# Анализа алгоритама за сортирање

Никола Анђелковић (32/2025)
*Математички факултет, Универзитет у Београду*

---

## 1 Концепт временске сложености

Проучавање ефикасности алгоритама представља кључну дисциплину у теоријском рачунарству. Како се хардверске могућности рачунара непрестано развијају, мерење апсолутног времена извршавања у секундама постаје непоуздан параметар. Уместо тога, користи се концепт **временске сложености**, који описује број елементарних операција у зависности од величине улаза $n$.

### 1.1 Асимптотска анализа и велико "O" нотација

Циљ асимптотске анализе је да се занемаре детаљи имплементације и карактеристике хардвера, фокусирајући се на стопу раста функције за велике вредности $n$.

**Дефиниција 1.** *Кажемо да је функција $T(n) = O(g(n))$ ако постоје позитивне константе $c$ и $n_0$ такве да важи:*

$$(\forall n \geq n_0) \quad 0 \leq T(n) \leq c \cdot g(n).$$

*Функција $g(n)$ представља **асимптотску горњу границу** времена извршавања алгоритма.*

### 1.2 Доминантне класе сложености

У теоријској анализи алгоритама, класе сложености се организују према брзини раста функције која описује број операција. Разликујемо следеће доминантне класе, поређане од најефикаснијих до најмање ефикасних:

1. **Константна сложеност ($O(1)$):** Време извршавања не зависи од величине улаза. Пример је приступ елементу низа по индексу.
2. **Логаритамска сложеност ($O(\log n)$):** Број корака расте логаритамски са величином улаза, као код бинарне претраге.
3. **Линеарна сложеност ($O(n)$):** Време извршавања расте пропорционално броју елемената.
4. **Лог-линеарна сложеност ($O(n \log n)$):** Карактеристична за ефикасне алгоритме сортирања засноване на поређењу.
5. **Квадратна сложеност ($O(n^2)$):** Јавља се код алгоритама са угнежђеним петљама.

6. **Кубна сложеност ($O(n^3)$):** Типична за неке алгоритме у линеарној алгебри.

7. **Експоненцијална сложеност ($O(2^n)$):** Време расте експоненцијално, као код бруталне претраге.

8. **Факторијелна сложеност ($O(n!)$):** Јавља се код проблема који захтевају испитивање свих пермутација.

## 1.3 Значај у пракси

Разлике између различитих класа временске сложености постају драматично изражене са порастом величине улаза $n$. Иако за мале вредности $n$ алгоритми могу показивати сличне перформансе, за велике улазе чак и мале разлике у реду раста доводе до огромних разлика у времену извршавања.

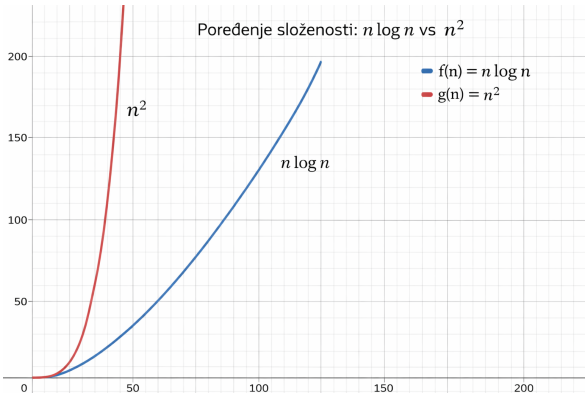
На пример, за $n = 10^6$:

- Алгоритам сложености $O(1)$ извршава се у константном времену.
- Алгоритам сложености $O(\log n)$ захтева око 20 корака.
- Алгоритам сложености $O(n)$ извршава око $10^6$ операција.
- Алгоритам сложености $O(n \log n)$ извршава приближно $2 \cdot 10^7$ операција.
- Алгоритам сложености $O(n^2)$ захтева око $10^{12}$ операција.
- Алгоритам сложености $O(2^n)$ или $O(n!)$ постаје практично неизводљив чак и за много мање вредности $n$.

Ова анализа показује да избор алгоритма директно утиче на изводљивост решавања проблема, посебно у условима обраде великих количина података.

# 2 Класификација одабраних алгоритама

Пре детаљне анализе појединачних метода, корисно је извршити њихову класификацију према временској сложености. У овом раду разматрамо пет основних алгоритама за сортирање.




