Capitolul 5 - Sortari simple

Introducere

```
-> O sortare este ordonarea unei colectii de elemente dupa un criteriu dat, rezultatul fiind o permutare a datelor de intrare
```

Cum analizam algoritmii de sortare?

```
-> Criterii relevante : numarul de comparatii si numarul de interschimbari
-> Pentru algoritmii de sortare se aproximeaza asimptotic pentru cazul cel mai favorabi, cel mai defavorabil si cazul mediu
```

Terminologie si notatii

```
typedef struct{
        int cheie;
        //alte campuri
}tipElement_t;
```

```
-> Campul cheie va fi cel mai important din punct de vedere al sortarii
-> pentru tipul definit anterior, ca de altfel pentru orice elemente care trebuie sortate, avem definiti urmatorii operatori :
        1. in = Compara(el1, el2)
                        -> el1 si el2 vor fi comparate pe baza campului cheie
                        -> se vor folosie fie operatori logici (<,>, <=,>=,==) pentru compararea câmpului cheie, fie funcții predefinite (ex. strcmp în cazul șirurilor de caractere)
        2.Interschimba(pel1, pel2)
                        -> interschimba in memorie oel1, pel2 == functia swap
```

```
// o implementare propusa
void swap(tip_element *el1, tip_element *el2) {
        tip_element tmp;

        tmp = *el1;
```

```
        *el1 = *el2;

        *el2 = tmp;
}
```

Find dat un set de articole, de tipul structură definit anterior:

$$a_1, a_2, a_3 \dots a_n$$

Prin **sortare** se înțelege permutarea elementelor şirului într-o anumită ordine:

$$a_{k1}, a_{k2}, a_{k3} \dots a_{kn}$$

astfel încât şirul cheilor să devină **monoton crescător**

$$a_{k1}.cheie \leq a_{k2}.cheie \leq a_{k3}.cheie \leq \cdots \leq a_{kn}.cheie$$

```
-> Desi un set de date de intrare poate contine valori care se repeta, exista si algoritmi de sortare care functioneaza exclusiv pe date unice

-> O metoda de sortare se spune ca este stabila daca dupa sortare, ordinea relativa e elementelor cu chei egale coincide cu cea initiala
        -> Această stabilitate este esenţială în special în cazul în care se execută sortarea după mai multe chei

-> Metodele de sortare sunt clasificate in doua mari categorii după cum sunt înregistrate elementele de sortat:
                1. Sunt înregistrate ca şi tablouri în memoria centrală a sistemului de calcul, ceea ce conduce la sortarea tablourilor numită sortare internă
                2. Sunt înregistrate într-o memorie externă, ceea ce conduce la sortarea fişierelor (secvenţelor) numită şi sortare externă
```

Sortarea tablourilor

```
-> Tablourile se înregistrează în memoria centrală a sistemelor de calcul, motiv pentru care sortarea tablourilor se mai numeşte şi sortare internă
-> Cerinţa fundamentală care se formulează faţă de metodele de sortare a tablourilor se referă la utilizarea cât mai economică a zonei de memorie disponibile.
-> Din acest motive pentru început, prezintă interes numai algoritmii care realizează sortarea "in situ", adică chiar în zona de memorie alocată tabloului.
-> Pornind de la această restricţie, în continuare algoritmii vor fi
```

```
clasificaţi în funcţie de eficienţa lor, respectiv în funcţie de timpul de
execuţie pe care îl necesită.
```

Aprecierea cantitativa a eficientei unui algoritm de sortare se realizeaza prin intermediul unor **indicatori specifici** care depind de numarul total n al elementelor care trebuiesc sortate:
1. Numarul comparatiilor de chei notat cu C pe care le executa algoritmul in vederea sortarii
2. Numarul de interschimbari de elemente, respectiv numarul de miscari de elemente executate dupa algoritm, notat cu M

În cazul unor algoritmi de sortare simpli bazaţi pe aşa-zisele metode directe de sortare atât C cât şi M sunt proporţionali cu n^2 adica sunt $O(n^2)$

Există însă şi metode avansate de sortare, care au o complexitate mult mai mare şi în cazul cărora indicatorii C şi M sunt de ordinul lui $n * log2n(O(n * log2n))$. Raportul $n^2/(n * log2n)$, care ilustrează câştigul de eficienţă realizat de aceşti algoritmi, este aproximativ egal cu 10 pentru n = 64, respectiv 100 pentru n = 1000.

Sortari simple

Metode de sortare directe

Fac parte din categoria Θ(n^2)

Prezintă interes din următoarele motive:

1. Sunt foarte potrivite pentru explicitarea principiilor majore ale sortării.
2. Funcțiile care le implementează sunt scurte şi relativ uşor de înţeles.
3. Deşi metodele avansate necesită mai puţine operaţii, aceste operaţii sunt mult mai complexe în detaliile lor, respectiv metodele directe se dovedesc a fi superioare celor avansate pentru valori mici ale lui n.
4. Reprezintă punctul de pornire pentru metodele de sortare avansate.

