ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Сложеност на алгоритми

АЛГОРИТМИ И ПОДАТОЧНИ СТРУКТУРИ

- предавања -

Алгоритми и податочни структури

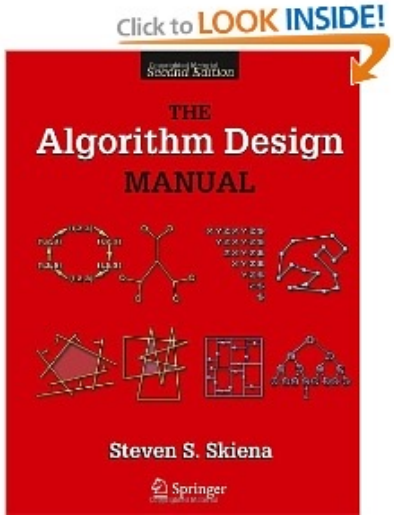
- **Професори:**
  - проф. д-р Ана Мадевска Богданова
  - проф. д-р Владимир Трајковиќ
  - проф. д-р Слободан Калајџиски
  - вон. проф. д-р Магдалена Костоска
  - вон. проф. д-р Христина Михајлоска
  - вон. проф. д-р Бојана Котеска
  - вон. проф. д-р Ефтим Здравевски
  - доц. д-р. Илинка Иваноска

- **Асистенти:**
  - асс. Александар Тенев
  - асс. Ненад Анчев
  - асс. Славе Темков
  - асс. Јана Кузманова
  - дем. Мартин Динев

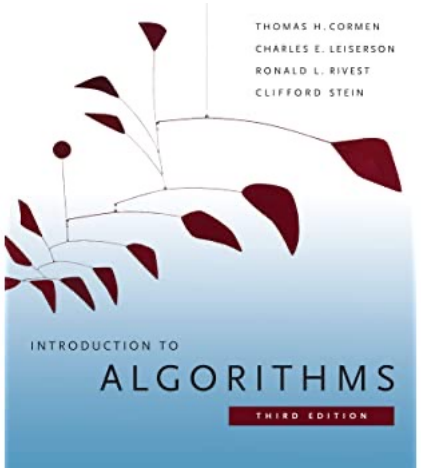
Содржина

- Сложеност на алгоритми
- Низи и динамички листи
- Архетипи на алгоритми - општо
- Архетипи на алгоритми - продолжение
- Еднодимензионални податочни структури (редици, магацини), приоритетни листи
- Сортирање
- Хеш функции
- Стебла (општи, бинарни)
- Стебла (АВЛ, останати)
- Графови вовед
- Графови манипулација

Литература

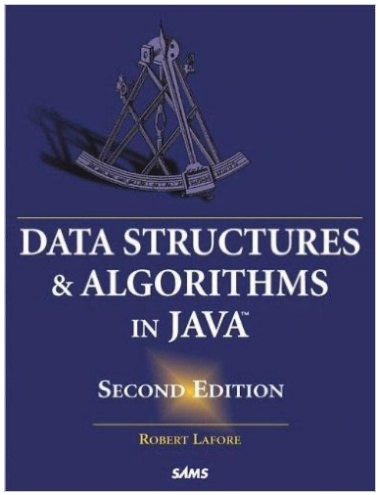


- The Algorithm Design Manual
  - Steven S. Skiena
  - 2008
  - http://www3.cs.stonybrook.edu/~skiena/373/videos/

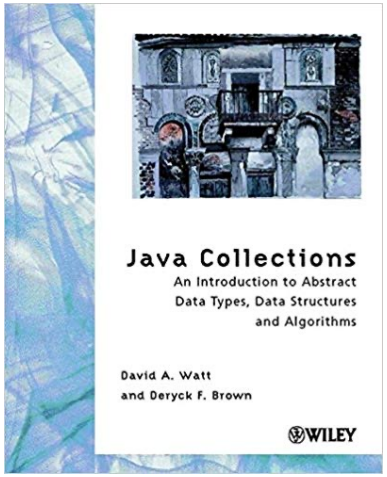


- Introduction to Algorithms, 3rd Edition (The MIT Press)
  - Cormen, Leiserson, Rivest, Stein
  - 2009

