# OSNOVE PROGRAMIRANJA U PAJTONU

## PREDAVANJE 11:
## MERENJE VREMENA I SLOŽENOST ALGORITMA

Miloš Kovačević

Đorđe Nedeljković

Marija Petronijević

Dušan Isailović

# SADRŽAJ PREDAVANJA

- Merenje vremena
- Dinamičko progrmiranje
- Funkcije vremenske složenosti
- Asimptotska notacija
- Linearno i binarno pretraživanje

# EFIKASNOST ALGORITMA I MERENJE VREMENA

Ključan zadatak: osmisliti efikasan algoritam za zadati problem.

Efikasnost = manji utrošak vremena i/ili memorijskih resursa.

Poređenje algoritama A i B zahteva merenje vremena izvršavanja za iste skupove ulaznih podataka.

Merenje vremena u računaru:
hardverski časovnici realnog i sistemskog vremena.

Realni časovnik: dan, mesec godina, sati, minuti i sekunde.

Realni časovnik radi i bez napajanja (baterija na matičnoj ploči).

# SISTEMSKO VREME

Sistemski sat: programabilni sat koji ažurira stanje
na svaku jedinicu vremena - tik (mili, mikro ili čak nano sekunda)

Po uključenju, inicijalizuje se sa sata realnog vremena i meri broj tikova od referentnog datuma u prošlosti – epohe.

Epoha je obično 1. januar 1970. u 00:00:00 po univerzalnom vremenu.

Univerzalno vreme: trenutno vreme koje važi u Griniču (Velika Britanija) – meridijan koji odvaja istočnu od zapadne zemljine polulopte.

Kako je sistemski sat precizniji u odnosu na realni,
sat realnog vremena povremeno se sinhronizuje
uz pomoć preračunatog sistemskog vremena.

# MODUL `time`

| funkcija | opis |
|---:|---|
| `gmtime()` | vraća objekat koji reprezentuje tekuće univerzalno vreme |
| `localtime()` | vraća objekat koji reprezentuje tekuće lokalno vreme |
| `asctime()` | vraća tekstualnu reprezentaciju vremenskog objekta |
| `sleep()` | pauzira izvršavanje programa za navedeni broj sekundi |
| `time()` | vraća broj proteklih sekundi od epohe koja se određuje sa `gmtime(0)` |
| `perf_counter()` | precizno meri trajanje pojedinih delova programa |

# MODUL `time` - PRIMER

```
>>> time.gmtime()   # univerzalno vreme
time.struct_time(tm_year=2017, tm_mon=4, tm_mday=2, tm_hour=17,
tm_min=22, tm_sec=27, tm_wday=6, tm_yday=92, tm_isdst=0)
>>> time.localtime() # lokalno vreme (Beograd)
time.struct_time(tm_year=2017, tm_mon=4, tm_mday=2, tm_hour=19,
tm_min=31, tm_sec=30, tm_wday=6, tm_yday=92, tm_isdst=1)
>>> time.asctime(time.localtime())  # tekstualni prikaz lok. vremena
'Sun Apr  2 19:31:58 2017'
>>> time.asctime(time.gmtime(0))  # tekstualni prikaz epohe
'Thu Jan  1 00:00:00 1970'
>>> time.time()  # broj proteklih sekundi od epohe
1491154398.6359384
>>> time.sleep(1)  # pauzira sa radom jednu sekundu
>>> time.time(), time.sleep(1), time.time()   # primer za sleep
(1491155266.9192047, None, 1491155267.9215555)
```

**Problem 7.1 — Fibonačijevi brojevi - rekurzija sa pamćenjem.** Rekurentna formula za $n$-ti član fibonačijevog niza je $F(n) = F(n-1) + F(n-2)$. Porediti trajanje izračunavanja $F(n)$, u verzijama bez i sa pamćenjem prethodno izračunatih članova. ■

```
# Poređenje dve rekurzivne verzije za računanje Fib(n)
import time
# klasičan rekurzivni algoritam
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Ideja pamćenja međurezultata:
prilikom izračunavanja fib(k) proverava se da li je ranije već izračunato!

Na primer: f(4) = f(3) + f(2), nakon ovoga f(2), f(3) i f(4) su izračunati.

f(5) = f(4) + f(3), f(4) i f(3) nema potrebe izračunavati ponovo!

# FIBONAČI SA PAMĆENJEM – DINAMIČKO PROGRAMIRANJE

Za potrebe pamćenja koristimo rečnik – ključ je k, a vrednost fib(k).