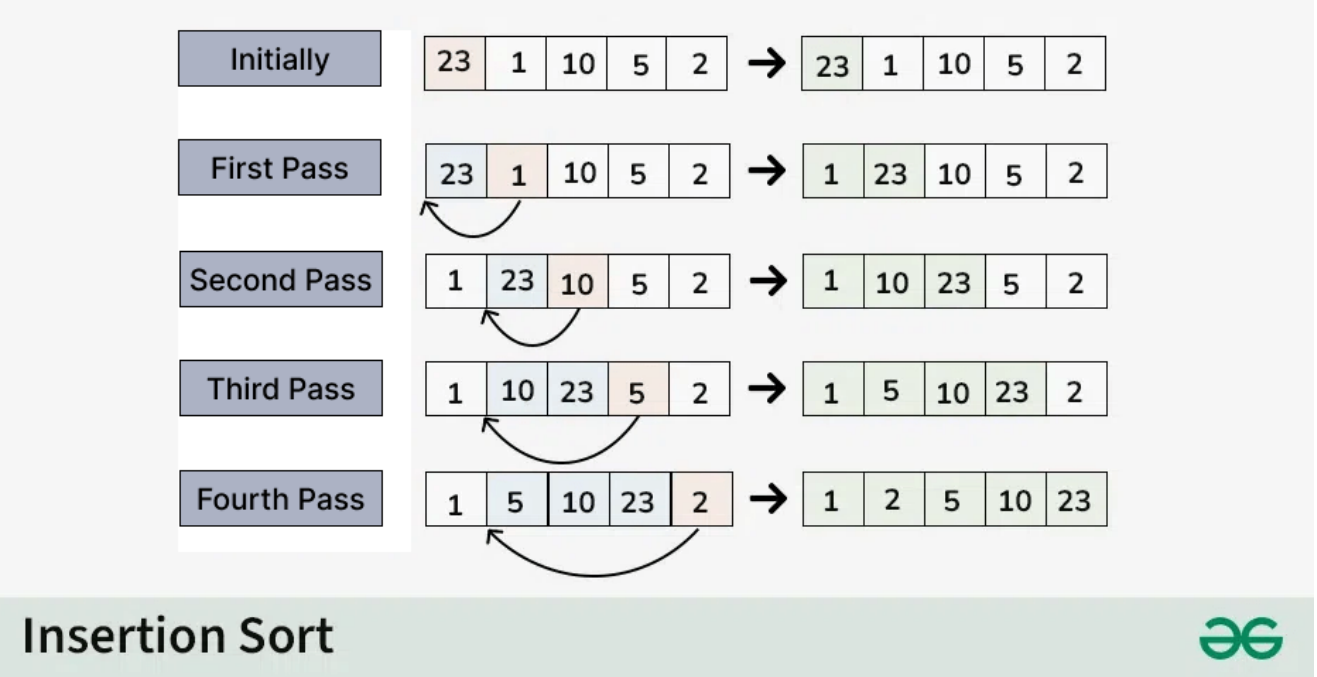
Metodele de sortare care se realizeaza "in situ" se pot clasifica in trei mari categorii:
- Sortarea prin insertie
- Sortarea prin selectie
- Sortarea prin interschimbare