Литература



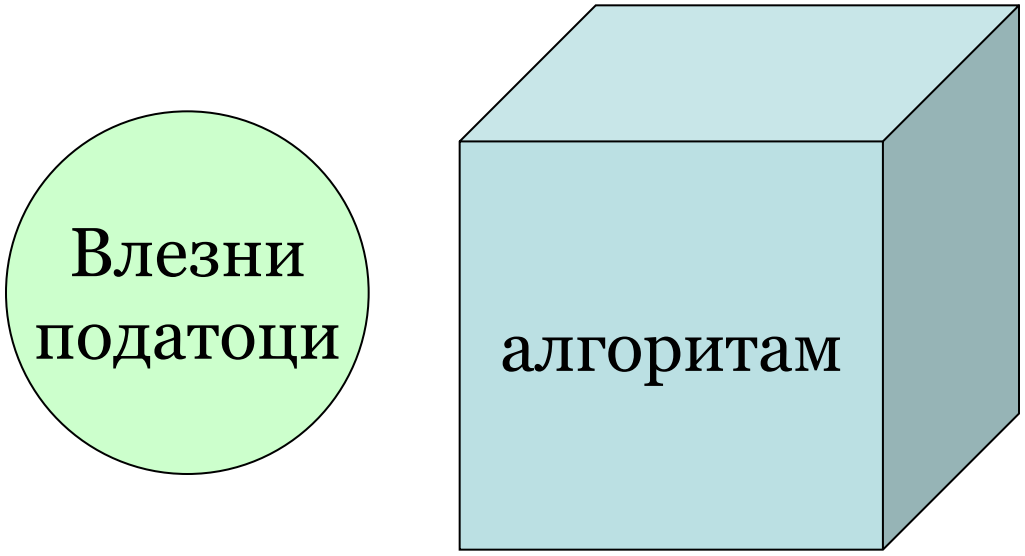
- Data Structures & Algorithms in Java, 2nd edition
  - Robert Lafore
  - 2002



- *Java Collections*
  - David Watt and Deryck Brown
  - 2001

Алгоритми

- Пред да се започне со пишување на програмскиот код на некој проблем што се решава, треба да се смисли решението на проблемот - **алгоритмот**



Алгоритми

- Алгоритам
  - Процедура за решавање даден проблем
  - се одвива чекор-по-чекор
  - Конечен број чекори
- Нотации за опис на алгоритмите
  - Говорен јазик
  - Псевдојазик
  - Програмски јазик

Какви се алгоритмите?

- Алгоритмите треба да бидат:
  - Коректни
  - Ефикасни
  - Лесни за имплементација

Податочни структури

- Информациите кои треба да бидат процесирани од страна на програмата се чуваат во **податочни структури - *data structures***
  - Од изборот на податочните структури зависи ефикасноста на алгоритмите

Пример:
наоѓање телефонски број за позната адреса и наоѓање на адреса за познат телефонски број

- Податочните структури може да бидат
  - Статични
    - промени само на вредностите
    - низи (*arrays*) или записи (*records*))
  - Динамички
    - промени и во изгледот, големината
    - магацини (*stacks*), листи (*lists*), дрва (*trees*), датотеки (*files*)).

Перформанси на алгоритмите

- Акции кои се изведуваат врз податочните структури:
  - Пребарување и пронаоѓање
  - Пребројување
  - Вметнување
  - Сортирање
  - Бришење
- За да се подобрат перформансите на алгоритмот треба да се напише добар, а потоа по можност и подобар алгоритам (**оптимизација**)

Како се споредуваат алгоритмите?