Слика 1: Поређење раста функција $O(n \log n)$ и $O(n^2)$.

Као што се види на графику, разлика у стопи раста постаје пресудна за велике улазне податке.

Алгоритме делимо у две категорије:

## 2.1 Алгоритми квадратне сложености ($O(n^2)$)

- **Bubble sort** (сортирање мехурићем)
- **Selection sort** (сортирање избором)
- **Insertion sort** (сортирање уметањем)

## 2.2 Алгоритми лог-линеарне сложености ($O(n \log n)$)

- **Merge sort** (сортирање спајањем)
- **Quick sort** (брзо сортирање)

## 2.3 Упоредни приказ параметара

| **Категорија** | **Алгоритми** | **Сложеност** | **Меморија** |
|---|---|---|---|
| *Једноставни* | Bubble, Selection, Insertion | $O(n^2)$ | $O(1)$ |
| *Напредни* | Merge, Quick | $O(n \log n)$ | $O(n)$ / $O(\log n)$ |

Табела 1: Подела алгоритама према кључним параметрима

# 3 Bubble sort

## 3.1 Основна идеја алгоритма

Алгоритам Bubble sort представља један од најједноставнијих алгоритама за сортирање. Заснива се на идеји да се у сваком пролазу кроз низ пореде суседни елементи и размењују уколико нису у исправном поретку. Овај поступак се понавља све док се не изврши пролаз без иједне размене, што значи да је низ сортиран.

Назив алгоритма потиче од чињенице да се већи елементи постепено „померају“ ка крају низа, слично мехурићима који испливавају на површину.

## 3.2 Пример рада алгоритма

Посматрајмо сортирање низа $(6, 1, 4, 3, 9)$.
**Први пролаз:**

- $(6, 1, 4, 3, 9) \to (1, 6, 4, 3, 9)$
- $(1, 6, 4, 3, 9) \to (1, 4, 6, 3, 9)$
- $(1, 4, 6, 3, 9) \to (1, 4, 3, 6, 9)$
- $(1, 4, 3, 6, 9) \to (1, 4, 3, 6, 9)$

**Други пролаз:**

- $(1, 4, 3, 6, 9) \to (1, 4, 3, 6, 9)$
- $(1, 4, 3, 6, 9) \to (1, 3, 4, 6, 9)$
- $(1, 3, 4, 6, 9) \to (1, 3, 4, 6, 9)$
- $(1, 3, 4, 6, 9) \to (1, 3, 4, 6, 9)$

**Трећи пролаз:** нема размена — алгоритам се зауставља.

## 3.3 Имплементација алгоритма

```
vector<int> a = {5, 3, 4, 2, 1};
    int n = a.size();
    bool bilo_razmena;
    do {
        bilo_razmena = false;
        for (int i = 0; i < n - 1; i++) {
            if (a[i] > a[i + 1]) {
                swap(a[i], a[i + 1]);
                bilo_razmena = true;
            }
        }
        n--;
    } while (bilo_razmena);
```

Овај програм реализује алгоритам Bubble sort тако што понавља пролазе кроз низ и размењује суседне елементе док не дође до пролаза без размена.

Након $k$-те итерације спољашње петље, $k$-ти највећи елемент налази се на својој коначној позицији.

## 3.4 Временска сложеност

- Најгори случај: $O(n^2)$ (низ обрнутог поретка)
- Просечан случај: $O(n^2)$
- Најбољи случај: $O(n)$ (ако је низ већ сортиран и користи се оптимизација са провером размена)

## 3.5 Просторна сложеност

Алгоритам користи константну додатну меморију јер се сортирање обавља директно унутар датог низа, без алокације помоћних структура чија величина зависи од броја елемената. Потребно је само неколико помоћних променљивих (индикатор замене и бројач петље), па је просторна сложеност

$$O(1).$$

## 3.6 Својства алгоритма

- **Стабилан алгоритам** — елементи са једнаким кључевима задржавају свој релативни поредак након сортирања.
- **Ради „in-place“** — не захтева додатни простор пропорционалан улазу, већ мења распоред елемената у постојећем низу.
- **Веома једноставан за имплементацију** — структура алгоритма је интуитивна и лако разумљива, што га чини погодним за учење основних концепата сортирања.
- **Неефикасан за велике скупове података** — временска сложеност је у просеку и у најгорем случају $O(n^2)$, па постоје знатно бржи алгоритми за велике улазе.
- **Добар за скоро сортиране низове** — ако је низ већ близу сортиран, алгоритам може брзо да заврши захваљујући провери да ли је било замена у итерацији.