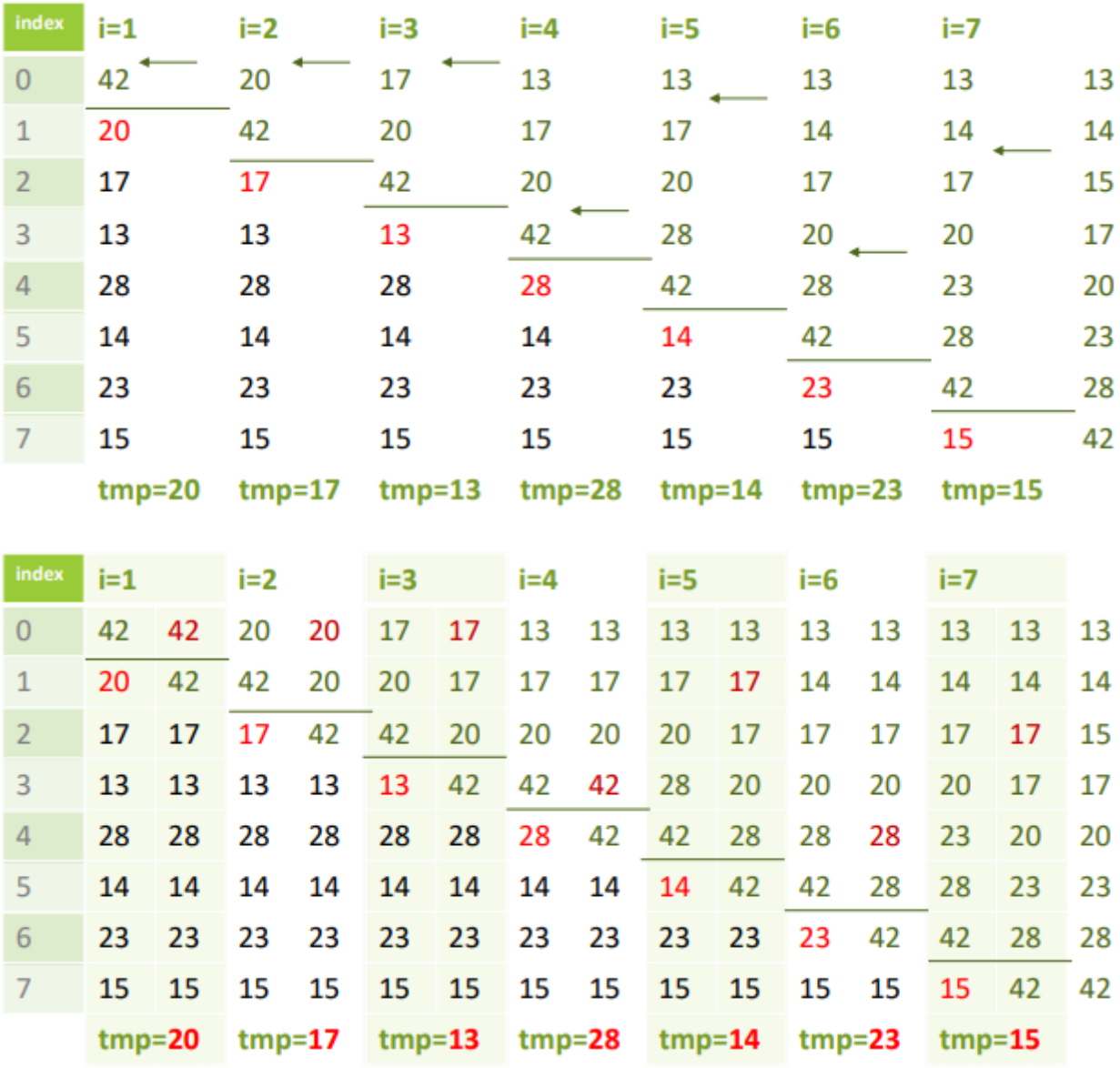
Insertion Sort

Este o sortare directa(simpla) care funcționează prin construirea treptată a unui șir sortat. Este un algoritm eficient pentru șiruri mici sau aproape sortate, dar mai puțin eficient pe măsură ce dimensiunea șirului crește. Ideea principală este să se „insereze” elementele într-un subșir deja sortat, mutându-le treptat la pozițiile corespunzătoare.

Pasii algoritmului

1. Algoritmul presupune că primul element al șirului este deja sortat.
2. Apoi, pentru fiecare element din șirul nesortat, se caută locul potrivit în subșirul sortat.
3. Elementul se mută în acea poziție, iar elementele mai mari din subșirul sortat se deplasează spre dreapta pentru a face loc.
4. Se repetă acest proces pentru toate elementele nesortate, până când întregul șir devine sortat.





index	i=1	i=2	i=3	i=4	i=5	i=6	i=7	
0	42	20	17	13	13	13	13	13
1	20	42	20	17	17	14	14	14
2	17	17	42	20	20	17	17	15
3	13	13	13	42	28	20	20	17
4	28	28	28	28	42	28	23	20
5	14	14	14	14	14	42	28	23
6	23	23	23	23	23	23	42	28
7	15	15	15	15	15	15	15	42
	tmp=20	tmp=17	tmp=13	tmp=28	tmp=14	tmp=23	tmp=15	

index	i=1		i=2		i=3		i=4		i=5		i=6		i=7		
0	42	42	20	20	17	17	13	13	13	13	13	13	13	13	13
1	20	42	42	20	20	17	17	17	17	17	14	14	14	14	14
2	17	17	17	42	42	20	20	20	20	17	17	17	17	17	15
3	13	13	13	13	13	42	42	42	28	20	20	20	20	17	17
4	28	28	28	28	28	28	28	42	42	28	28	28	23	20	20
5	14	14	14	14	14	14	14	14	14	42	42	28	28	23	23
6	23	23	23	23	23	23	23	23	23	23	23	42	42	28	28
7	15	15	15	15	15	15	15	15	15	15	15	15	15	42	42
	tmp=20		tmp=17		tmp=13		tmp=28		tmp=14		tmp=23		tmp=15		

Se face precizarea că în pasul i, din exemplul anterior, primele i elemente sunt deja sortate, astfel încât sortarea constă numai în a insera elementul a[i] la locul potrivit într-o secvenţă deja sortată.

Selectarea locului în care trebuie inserat a[i] se face parcurgând secvenţa destinaţie deja sortată a[0],...,a[i-1] de la dreapta la stânga şi comparând pe a[i] cu elementele secvenţei

Simultan cu parcurgerea, se realizează şi deplasarea spre dreapta cu o poziţie a fiecărui element testat până în momentul îndeplinirii condiţiei de oprire. În acest mod se face loc în tablou elementului care trebuie inserat.

Oprirea parcurgerii se realizează pe primul element a[j] care are cheia mai mică sau egală cu a[i]. Dacă un astfel de element a[j] nu există, oprirea se realizează pe a[0].

Implementarea algoritmului

```
//Pseudocod:
insertion_sort( a[], n)
```

```
        pentru i=1; i< n-1; i++
                reține a[i]
                mută toate elementele a[j] mai mari decât a[i] cu o poziție
                copiază a[i] pe poziția potrivită
```

```
// O varianta de implementare
void insertion_sort(tip_element a[], int n){
        int i, j;
        tip_element tmp;
        for(i = 1; i < n; i++){
                //salvarea elementului de inserat, intr-o zona de memorie
tampon
                tmp = a[i];
                for(j = i; (j > 0) && (tmp.cheie < a[j - 1].cheie); j--){
                        /* mutam elementele din sirul sortat cu cheia mai
mare decat a elementului de inserat spre dreapta */
                        a[j] = tmp;
                        /* inseram elementul la locul potrivit in sirul
sortat */
                }
        }
}
```

```
// O alta implementare -> geeks for geeks
#include <stdio.h>

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
```

```c
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver method
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
/*
****arr = {23, 1, 10, 5, 2}****

****Initial:****

- Current element is ****23****
- The first element in the array is assumed to be sorted.
- The sorted part until ****0th**** index is : ****[23]****

****First Pass:****

- Compare ****1**** with ****23**** (current element with the sorted part).
- Since ****1**** is smaller, insert ****1**** before ****23**** .
- The sorted part until ****1st**** index is: ****[1, 23]****

****Second Pass:****

- Compare ****10**** with ****1**** and ****23**** (current element with the
sorted part).
- Since ****10**** is greater than ****1**** and smaller than ****23**** ,
insert ****10**** between ****1**** and ****23**** .
- The sorted part until ****2nd**** index is: ****[1, 10, 23]****

****Third Pass:****

- Compare ****5**** with ****1**** , ****10**** , and ****23**** (current
element with the sorted part).
- Since ****5**** is greater than ****1**** and smaller than ****10**** ,
insert ****5**** between ****1**** and ****10****
- The sorted part until ****3rd**** index is ****: [1, 5, 10, 23]****

****Fourth Pass:****

- Compare ****2**** with ****1, 5, 10**** , and ****23**** (current element
with the sorted part).
```

```
- Since ****2**** is greater than ****1**** and smaller than ****5****
insert ****2**** between ****1**** and ****5**** .
- The sorted part until ****4th**** index is: ****[1, 2, 5, 10, 23]****

****Final Array:****

- The sorted array is: ****[1, 2, 5, 10, 23]****
*/
```