```
# rekurzivni algoritam sa pamćenjem
memorija = {}
def fib_memo(n):
    if n == 0 or n == 1:
        return 1
    elif n in memorija:
        return memorija[n]
    else:
        memorija[n] = fib_memo(n-1) + fib_memo(n-2)
        return memorija[n]
```

Dinamičko programiranje – razbijanje na potprobleme koji se preklapaju, pa ima smisla pamtiti međurezultate!

# MERENJE SISTEMSKOG VREMENA IZMEĐU DVA DOGAĐAJA

```
n = int(input('unesite n '))
# bez pamćenja
start = time.perf_counter()
f = fib(n)
t1 = time.perf_counter() - start
print('F({}) = {} za {:6.3e}'.format(n, f, t1))

# sa pamćenjem
start = time.perf_counter()
f = fib_memo(n)
t2 = time.perf_counter() - start
print('Fmemo({}) = {} za {:6.3e}'.format(n, f, t2))

print('Pamćenje skraćuje trajanje {:6.3f} puta!'.format(t1/t2))
```

Prilikom merenja treba obezbediti iste uslove (deterministički alg.: isti ulazi)

# VREMENSKO-PROSTORNA NAGODBA

```
unesite n 10
F(10) = 89 za 3.674e-05
Fmemo(10) = 89 za 1.067e-05
Pamćenje skraćuje trajanje  3.444 puta!
>>>
==== RESTART: C:\programi\p7_1.py ====
unesite n 30
F(30) = 1346269 za 4.098e-01
Fmemo(30) = 1346269 za 2.212e-05
Pamćenje skraćuje trajanje 18523.732 puta!
```

Vremensko-prostoran nagodba: ubrzanje se postiže na račun dodatne memorije

# EMPIRIJSKA FUNKCIJA VREMENSKE SLOŽENOSTI ALGORITMA

Empirijska fja vremenske složenosti formira se iz niza parova (veličina problema <ulaza>, trajanje ).

Primer:

```
# Poređenje dva algoritma za sumiranje prvih n prir. brojeva
import time
# Algoritam A
def sumaA(n):
    return n * (n + 1) // 2

# Algoritam B
def sumaB(n):
    suma = 0
    for i in range(1,n+1):
        suma += i
    return suma
```

Linijski algoritam

Iterativni algoritam

# EMPIRIJSKA FUNKCIJA VREMENSKE SLOŽENOSTI ALGORITMA

```
# A
for n in [100, 1000, 10000, 100000, 1000000]:
    start = time.perf_counter()
    s = sumaA(n)
    t = time.perf_counter() - start
    print('A: n={:9} s={:13} za {:6.3e}'.format(n, s, t))
# B
for n in [100, 1000, 10000, 100000, 1000000]:
    start = time.perf_counter()
    s = sumaB(n)
    t = time.perf_counter() - start
    print('B: n={:9} s={:13} za {:6.3e}'.format(n, s, t))
```

# EMPIRIJSKA FUNKCIJA VREMENSKE SLOŽENOSTI ALGORITMA