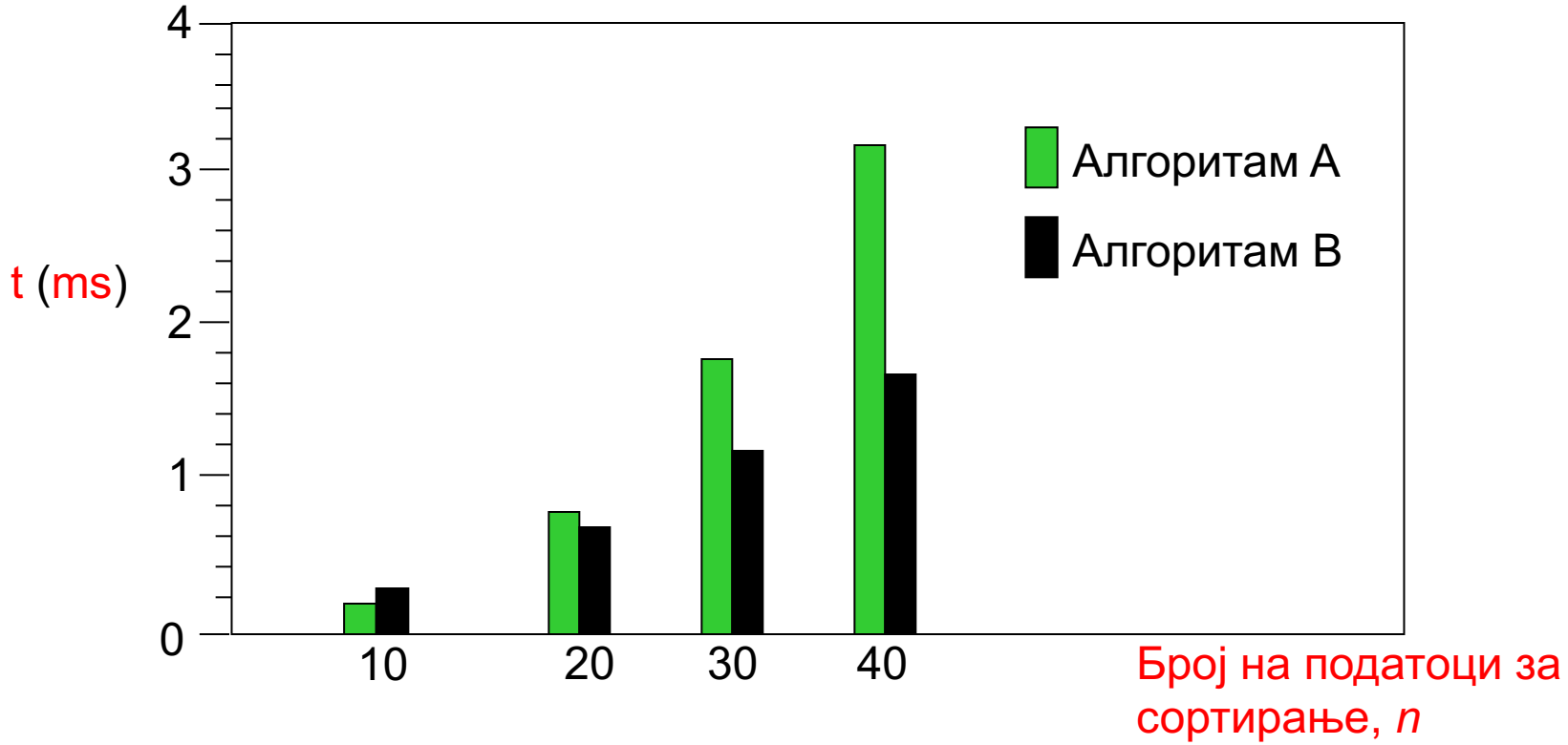
- Еден проблем може да се реши на неколку начини
  - Различни алгоритми

НО,

- Како да се процени нивната ефикасност?
- Како да се споредат?

Пример

- Споредба на два алгоритми за сортирање:



- Алгоритамот B е побрз од алгоритмот A.

Еве како се споредуваат алгоритми!

- Споредба на алгоритми се прави преку модел кој е независен од хардверот
- Хипотетички компјутер - Random Access Machine

Перформанси на алгоритмите

- Random Access Machine
  - Секоја едноставна операција (+, *, -, =, if, call) **се извршува во една временска единица**
  - Циклусите (јамките), процедурите и функциите се извршуваат во онолку временски единици колку што содржат итерации
  - Постои неограничена меморија

Броење чекори

```
Suma( x, n)
{
    int sum = 0;
    for (int i=0; i<=n; ++i)
        sum = sum*x+1;
    return sum;
}
```

Број на извршени операции

1	
1+(n+2)+(n+1)=2n+4	
(n+1)(1+1+1)=3n+3	
1	

Вкупно: T(n) = 5n+9

$$\sum_{i=0}^{n} x^i$$

T(n) е функција што го дава бројот на извршени операции во зависност бројот на влезните податоци

А споредувањето?