Analiza sortarii prin insertie

Algoritmul contine un ciclu exterior de dupa i care se reia de n - 1 ori
In cadrul fiecarui ciclu exterior se executa un ciclu interior de lungime variabila dupa j pana la indeplinirea conditiei
-> In pasul i al ciclului exterior, numarul minim de reluari ale ciclului interior este 0, iar numarul maxim este i

Numărul de comparații de chei (C) pentru bucla interioară depinde de numărul de elemente din șirul destinație mai mari decât elementul de inserat, astfel pentru bucla interioară avem:

- O singură comparație pentru datele deja sortate – cazul cel mai favorabil
- i comparații pentru datele în ordine inversă – cazul cel mai defavorabil
- (i+1)/2 în medie, presupunând că toate permutările celor n chei sunt egal posibile – cazul mediu. Probabilitatea se calculează în felul următor:

$$P = \frac{1+2+3+\cdots i}{i} = \frac{\frac{1}{2}i(i+1)}{i} = \frac{i+1}{2}$$

Numărul **total** de **comparații de chei(C)**:

- Cazul cel mai favorabil:

$$C_{min} = \sum_{i=1}^{n-1} \mathbf{1} = n-1 = O(n)$$

- Cazul cel mai defavorabil:

$$C_{max} = \sum_{i=1}^{n-1} \boldsymbol{i} = 1+2+\cdots+(n-1) = \frac{n(n-1)}{2} = O(n^2)$$

- Cazul mediu:

$$C_{me} = \sum_{i=1}^{n-1} \frac{\boldsymbol{i+1}}{\mathbf{2}} = \sum_{i=1}^{n-1} \frac{\boldsymbol{i}}{\mathbf{2}} + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{\frac{1}{2}n(n-1)}{4} + \frac{1}{2}(n-1) = \frac{n^2+n-2}{4} = O(n^2)$$

Numărul de **atribuiri de elemente M** pentru **bucla interioară** este egal cu numărul de comparații de chei din acea buclă (notat C_i), deoarece în interiorul buclei interioare avem doar o atribuire

Numărul de **atribuiri M** pentru o iterație a **buclei exterioare** este egal cu C_i +2 (numărul de atribuiri din bucla interioară tmp = a[i]; și a[j]=tmp), atfel numărul total M este:

$$\sum_{i=1}^{n-1}(C_i+2)$$

Numărul **total** de **atribuiri de elemente**(M):

- Cazul cel mai favorabil:

$$M_{min} = \sum_{i=1}^{n-1}(\mathbf{1}+\mathbf{2}) = 3(n-1) = O(n)$$

- Cazul cel mai defavorabil:

$$M_{max} = \sum_{i=1}^{n-1}(\boldsymbol{i}+\mathbf{2}) = \frac{n(n-1)}{2} + 2(n-1) = \frac{n^2+3n-4}{2} = O(n^2)$$

- Cazul mediu:

$$M_{med} = \sum_{i=1}^{n-1}(\frac{\boldsymbol{i}+\mathbf{1}}{\mathbf{2}}+\mathbf{2}) = \frac{\frac{1}{2}n(n-1)}{4} + \frac{3}{2}(n-1) = \frac{n^2+5n-6}{4} = O(n^2)$$

Avantaje

O sortare simpla

O sortare stabila

Necesita putin spatiu de memorie aditional, iar acesta nu depinde de dimensiunea setului de date de intrare (O(1))

Avantajoasa pentru seturi de date sortate partial (vezi cazul cel mai favorabil)

Nu necesita cunoasterea tuturor elementelor in avans => poate fi adaptata pentru situatii de programare dinamică (pentru date care se primesc secvențial)

Dezavantaje

Ineficientă pentru cazul cel mai defavorabil și mediu (atât C cât și M sunt O(n^2))

În general, nu este o metodă potrivită de sortare cu ajutorul calculatorului, deoarece inserţia unui element presupune deplasarea poziţie cu poziţie în tablou a unui număr de elemente, deplasare care este neeconomică

Selection Sort

Este o sortare directa(simpla).

Asemenea sortării prin inserție și sortarea prin selecție împarte tabloul într-un șir sursă (nesortat) și unul destinație (sortat)

Sortarea prin selecţie foloseşte procedeul de a selecta elementul cu cheia minimă din șirul sursă şi de a schimba între ele poziţia acestui element cu cea a elementului următor șirului destinație.

Se repetă acest procedeu cu cele n-1 elemente rămase, apoi cu cele n-2, etc. terminând cu ultimele două elemente.

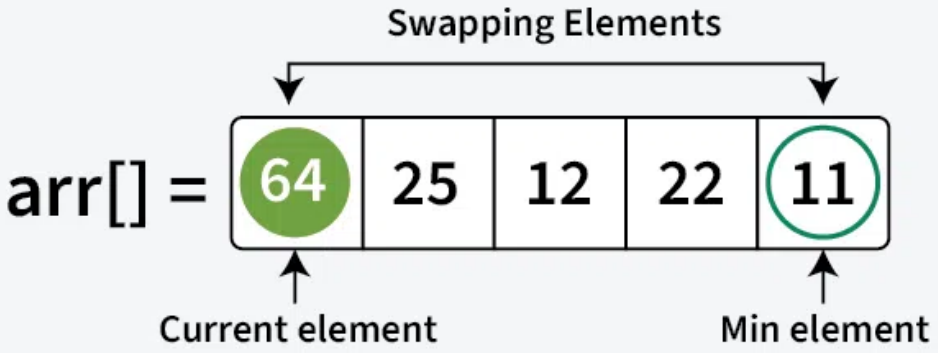
Această metodă este oarecum opusă sortării prin inserţie care consideră la fiecare pas un singur element al secvenţei sursă şi toate elementele secvenţei destinaţie în care se caută de fapt locul de inserţie.

