]
    niz[n-1] = x
    i = 0
    while niz[i] ! = x:
        i += 1
    niz[n-1] = last
    return i if i < n-1 or last = = x else -1
```

# Binarna pretraga

- Radi samo na **sortiranom nizu**!
- U svakom koraku:
  - Pronađe **srednji element**
  - **Uporedi** ga sa traženim
  - Ako je:
    - **Jednak** → Kraj!
    - **Manji** → Traži desno
    - **Veći** → Traži levo
- Efikasnije od linearne jer se **broj koraka smanjuje logaritamski** (*polovi se u svakom koraku*)
- Kompleksnost: $O(log\, n)$
- **Stabilno**, **brzo** i **često korišćeno u praksi**

Search for 47

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 7 | 10 | 14 | 23 | 45 | 47 | 53 |

```
def binarySearch(niz, x):
    low, high = 0, len(niz) - 1
    while low < = high:
        mid = (low + high) // 2
        if niz[mid] = = x:
            return mid
        elif niz[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

# Rekurzivna binarna pretraga

📌**Šta je rekurzija?**

- **Funkcija koja poziva samu sebe** za manji deo problema

🔁**Kod binarne pretrage:**

- Delimo niz na **manji deo** svaki put
- Funkcija se **sama zove** za novu polovinu niza
- Mora imati **bazni slučaj** (kada se ne poziva dalje)

```
def binarySearchRec(niz, x, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    if niz[mid] = = x:
        return mid
    elif niz[mid] < x:
        return binarySearchRec(niz, x, mid + 1, high)
    else:
        return binarySearchRec(niz, x, low, mid - 1)
```