- Да разгледаме 2 алгоритми, *A* и *B*, коишто решаваат ист проблем.
- Сме направиле анализа за потребниот број чекори за секој од алгоритмите
  - $T_A(n)$ и $T_B(n)$
  - n е мерка за големината на проблемот
    - n е број на влезни податоци
- Значи останува **само** да се споредат двете функции $T_A(n)$ и $T_B(n)$ и ќе одредиме кој е победникот!

НО!

- Може да провериме за некоја конечна вредност за n (n_0) кој алгоритам е побрз (помал број на операции)
- Во општ случај не знаеме колку може да биде n.
  - 10, 100, ..100 000…1000 000 ??
- Може да се покаже дека **ако** $T_A(n) <= T_B(n)$ **за сите** n>=0,
  - Алгоритамот А е побрз, подобар!
- СЕПАК,
  - Не го знаеме n однапред !!
- ЗАТОА – ќе го разгледаме поставување горна асимптотска граница за многу големи проблемски величини
  - За големо n

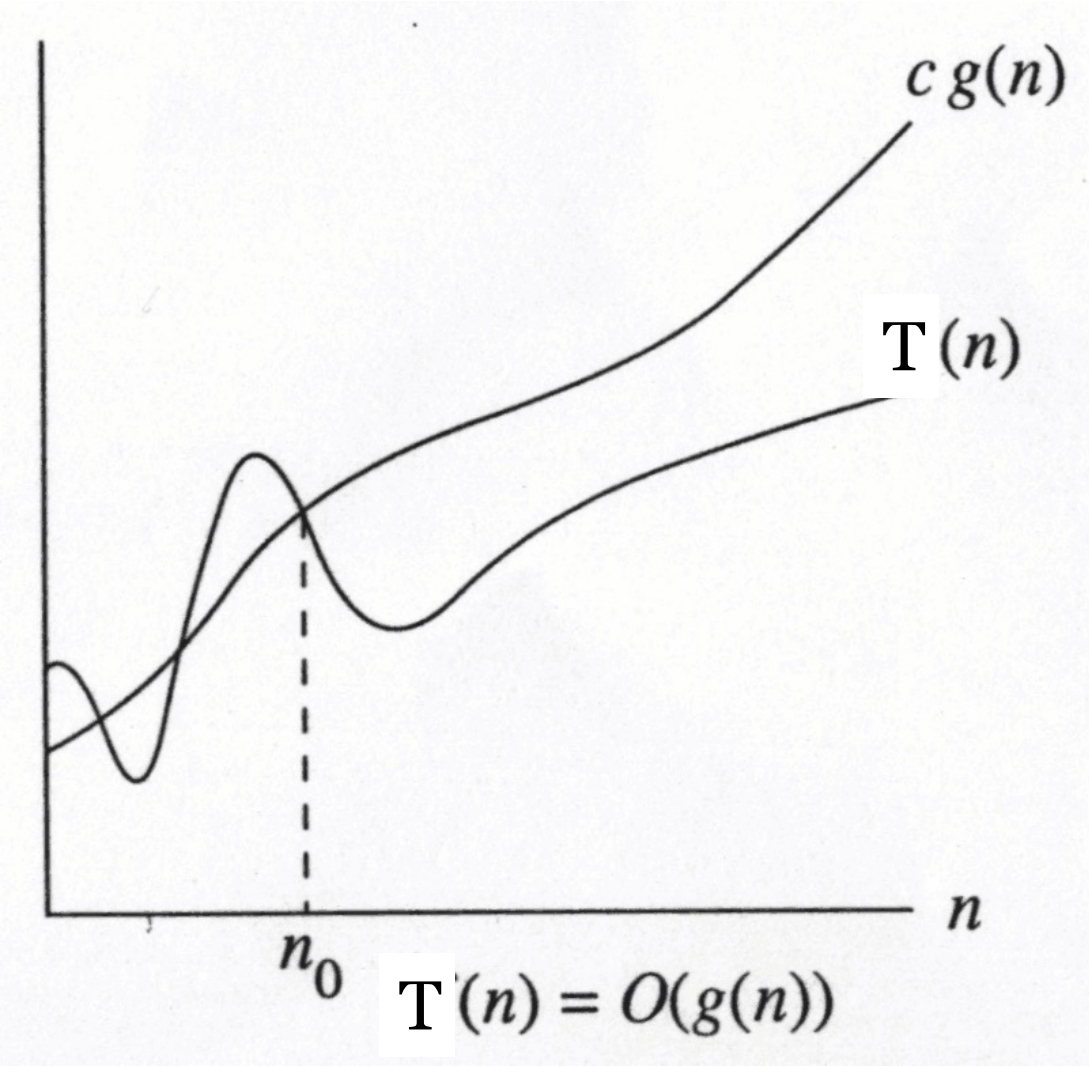
Исто така...