## 3.7 Практична оцена

Иако је Bubble sort користан за илустрацију основних концепата сортирања и анализе алгоритама, у пракси се готово никада не користи због лоших перформанси. Чак и једноставнији алгоритми попут insertion sort показују боље резултате у већини ситуација.

# 4 Selection sort

## 4.1 Основна идеја

Алгоритам selection sort заснива се на идеји да се у свакој итерацији проналази најмањи елемент у несортираном делу низа и поставља на његову коначну позицију. Конкретно, у $i$-тој итерацији одређује се позиција $m$ најмањег елемента у сегменту од $i$ до краја низа, након чега се елемент на позицији $i$ размењује са елементом на позицији $m$.

На тај начин се постепено гради сортирани префикс низа, при чему се у свакој итерацији на своје место доводи следећи по величини елемент.

## 4.2 Пример рада

Посматрајмо сортирање низа:

5 3 4 2 1

- $i = 0$, минимум је на позицији 4 → размена: 1 3 4 2 5
- $i = 1$, минимум је на позицији 3 → размена: 1 2 4 3 5
- $i = 2$, минимум је на позицији 3 → размена: 1 2 3 4 5
- $i = 3$, минимум је на позицији 3 → нема промене

Након ових корака низ је сортиран.

## 4.3 Имплементација

```
vector<int> a{5, 3, 4, 2, 1};
int n = a.size();
for (int i = 0; i < n-1; i++) {
    int m = i;
    for (int j = i+1; j < n; j++)
        if (a[j] < a[m])
            m = j;
    swap(a[i], a[m]);
}
```

## 4.4 Временска сложеност

У свакој итерацији алгоритам пролази кроз преостали део низа како би пронашао минимум. Укупан број поређења је

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2},$$

па је временска сложеност у најбољем, просечном и најгорем случају

$$O(n^2).$$

Број замена је највише $n - 1$, што је знатно мање у поређењу са неким другим алгоритмима квадратне сложености.

## 4.5 Просторна сложеност

Алгоритам користи само неколико помоћних променљивих за чување индекса и привремену размену елемената, па је просторна сложеност

$$O(1).$$

## 4.6 Својства алгоритма

- ради „in-place“
- није стабилан алгоритам
- једноставан за имплементацију
- увек врши исти број поређења без обзира на улаз
- мали број замена
- неефикасан за велике скупове података

## 4.7 Практична оцена

Selection sort се ретко користи за велике скупове података због квадратне временске сложености. Међутим, може бити погодан када је број елемената мали или када је цена замене елемената велика, јер врши минималан број замена. Због једноставности често се користи у едукативне сврхе за илустрацију основних принципа сортирања.

# 5 Insertion sort

## 5.1 Основна идеја

Алгоритам insertion sort сортира низ тако што постепено гради сортирани део низа и у њега убацује један по један елемент на одговарајуће место. У свакој итерацији посматра се следећи елемент и помера се улево све док се не пронађе позиција на којој је поредак исправан.

Другим речима, након $i$-те итерације првих $i+1$ елемената низа чини сортирани подниз.

## 5.2 Пример рада

Посматрајмо сортирање низа:

5 3 4 1 2

- 5 | 3 4 1 2
- 3 5 | 4 1 2
- 3 4 5 | 1 2
- 1 3 4 5 | 2
- 1 2 3 4 5

Вертикална линија означава границу између сортираног и несортираног дела низа.

## 5.3 Имплементација

```
vector<int> a{5, 3, 4, 2, 1};
int n = a.size();
for (int i = 1; i < n; i++) {
    for (int j = i; j > 0 && a[j] < a[j-1]; j--)
        swap(a[j], a[j-1]);
}
```

## 5.4 Временска сложеност

- најбољи случај (већ сортиран низ): $O(n)$
- просечан случај: $O(n^2)$
- најгори случај (обрнути поредак): $O(n^2)$

У најгорем случају сваки елемент мора да се помери кроз скоро цео сортирани део низа, што доводи до квадратичног броја операција.