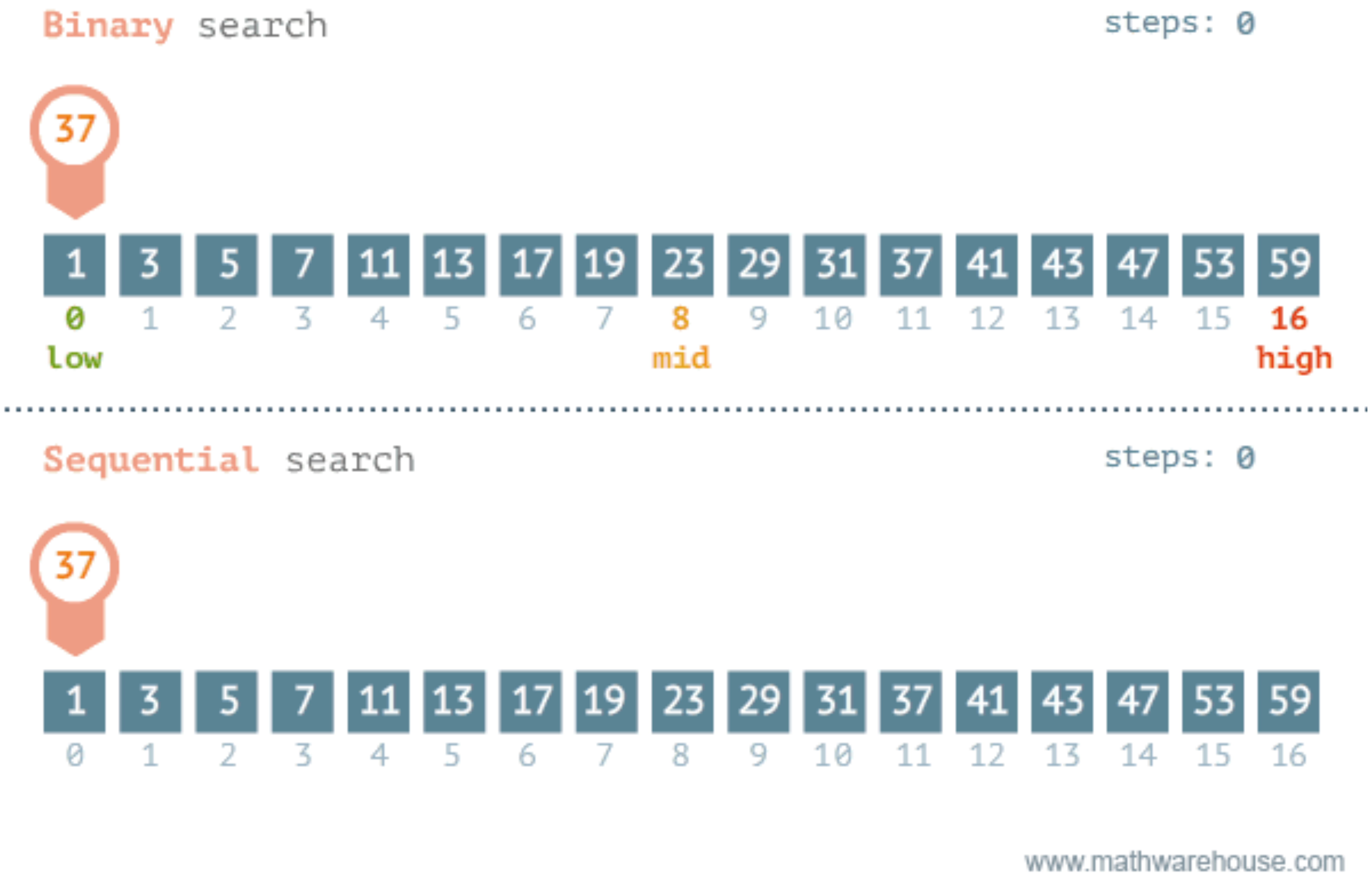
🧠**Prednosti:**

- Kratko, jasno
- Lako za **dokazivanje i razumevanje** (*idalno za teroiju*)

⚠️**Mane:**

- Troši **više memorije** (*novi poziv = novi stack frame*)

# Linearna vs Binarna pretraga



# Zadaci za vežbu — Presek dva niza (*349.*)

1. Data su ti dva niza celobrojnih vrednosti *nums1* i *nums2*. Potrebno je da vratiš **niz njihovih zajedničkih elemenata** (*presek*).Svaki element u rezultatu mora biti **jedinstven** (nema duplikata), a redosled elemenata **nije bitan**.

**Example 1:**

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2]
```

**Example 2:**

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [9,4]
Explanation: [4,9] is also accepted.
```

# Zadaci za vežbu — Pronađi poziciju (35.)

2. Dat ti je sortiran niz različitih celih brojeva i jedna ciljna vrednost (*target*). Vrati indeks te vrednosti **ako postoji** u nizu. Ako ne postoji, vrati indeks na kojem bi ta vrednost bila umetnuta kako bi niz ostao sortiran.

**Example 1:**

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

**Example 2:**

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

**Example 3:**

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

# Zadaci za vežbu — Koren (69.)

3. Dat ti je nenegativan ceo broj *x*. Vrati **kvadratni koren od *x***, **zaokružen nadole** na najbliži ceo broj. Rezultat mora takođe biti nenegativan ceo broj. Ne smeš koristiti ugrađene funkcije ili operatore za stepenovanje.

**Example 1:**

```
Input: x = 4
Output: 2
Explanation: The square root of 4 is 2, so we return 2.
```

**Example 2:**

```
Input: x = 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2
is returned.
```

# Zadaci za vežbu — Nedostajuć broj (268.)

4. Dat ti je niz *nums* koji sadrži *n* različitih celih brojeva iz opsega od *0* do *n*. Vrati jedini broj iz tog opsega koji nedostaje u nizu.

**Example 1:**

> **Input:** `nums = [3,0,1]`
>
> **Output:** 2
>
> **Explanation:**
>
> `n = 3` since there are 3 numbers, so all numbers are in the range `[0,3]`. 2 is the missing number in the range since it does not appear in `nums`.

**Example 2:**

> **Input:** `nums = [0,1]`
>
> **Output:** 2
>
> **Explanation:**
>
> `n = 2` since there are 2 numbers, so all numbers are in the range `[0,2]`. 2 is the missing number in the range since it does not appear in `nums`.

**Example 3:**

> **Input:** `nums = [9,6,4,2,3,5,7,0,1]`
>
> **Output:** 8
>
> **Explanation:**
>
> `n = 9` since there are 9 numbers, so all numbers are in the range `[0,9]`. 8 is the missing number in the range since it does not appear in `nums`.

# Zadaci za vežbu — Više ili manje (374.)

5. Igramo igru pogađanja broja. Pravila igre su sledeća:

   - Ja biram broj iz opsega od ***1* do *n***.
   - Tvoj zadatak je da pogodiš koji sam broj izabrao.

   Svaki put kada pogodiš pogrešan broj, reći ću ti da li je moj broj **veći** ili **manji** od tvog pokušaja.

   Imaš unapred definisanu funkciju *int guess(int num)*, koja vraća tri moguća rezultata:

   - *-1*: Tvoja pretpostavka je **veća** od broja koji sam izabrao. (tj. *num > pick*)
   - *1*: Tvoja pretpostavka je **manja** od broja koji sam izabrao. (tj. *num < pick*)
   - *0*: Pogodio si tačan broj. (tj. *num == pick*)

   Vratiti broj koji sam izabrao.

# Zadaci za vežbu — Više ili manje (374.)

**Example 1:**

> Input: n = 10, pick = 6
> Output: 6

**Example 2:**

> Input: n = 1, pick = 1
> Output: 1

**Example 3:**

> Input: n = 2, pick = 1
> Output: 1