- Пресметувањето на T(n) е исцрпувачко за алгоритми со голем број инструкции и операции
- Не е потребно да се знае точниот број операции
  - за да се одреди сложеноста на даден алгоритам
  - за да се одреди кој алгоритам е поефикасен (побрз)
- Ни треба оценувач (функција) што ќе биде горна асимптотска граница на T(n)

Дефиниција на “Големо О”

- Најчесто користениот метод и нотација за проценка на сложеност на алгоритам е *Големо О*
- *Големо* О е асимптотско време на извршување (број на операции) на даден алгоритам
- Ни треба одговор на прашањето: *Како расте времето на извршување на алгоритмот како функција од бројот на влезни податоци?*
- *Големо О* е горна граница
- Математичка алатка
- Крие голем број непотребни детали со придружување на едноставна проценка (функција) кон даден алгоритам

Дефиниција на “Големо О”



Формална дефиниција на ”Големо О”

- T(n) е O(g(n)) ако постојат позитивни константи c и n_0 такви што $T(n) \leq c\, g(n)$ за $n \geq n_0$
  - n е бројот на влезните податоци
  - T(n) е функција која го опишува вистинското време на извршување на алгоритамот (број на извршени операции – една операција се изведува во една временска единица)
  - g(n) е функција што ја карактеризира горната граница на T(n) (асимптотска граница на T(n))
    - најблиску е до T(n), а секогаш е над неа, за било кое n, почнувајќи од некое n_0
    - гарантирано е дека T(n) не може да биде поголемо од g(n) за било кое $n \geq n_0$

Како се одредува O(g(n))?

- Може уште повеќе да се олесни проценувањето за сложеноста на алгоритмите

- O(g(n)) - се зема САМО членот со најголемиот степен на асимптотската функција g(n)

Правила за користење на ознаката O(..)

- Се отстрануваат сите терми, освен термот (членот) со највисок степен
  - Пример
$$O(n^2 + n\log n + n) \rightarrow O(n^2)$$
  - g(n)= $n^2 + n\ logn + n$

- Се отстрануваат константите
  - Пример
$$O(3n^2) \rightarrow O(n^2)$$
$$O(1024) \rightarrow O(1)$$

Споредба на алгоритми

- Според RAM моделот имаме:
  - Најдобро време на извршување
  - Средно време на извршување
  - Најлошо време на извршување на алгоритмот
    - Големо О

Пример:
Да се напише програмски код во псевдо јазик за реализација на сумата $\sum_{i=1}^{n} i^3$

Споредба на алгоритми

```
sum (n)
{
    partial_sum = 0;
    for (i=1; i<=n; i++)
        partial_sum += i*i*i;
    return partial_sum;
}
```

$$T(n) = 6n + 4$$

1

1 + n+1 + n = 2n+2

n+n+n+n = 4n

1

$O(n)$

- Декларациите не влијаат на времето на извршување

Се нарекува **КОМПЛЕКСНОСТ** на алгоритамот

Правила при споредба на алгоритми

- Циклуси:
  - Сума на времињата потребни да се извршат операциите во циклусот помножено со бројот на итерации

```
for (int i=0; i<n; i++)
{
    a += i;
    b *=i;
}
```

2n

$O(n)$

Правила при споредба на алгоритми

- Вгнездени циклуси:
  - Сума на времињата потребни да се извршат операциите во циклусот помножено со производот на бројот на итерации на циклусите

```
for (i=0; i<n; i++)
{
    for (j=0; j<n; j++)
        b *=i;
}
```