## 5.5 Просторна сложеност

Алгоритам користи константну количину додатне меморије, па је

$$O(1).$$

## 5.6 Својства алгоритма

- стабилан алгоритам
- ради „in-place“
- адаптиван — брз ако је низ скоро сортиран
- једноставан за имплементацију
- ефикасан за мале скупове података

## 5.7 Практична оцена

Иако има квадратичну временску сложеност, insertion sort је у пракси веома користан за мале низове или скоро сортиране податке. Због малог константног фактора и добре локалности меморије, често се користи као завршна фаза у сложенијим алгоритмима сортирања (на пример када се мали поднизови додатно уређују).

# 6 Merge sort (Сортирање обједињавањем)

## 6.1 Основна идеја

Алгоритам Merge sort почива на стратегији „подели па владај" (*divide-and-conquer*). Процес се заснива на рекурзивном дељењу низа на два дела чије се дужине разликују највише за један. Након што се делови рекурзивно сортирају до нивоа једночланих поднизова, приступа се кључној операцији — **обједињавању**. За разлику од интуитивних метода, овде се проблем димензије $n$ своди на два проблема димензије $n/2$, што драматично побољшава ефикасност.

## 6.2 Пример рада

Претпоставимо да сортирамо низ од 7 елемената: $(38, 27, 43, 3, 9, 82, 10)$.

- **Подела:** Низ се дели на леву половину дужине 3 и десну дужине 4 (или обрнуто). Процес се наставља док не добијемо седам засебних елемената.
- **Обједињавање:** Сортирани поднизови се спајају коришћењем помоћног низа. На пример, поднизови $(27, 38)$ и $(3, 43)$ се обједињују упоређивањем водећих елемената, дајући сортиран сегмент $(3, 27, 38, 43)$.
- **Копирање:** Након обједињавања у помоћном низу, резултат се враћа у оригинални вектор.

## 6.3 Имплементација

```
void merge(vector<int>& a, int l, int s, int d, vector<int>& tmp) {
    int i = l, j = s, k = 0;
    while (i < s && j <= d)
        tmp[k++] = a[i] <= a[j] ? a[i++] : a[j++];
    while (i < s) tmp[k++] = a[i++];
    while (j <= d) tmp[k++] = a[j++];
    for (int p = 0; p < k; p++)
        a[l + p] = tmp[p];
}

void mergesort(vector<int>& a, int l, int d, vector<int>& tmp) {
    if (l < d) {
        int s = l + (d - l) / 2 + 1;
        mergesort(a, l, s - 1, tmp);
        mergesort(a, s, d, tmp);
        merge(a, l, s, d, tmp);
    }
}
```

## 6.4 Временска сложеност

За разлику од *Selection sort*-а где је $T(n) = T(n-1)+O(n)$, код *Merge sort*-а важи рекурентна једначина:

$$T(n) = 2T(n/2) + O(n)$$

На основу ове једначине, временска сложеност у свим случајевима припада класи:

$$O(n \log n)$$

## 6.5 Просторна сложеност

Алгоритам захтева додатни помоћни низ величине $O(n)$. Иако рекурзија креира $O(\log n)$ стек оквира, доминантни фактор је помоћни простор, па је укупна додатна просторна сложеност:

$$O(n)$$

## 6.6 Својства алгоритма

- **Стабилност:** Алгоритам је стабилан јер при обједињавању, у случају једнакости, предност даје елементу из левог подниза.
- **Није „in-place”:** Неопходна је алокација помоћне меморије пропорционалне величини улаза.
- **Није репна рекурзија:** Због два рекурзивна позива и операције након њих, елиминација рекурзије је комплексна.

## 6.7 Практична оцена

*Merge sort* је значајно ефикаснији од квадратних алгоритама за велике $n$. Његова главна снага је гарантована сложеност $O(n \log n)$ чак и у најгорем случају, што га чини поузданијим од *Quick sort*-а у специфичним ситуацијама, упркос већој потрошњи меморије.

# 7 Quick sort (Брзо сортирање)

## 7.1 Основна идеја

*Quick sort* припада групи „подели-па-владај" алгоритама. Слично као код *Selection sort*-а, циљ је да се у сваком кораку одређени елемент доведе на своју коначну позицију. Тај елемент се назива **пивот**.