```
A: n=      100 s=          5050 za 8.691e-06
A: n=     1000 s=        500500 za 8.296e-06
A: n=    10000 s=      50005000 za 8.296e-06
A: n=   100000 s=    5000050000 za 7.901e-06
A: n=  1000000 s= 500000500000 za 8.296e-06
B: n=      100 s=          5050 za 3.635e-05
B: n=     1000 s=        500500 za 1.525e-04
B: n=    10000 s=      50005000 za 1.467e-03
B: n=   100000 s=    5000050000 za 1.458e-02
B: n=  1000000 s= 500000500000 za 6.069e-02
```

Trajanje A se ne menja sa porastom n, za B raste linearno sa porastom n.

# ANALTIČKA FUNKCIJA VREMENSKE SLOŽENOSTI ALGORITMA

Funkcija vremenske složenosti: analitička veza
između veličine problema (ulaza) i trajanja izvršavanja.

Pretpostavka: trajanje aritmetičko logičkih operacija nezavisno od veličine argumenata, a čitanja i pisanja nezavisno od mem. Lokacije.

```
3  # Algoritam A
4  def sumaA(n):
5      return n * (n + 1) // 2
6
7  # Algoritam B
8  def sumaB(n):
9      suma = 0
10     for i in range(1,n+1):
11         suma += i
12     return suma
```

Neka izračunavanje u r5 traje c jedinica vremena, dodela u r9 c1, izvršavanje petlje u r10 c2, sabiranje u r11 c3, a r12 c4.

Sada je:

$$t_A(n) = c$$

$$t_B(n) = c_1 + n(c_2 + c_3) + c_4 = an + b$$

$$t_B(n) > t_A(n), \text{ za } n > \frac{c-b}{a}$$