Selecţia în schimb presupune toate elementele secvenţei sursă dintre care îl selectează pe cel cu cheia cea mai mică şi îl depozitează ca element următor al secvenţei destinaţie.

Pasii algoritmului

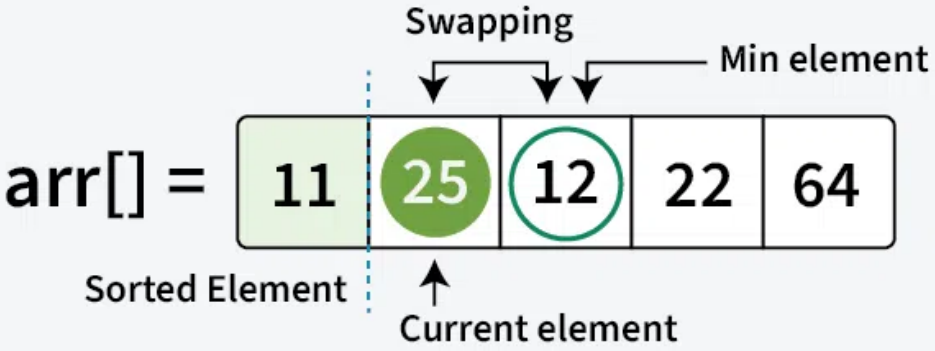
1. Se împarte șirul în două părți: una sortată și una nesortată. La început, întreaga listă este nesortată.
2. Se caută cel mai mic element din partea nesortată.
3. Cel mai mic element găsit este schimbat cu primul element al părții nesortate, plasându-l în partea sortată.
4. Se repetă procesul pentru fiecare element din partea nesortată, până când întregul șir devine sortat.