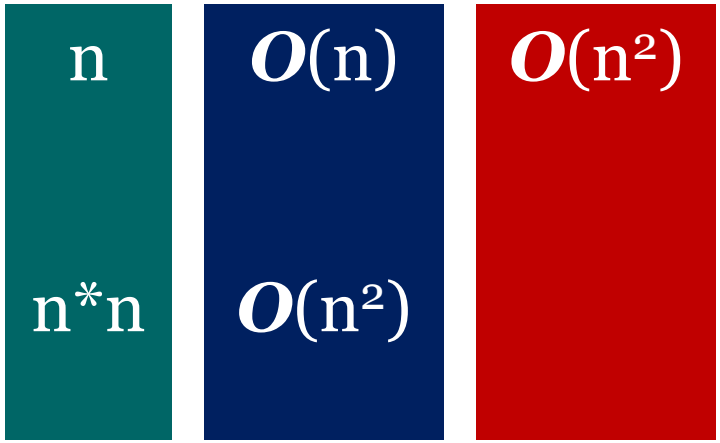
n*n

$O(n^2)$

Правила при споредба на алгоритми

- Последователни времиња:
  - Кај последователни извршувања се зема најголемата кардиналност од блоковите на последователни кодови

```
for (i=0; i<n; i++)
    a[i] = 0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i] += a[j]+i+j;
```



- Комплексноста на алгоритамот е $O(n^2)$

Правила при споредба на алгоритми

- IF / ELSE:
  - Збир од времетраењето на условот и подолгото време од времетраењата на блоковите S1 и S2

```
if (condition)
    S1
else
    S2
```

Правила при споредба на алгоритми

- Рекурзии:
  - Комплексноста на рекурзијата се разгледува од случај до случај и нема генерален шаблон за нејзиното времетраење

```
factorial (n)
{
    if (n<=1)
        return 1;
    else
        return (n*factorial(n-1));
}
```

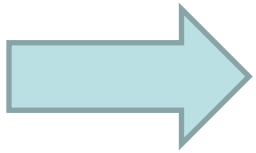
Правила при споредба на алгоритми

- Рекурзии:

```
fib (n)
{
    if (n<=1)
        return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```

T(n) – време на извршување на fib(n)

T(n) = T(n-1) + T(n-2) + 2



$O(2^n)$

Принципи на споредба

Низ следниот пример, решен со 3 различни алгоритми ќе се прикаже **практичниот начин** на проценка на нивната сложеност

- Пример:
  - Да се најде најголемата позитивна подсума во низа од n цели броеви
  - Влез: -2, 11, -4, 13, -5, -2

20

Примерот решен со три вгнездени циклуси

```
max_subsequence_sum( a[],  n )
{

    max_sum = 0; best_i = best_j = -1;
    for( i=0; i<n; i++ )                    → n
        for( j=i; j<n; j++ )                → n-i+1
        {
            this_sum=0;
            for( k = i; k<=j; k++ )         → j-i+1
                this_sum += a[k];
            if( this_sum > max_sum )
            {
                /* update max_sum, best_i, best_j */
                max_sum = this_sum;
                best_i = i;
                best_j = j;
            }
        }
        return max_sum;
}
```

Комплексноста на алгоритмот е $O(n^3)$

Примерот решен со два вгнездени циклуса

```
max_subsequence_sum( a[],  n )
{

    max_sum = 0; best_i = best_j = -1;
    for( i=0; i<n; i++ )
    {
        this_sum=0;
        for( j = i; j<=n; j++ )
        {
                this_sum += a[j];
                if( this_sum > max_sum )
                {
                    /* update max_sum, best_i, best_j */
                    max_sum = this_sum;
                    best_i = i;
                    best_j = j;
                }
        }
        return max_sum;
}
```

Комплексноста на алгоритмот е $O(n^2)$

Примерот решен со еден циклус

```
max_subsequence_sum( a[],  n )
{
    i = this_sum = max_sum = 0; best_i = best_j = -1;
    for( j=0; j<n; j++ )
    {
        this_sum += a[j];
        if( this_sum > max_sum )
        {
            /* update max_sum, best_i, best_j */
            max_sum = this_sum;
            best_i = i;
            best_j = j;
        }
        else
        if( this_sum < 0 )
        {
            i = j + 1;
            this_sum = 0;
        }
    }
    return max_sum;
}
```

Комплексноста на алгоритмот е $O(n)$

А што кога...

- ...податоците постојано се делат со 2?

Комплексноста се опишува со **логаритамска функција**!