# ANALTIČKA FUNKCIJA VREMENSKE SLOŽENOSTI ALGORITMA

Funkciju složenosti možemo posmatrati u najpovoljnijem, prosečnom i najnepovoljnijem slučaju izvršavanja – primer pretrage broja u listi.

$$t_A(n) = c \qquad t_B(n) = c_1 + n(c_2 + c_3) + c_4 = an + b$$

Konstante a, b i c zavise od hardvera na kom se program izvršava.

Treba nam univerzalan pristup poređenja, nezavisan od hardvera:

*t(n)* treba izraziti preko dominantnih operacija (sabiranje kod sumiranja)

*t(n)* treba posmatrati u graničnom slučaju kad *n* teži beskonačno

# FUNKCIJA VREMENSKE SLOŽENOSTI – UNIVERZALNI PRISTUP

najnepovoljniji slučaj, $t_A(n) = an^2 + bn + c$, na *brzom* računaru

$t_B(n) = dn + e$, na *sporom* računaru

*t(n)* treba posmatrati u graničnom slučaju kad *n* teži beskonačno:

$$\lim_{n \to \infty} \frac{t_B(n)}{t_A(n)} = \frac{dn + e}{an^2 + bn + c} = 0$$

! Nezavisno od činjenice da su računari sve brži, pametan programer, koji će osmisliti efikasan algoritam i dalje je na ceni! Dobar algoritam i spor računar, *bolji* su tim od lošeg algoritma i brzog računara.

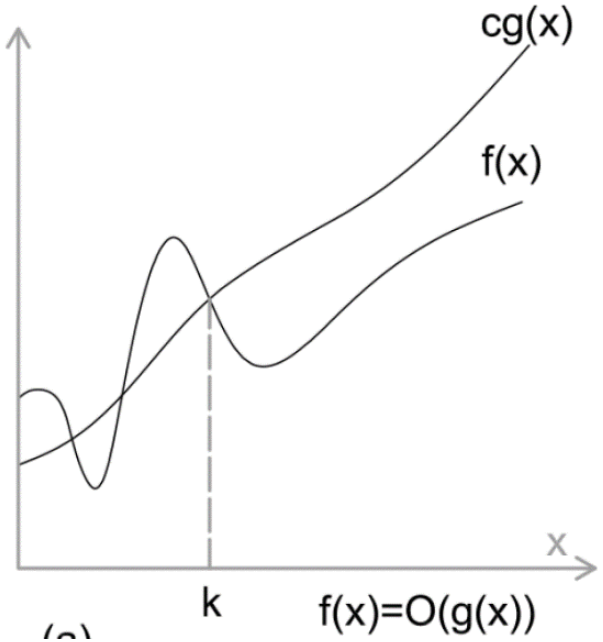
# ASIMPTOTSKA NOTACIJA O(n)

Posmatra se funkcija složenosti $f : \mathbb{R}^+ \to \mathbb{R}^+$