01 Step | Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



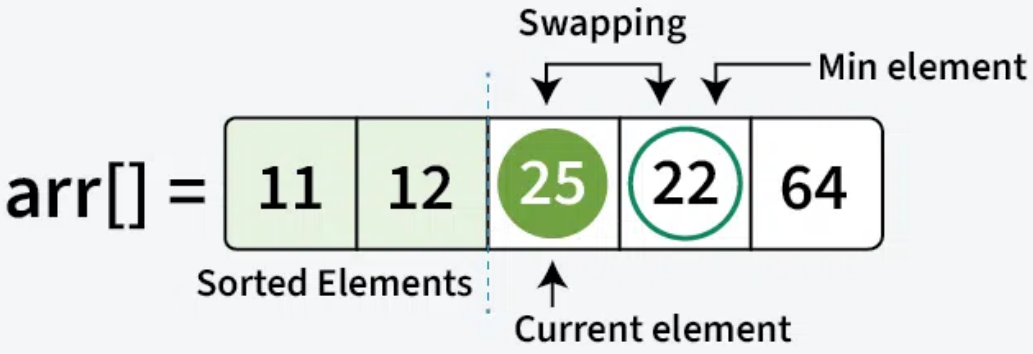
Selection Sort Algorithm

02 Step | Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



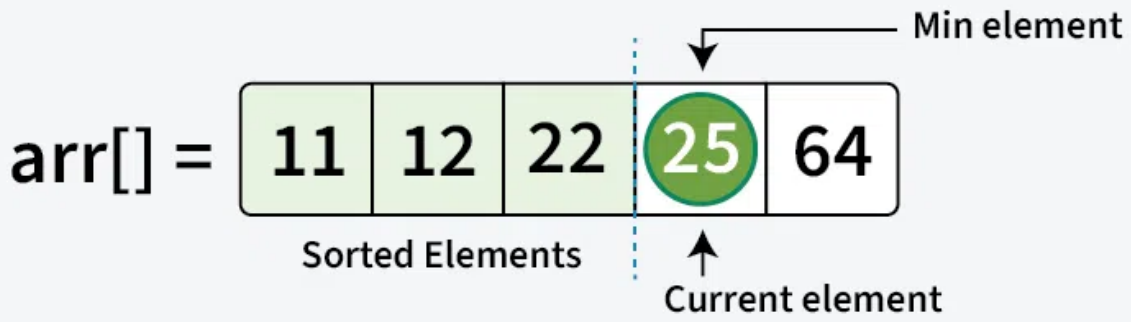
Selection Sort Algorithm

03 Step | Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).

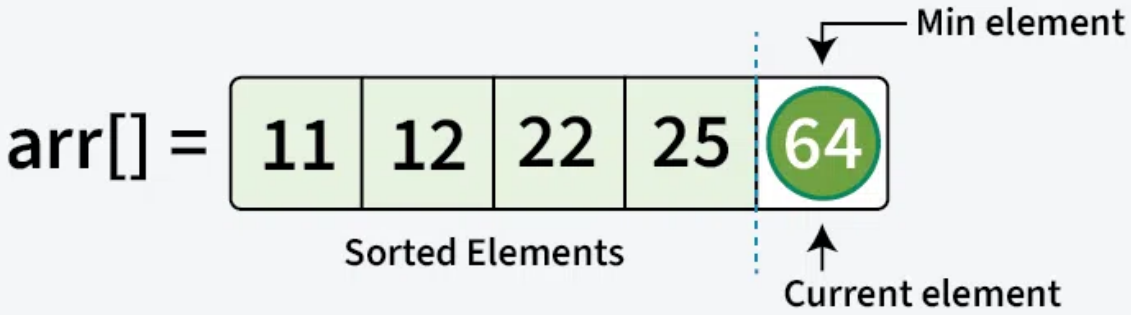


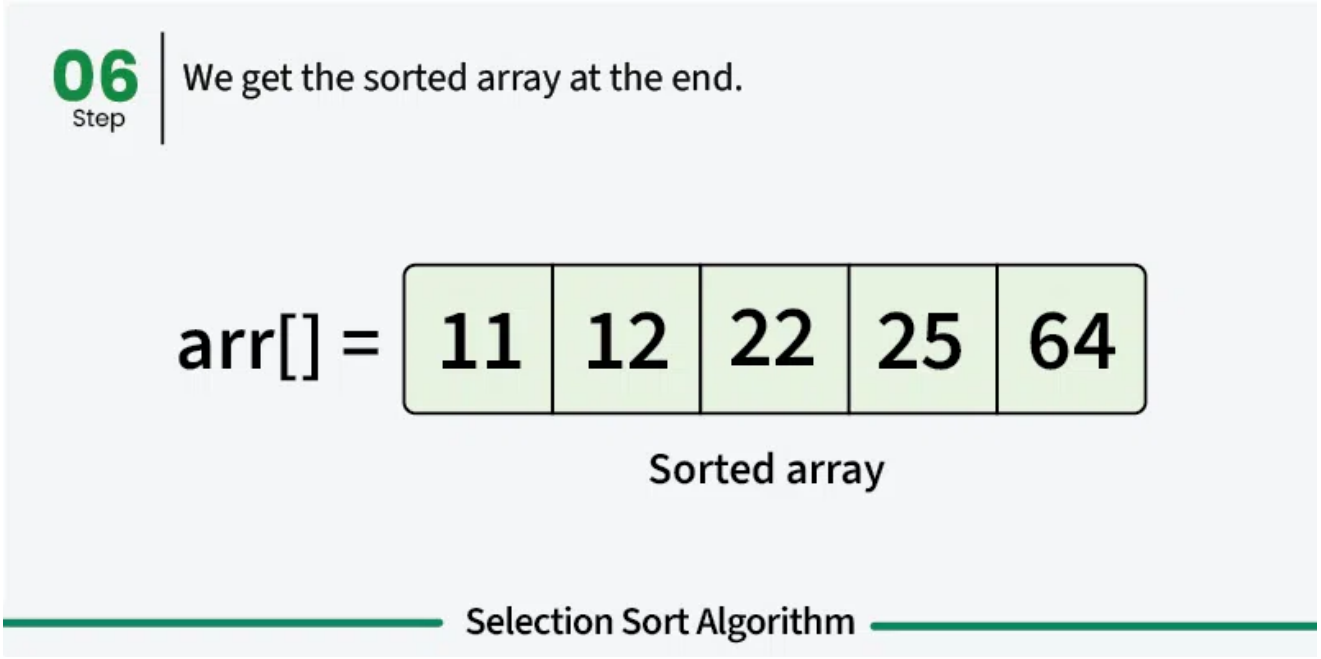
Selection Sort Algorithm

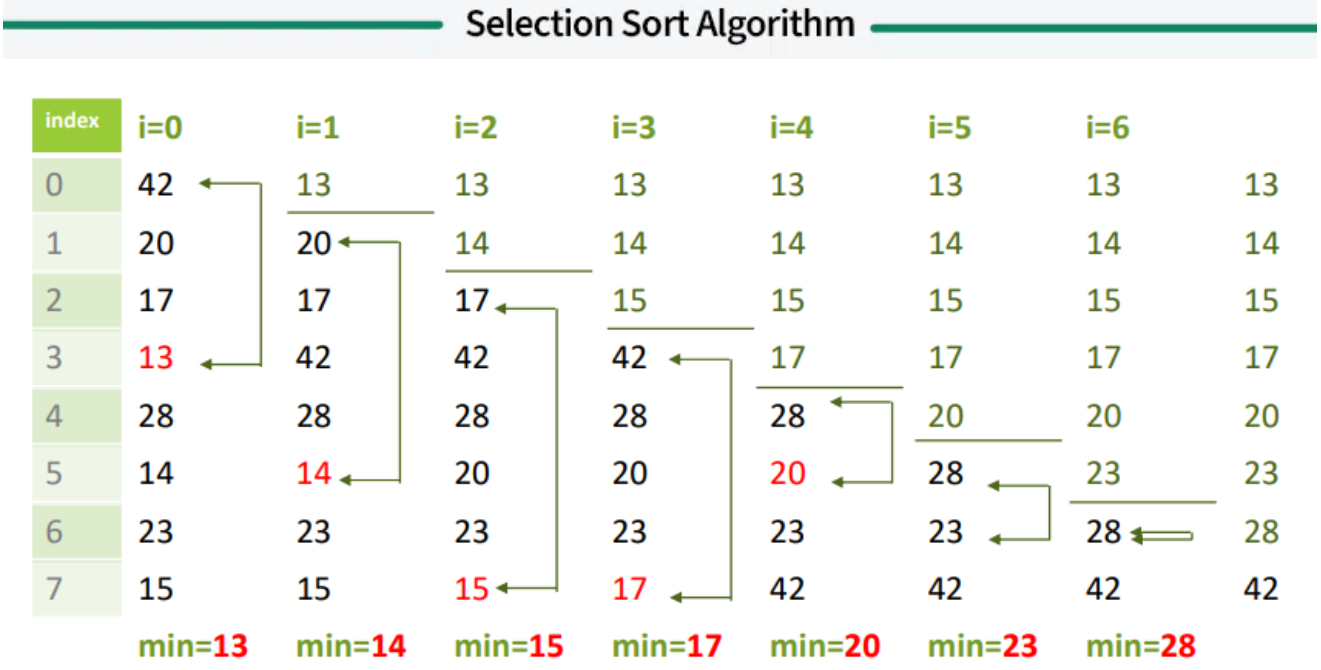
04 Step | Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