```
foo (n) {
        // n > 0
        total = 0;
        while (n > 1) {
                n = n / 2;
            total++;
        }
        return total;
    }
```

Комплексноста на алгоритмот е $O(\log n)$

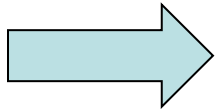
$\log_2 8 = ?$

Одговор: 3

Назад кон примерот

```
max_sub_sum( a[], left, right )
{
if ( left == right ) /* Base Case */
        if( a[left] > 0 ) return a[left];
        else return 0;
    center = (left + right )/2;
    max_left_sum = max_sub_sum( a, left, center );
    max_right_sum = max_sub_sum( a, center+1, right );
    max_left_border_sum = 0; left_border_sum = 0;
```

```
int  max_sub_seq_sum( int a[],
unsigned int n )
{
        return max_sub_sum( a, 0, n-
}
```



... продолжение

```
    for( i=center; i>=left; i-- )
    {
        left_border_sum += a[i];
        if( left_border_sum > max_left_border_sum )
            max_left_border_sum = left_border_sum;
    }
    max_right_border_sum = 0; right_border_sum = 0;
    for( i=center+1; i<=right; i++ )
    {
        right_border_sum += a[i];
        if( right_border_sum > max_right_border_sum )
            max_right_border_sum = right_border_sum;
    }
    return max3( max_left_sum, max_right_sum,
     max_left_border_sum + max_right_border_sum );
}
```

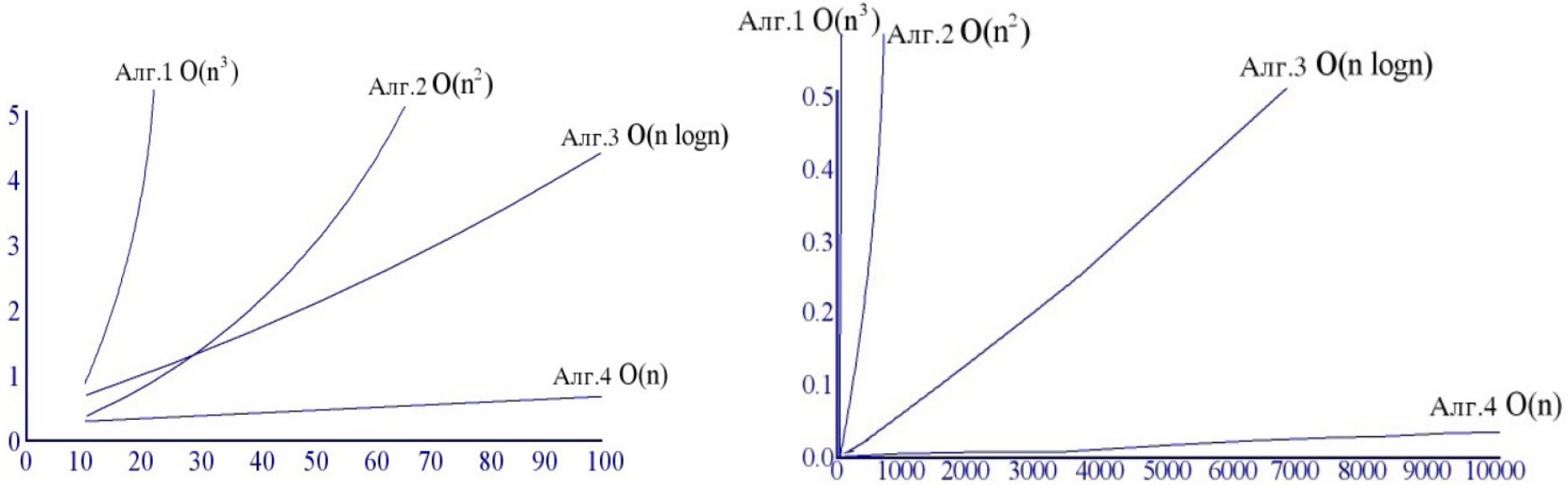
Комплексноста на алгоритмот е $O(n\log_2 n)$

Споредба на алгоритмите

Алгоритам		1	2	3	4
кардиналност		$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