*f* je reda složenosti O(g), kaže se i pripada klasi funkcija O(g), ili je O(g), ako postoje pozitivne realne konstante c i *k*, takve da važi:

$$\forall x > k, \quad f(x) \leq cg(x)$$

Ako je *f* reda složenosti O(g),
očigledno je ograničena odozgo sa cg(x)
odnosno ne raste brže od cg(x)



# OSOBINE ASIMPTOTSKE NOTACIJE O(n)

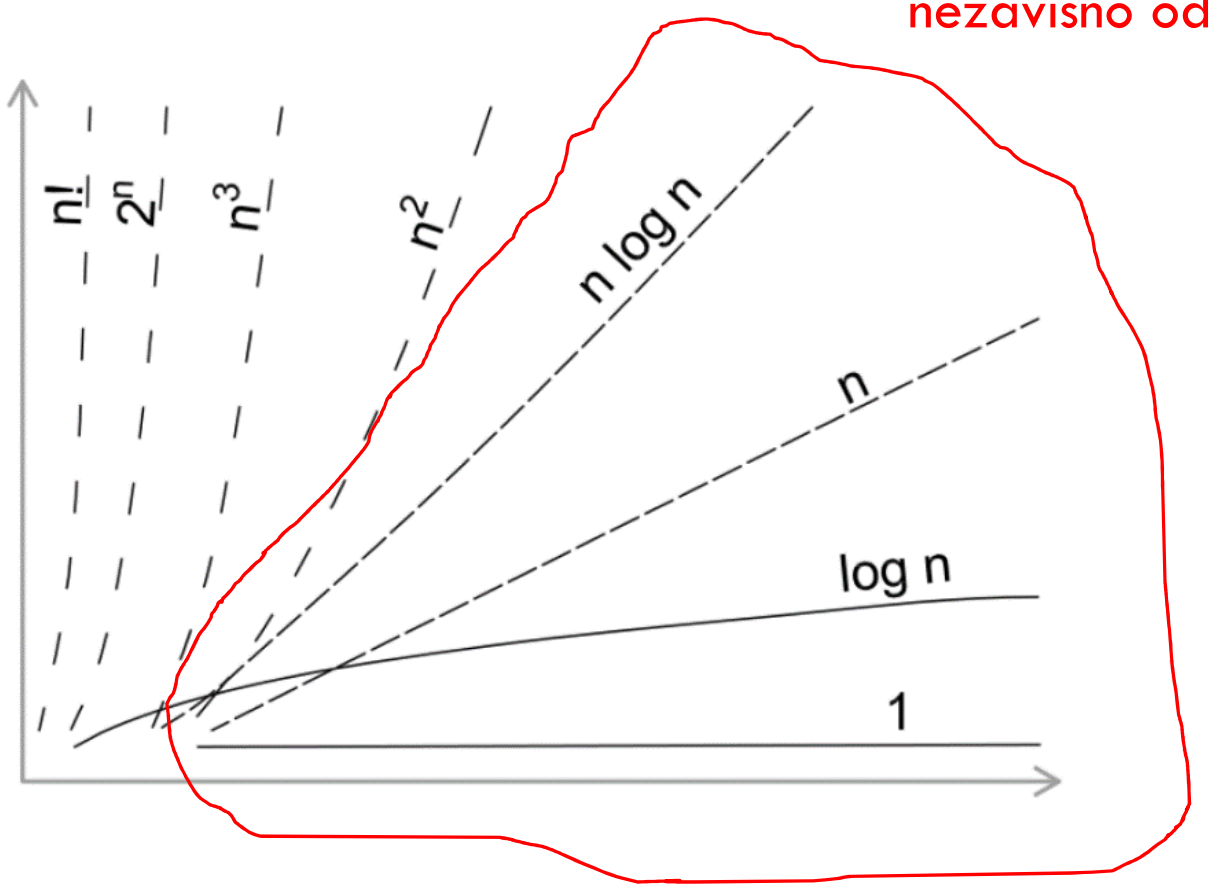
- $f$ je $O(f)$.
- ako je $f$ u $O(g)$ i $g$ u $O(f)$, onda su $f$ i $g$ istog *reda složenosti*.
- ako je $f$ u $O(g)$ i $\forall x > 0,\ \ h(x) \geq g(x)$, onda važi i $f$ u $O(h)$.

Očigledno, ako je *f* reda O(n), onda je i $O(n^2)$,
ali se u analizi najnepovoljnijeg slučaja gleda što jednostavnija funkcija

Prilikom računanja asimptotskog reda,
treba zanemariti sve članove nižeg reda i sve konstante

# REDOVI SLOŽENOSTI

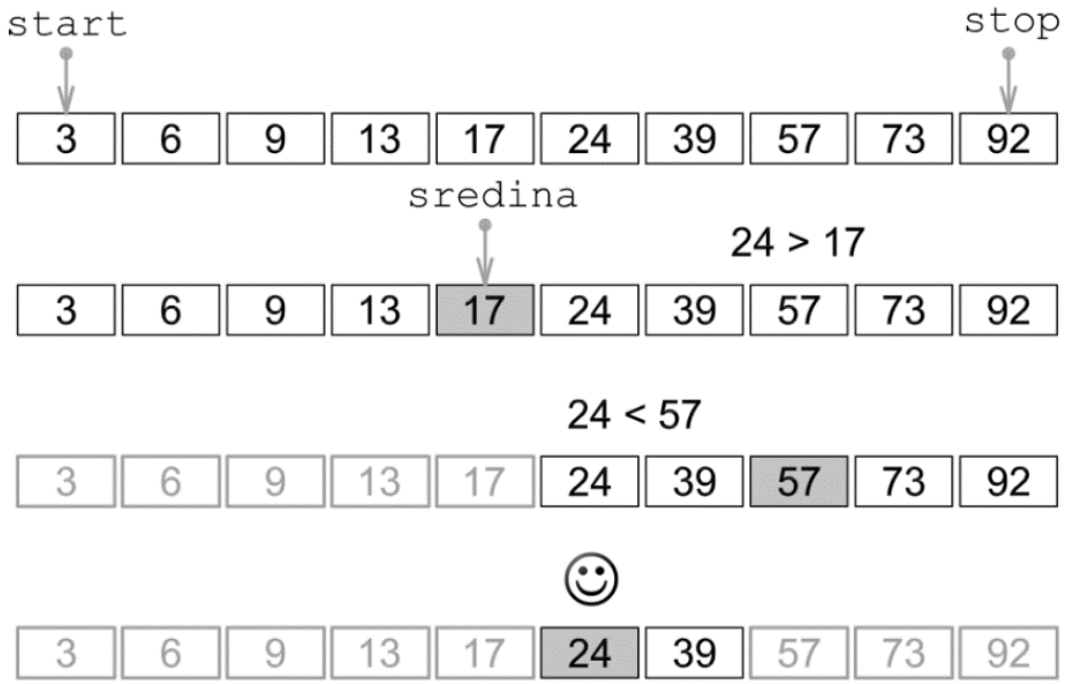
Efikasni algoritmi,
nezavisno od hardvera



# PRETRAŽIVANJE SORTIRANE LISTE – BINARNA PRETRAGA

Posmatra se lista brojeva L koji su sortirani od najmanjeg ka najvećem.

Algoritam binarne pretrage pripada klasi podeli pa vladaj.
Zasniva se na pametnoj redukciji prostora svih mogućih rešenja
(svaki put na polovinu prethodnog skupa!)

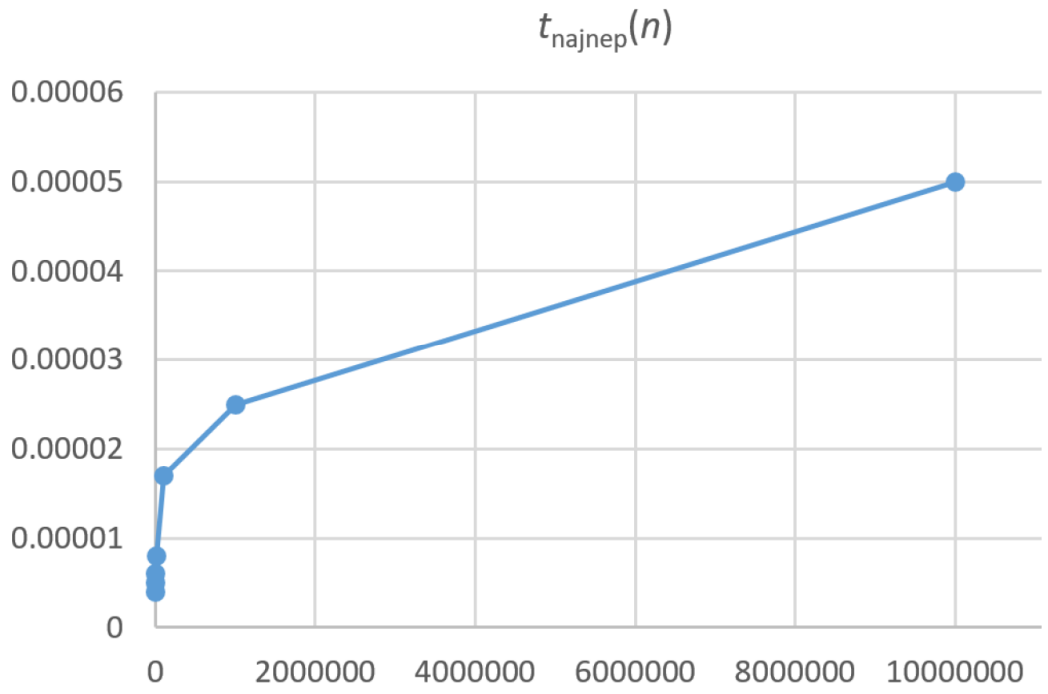


# PRETRAŽIVANJE SORTIRANE LISTE – BINARNA PRETRAGA