05 Step | Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).







Implementarea algoritmului

```
/// pseudocod
selection_sort( a[], n)
        pentru i=0; i< n-1; i++
                reține elementul cu valoarea minima din șirul a[i],…, a[n-1]
                interschimba elementul cu valoarea minima cu a[i]
```

```
void swap(tip_element *el1, tip_element *el2){
        tip_element tmp;
        tmp = *el1;
        *el1 = *el2;
        *el2 = tmp;
}
void selection_sort(tip_element a[], int n) {
        int i, j, min; /* min retine INDEXUL elementului cu valoare minima
```

```
*/
        for (i = 0; i < n - 1; i++) {
                min = i;
                /* cautam minimul in sirul sursa */
                for (j = i + 1; j < n; j++){
                if (a[j].cheie < a[min].cheie)
                        min = j;
                }
                swap(&a[min], &a[i]); /* interschimba cele doua elemente */
        }
}
```

```
//implementare geeks for geeks
#include <stdio.h>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {

        // Assume the current position holds
        // the minimum element
        int min_idx = i;

        // Iterate through the unsorted portion
        // to find the actual minimum
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller element is found
                min_idx = j;
            }
        }

        // Move minimum element to its
        // correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
```

```c
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Analiza sortarii prin selectie

- Ca in cazul sortarii prin insertie avem tot doua cicluri (for)
- Ciclul exterior se executa de (n - 1) ori
- Ciclul interior se executa de (n - 1 - i) ori
- Comparatiile de elemente (C) se executa doar in ciclul interior

C este același pentru **toate cazurile**:

$$C = \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Pentru varianta de implementare propusă anterior, avem și numărul de interschimbări constant.

Avem câte 3 atribuiri (date de interschimbarea elementelor-swap) pentru fiecare valoare a lui *i*

$$M = 3(n-1)$$

Cazul cel mai favorabil este atins când elementele sunt deja sortate

Se observă dezavantajul faptului că în ciuda faptului că tabloul este sortat, se fac totuși interschimbări (elementul cu el însuși, când indecșii *min* și *i* au aceeași valoare).

M_{min} poate fi îmbunătățit prin adăugarea unei verificări suplimentare, dar în acest caz introducem *n-1* comparații (comparații de indecși în acest caz, nu de elemente, neinfluențând valoarea lui C).