влез	$n = 10$	0.00103	0.00045	0.00066	0.00034
	$n = 100$	0.47015	0.01112	0.00486	0.00063
	$n = 1{,}000$	448.77	1.1233	0.05843	0.00333
	$n = 10{,}000$	NA	111.13	0.68631	0.03042
	$n = 100{,}000$	NA	NA	8.0113	0.29832

Времиња на извршување на секој од претходните алгоритми

Споредба на алгоритмите преку големо О



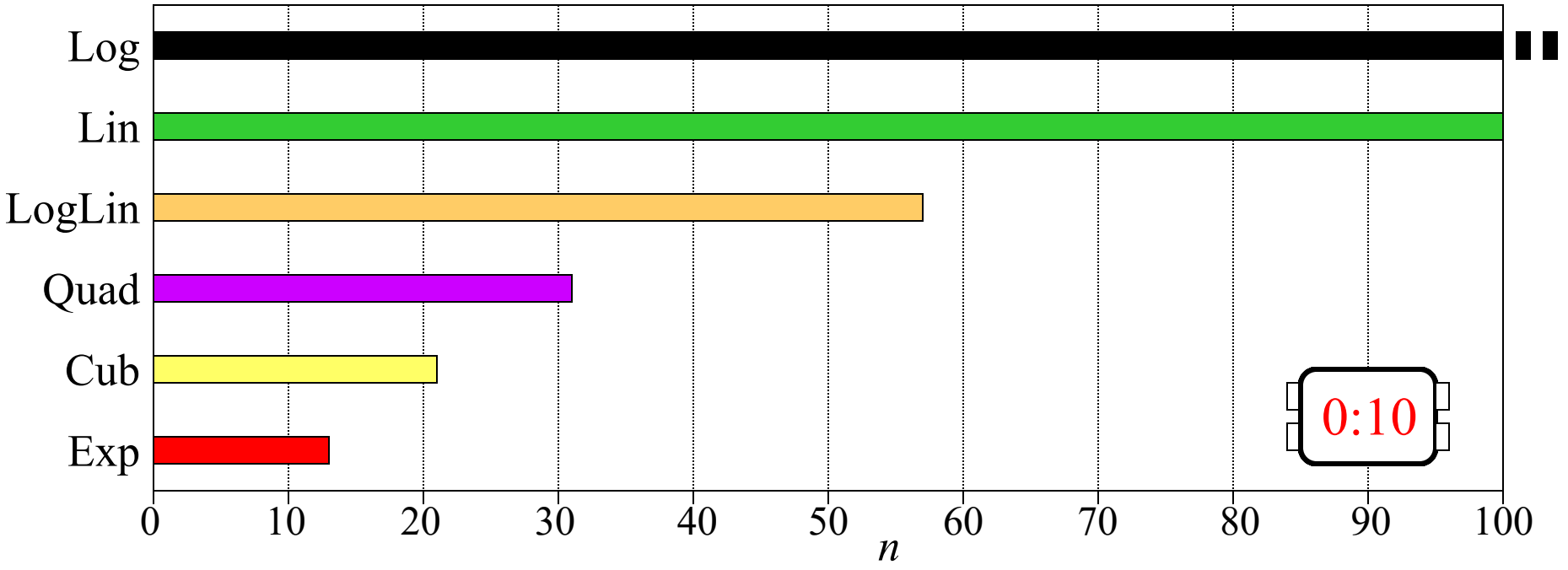
Времиња на извршување на секој од претходните алгоритми

Најчести кардиналности

функција	Име
c	Константа
$\log n$	Логаритамска
$\log^2 n$	Квадратно-логаритамска
n	Линеарна
$n \log n$	
n^2	Квадратна
n^3	Кубна
2^n	Експоненционална

Пример

- Се споредува колку податоци (n) може да обработи секој од алгоритмите за 1, 2, …, 10 секунди:



Брзини на растење на различни кардиналности

n	1	2	4	16	256	4096	65536
$\ln n$	0	1	2	4	8	12	16
$n \cdot \ln$	0	2	8	64	2048	49152	1048576
n^2	1	2	16	256	65536	16777216	4.295×10^9
n^3	1	8	64	4096	16777216	6.872×10^{10}	2.815×10^{14}
2^n	2	4	16	65536	1.16×10^{73}	$> 10^{1232}$	Огромно

Нотации

- **O-нотација**
  - Горна граница на времето на извршување на алгоритамот. Ја мери комплексноста (сложеноста) во **најлош случај**.
- **Ω-нотација**
  - Долна граница на времето на извршување на алгоритамот. Ја мери комплексноста (сложеноста) во **најдобар случај (не е многу корисно).**
- **θ-нотација**
  - Долна и горна граница на времето на извршување на алгоритамот.
  - Се користи при истражување во полето на анализа на алгоритми