```
# Binarno pretraživanje
def bin_pretraga(L, x):
    '''Pronalazi indeks od x, u L. Vraća None ako x nije u L'''
    start, stop = 0, len(L)-1
    while start <= stop:
        sredina = (start + stop) // 2
        if x < L[sredina]:
            stop = sredina - 1
        elif x > L[sredina]:
            start = sredina +1
        else:
            return sredina
    return None
```

# EMPIRIJSKA FUNKCIJA SLOŽENOSTI

```
# testiranje za najnepovoljniji slučaj
import time as t
print('{:8}\t{}'.format('n', 'Tnajnep'))
for i in range(1, 8):
    n = 10**i
    L = [k for k in range(n)]
    t_start = t.perf_counter()
    bin_pretraga(L, n)
    t_najnep = t.perf_counter() - t_start
    print('{:<8d}\t{:<8.6f}'.format(n, t_najnep))
```



# BINARNA PRETRAGA – ASIMPTOTSKA KOMPLEKSNOST

```
while start <= stop:
    sredina = (start + stop) // 2
    if x < L[sredina]:
        stop = sredina - 1
    elif x > L[sredina]:
        start = sredina +1
    else:
        return sredina
return None
```

Neka L ima n = $2^k$ elemenata

Posmatrajmo poređenje kao dominantnu operaciju.

Koliko ima poređenja kada se element ne nalazi u listi (najnepovoljniji slučaj)?

$$t_{najnep}(n) = 3(k+1)+1$$

$$t_{najnep}(n) = 3\log_2 n + 4$$

Binarna pretraga $O(\log_2 n)$