Кључна разлика у односу на *Merge sort* је следећа:

- Код **Merge sort**-а: Раздвајање на мање проблеме је тривијално (на пола), а обједињавање резултата је нетривијално.
- Код **Quick sort**-а: Раздвајање (корак партиционисања) је нетривијално, док је обједињавање резултата тривијално.

Суштина је у партиционисању: сви елементи мањи или једнаки пивоту иду лево, а сви већи десно. Након тога, пивот је на свом месту, а поступак се рекурзивно наставља.

## 7.2 Пример рада

Посматрајмо партиционисање низа око пивота:

- **Почетни низ:** $(10, 80, 30, 90, 40, 50, 70)$. Нека је пивот 70.
- **Партиционисање:** Елементи се прегрупишу тако да добијемо $(10, 30, 40, 50, 70, 90, 80)$.
- **Резултат:** Пивот 70 је сада на својој коначној позицији. Сада рекурзивно сортирамо леви подниз $(10, 30, 40, 50)$ и десни $(90, 80)$.

## 7.3 Имплементација

Имплементација се заснива на рекурзивним позивима функције `quicksort` након што се изврши кључни корак партиционисања низа око изабраног пивота.

```
using namespace std;

int particionisi(vector<int>& a, int l, int d) {
    int pivot = a[l];
    int i = l, j = d;
    while (i < j) {
        while (i < j && a[j] >= pivot) j--;
        a[i] = a[j];
        while (i < j && a[i] <= pivot) i++;
        a[j] = a[i];
    }
    a[i] = pivot;
    return i;
}

void quicksort(vector<int>& a, int l, int d) {
    if (l < d) {
        int p = particionisi(a, l, d);
        quicksort(a, l, p - 1);
        quicksort(a, p + 1, d);
    }
}

void quicksort(vector<int>& a) {
    if (!a.empty())
        quicksort(a, 0, a.size() - 1);
}
```

## 7.4 Временска сложеност

Временска сложеност директно зависи од тога колико добро пивот дели низ:

- **Просечан случај:** Ако пивот дели низ на приближно једнаке делове, важи $T(n) = 2T(n/2) + O(n)$, што даје сложеност $O(n \log n)$.
- **Најгори случај:** Ако пивот увек дели низ тако да један део има 0, а други $n - 1$ елемената, важи $T(n) = T(n-1) + O(n)$, што води до $O(n^2)$.

## 7.5 Просторна сложеност

Алгоритам ради „у месту” (*in-place*), али користи простор на програмском стеку за рекурзивне позиве. У просеку се формира $O(\log n)$ стек оквира, док у најгорем случају тај број може бити $O(n)$.

## 7.6 Својства алгоритма

- **Није стабилан:** Редослед једнаких елемената се може променити током партиционисања.
- **Ефикасност:** У пракси најбржи алгоритам за дугачке низове.
- **Хибридни приступ:** Реалне имплементације често користе *Insertion sort* за поднизове краће од неколико десетина елемената.

## 7.7 Практична оцена

*Quick sort* представља „златни стандард" сортирања у пракси. Иако теоретски може пасти у квадратну сложеност, уз добре хеуристике избора пивота (попут избора медијане), он надмашује остале алгоритме због малих константи и одличне локалности података.

# 8 Закључак

Кроз анализу пет основних алгоритама за сортирање — *Bubble, Selection, Insertion, Merge* и *Quick sort* — јасно се уочава фундаментална разлика између елементарних и напредних приступа.

- **Елементарни алгоритми** ($O(n^2)$) су интуитивни и лаки за имплементацију, али њихова примена је оправдана само код веома малих скупова података или специфичних стања низа (нпр. скоро сортирани низови код *Insertion sort*-а).
- **Напредни алгоритми** ($O(n \log n)$) представљају основу модерног рачунарства. Док *Merge sort* нуди стабилност и гарантовану сложеност по цену меморије, *Quick sort* остаје најбржи избор за већину практичних примена услед малих константи и рада „у месту”.

Коначан избор алгоритма никада није апсолутан већ зависи од компромиса између временске сложености, расположиве меморије и карактеристика самог хардвера на којем се код извршава.