```c
if( i != min)
        swap(&a[min], &a[i]);  //interschimba cele doua elemente doar daca sunt diferite
```

Time Complexity: O(n^2) ,as there are two nested loops:

- One loop to select an element of Array one by one = O(n)
- Another loop to compare that element with every other Array element = O(n)
- Therefore overall complexity = O(n) * O(n) = O(n*n) = O(n^2)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables.

Avantaje

- O sortare simpla

- Necesită puțin spațiu de memorie adițional, iar acesta nu depinde de dimensiunea setului de date de intrare (O(1))
- Eficientă din punct de vedere al interschimbărilor

Dezavantaje

- Nu este o sortare stabilă (ordinea relativă a elementelor egale poate fi schimbată).
- Nu este eficientă din punct de vedere al comparațiilor ($O(n^2)$

Bubble Sort

O sortare simpla si directa

Ne imaginăm tabloul de sortat ca o coloană verticală, cu indexul 0 în partea ce mai de sus

Tabloul se parcurge de jos în sus (de la indecșii superiori spre cei inferiori) și de câte ori întâlnim două elemente adiacente care nu sunt în odinea potrivită unul față de altul, acestea se interschimbă

În acest mod elementele ajung pas cu pas la locul potrivit, avansând de jos în sus asemenea unor bule de gaz intr-un lichid, de unde și numele de bubble sort (din limba engleză)

Pentru acest algoritm interschimbarea elementelor două câte două este caracteristica dominantă, de unde și numele de sortare prin interschimbare din limba română

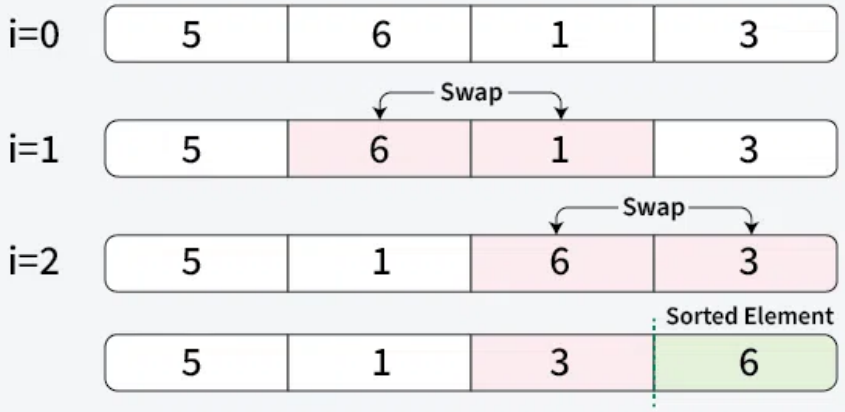
Principiul de bază al sortării prin interschimbare este următorul:
Se compară şi se interschimbă perechile de elemente alăturate, până când toate elementele sunt sortate.
Ca şi la celelalte metode, se realizează treceri repetate prin tablou, de la capăt spre început, de fiecare dată deplasând cel mai mic element al mulţimii rămase spre capătul din stânga al tabloului.

Pasii algoritmului

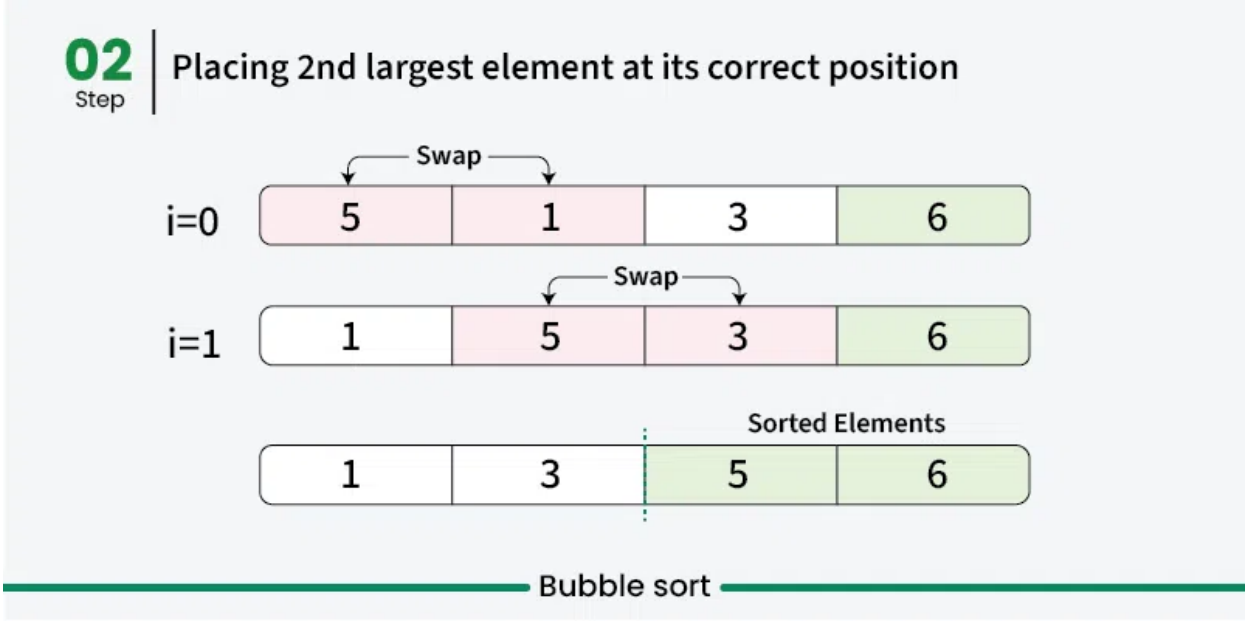
1. Parcurge lista de la început până la sfârșit.
2. Pentru fiecare pereche de elemente adiacente, le compară. Dacă elementul din stânga este mai mare decât cel din dreapta, le interschimbă.
3. Repetă procesul pentru întreaga listă, de fiecare dată fixând ultimul element sortat și repetând pentru elementele rămase.
4. Algoritmul se oprește atunci când, la o parcurgere completă, nu mai este necesară nicio interschimbare.

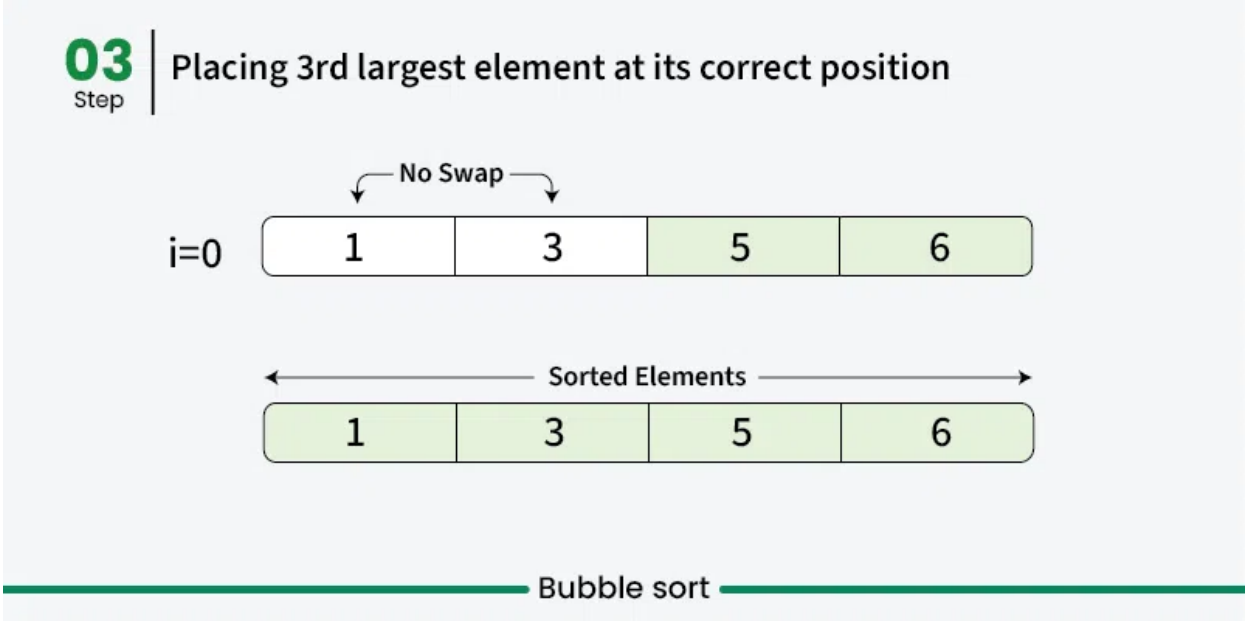
index	i=0	i=1	i=2	i=3	i=4	i=5	i=6	
0	42	13	13	13	13	13	13	13
1	20	42	14	14	14	14	14	14
2	17	20	42	15	15	15	15	15
3	13	17	20	42	17	17	17	17
4	28	14	17	20	42	20	20	20
5	14	28	15	17	20	42	23	23
6	23	15	28	23	23	23	42	28
7	15	23	23	28	28	28	28	42

01 Step | Placing the 1st largest element at its correct position



Bubble sort





Implementarea algoritmului

```
//pseudocod
bubble_sort( a[], n)
        pentru i = 0; i < n-1; i++
                pentru j = n-1; j > i; j--
                        interschimbă elementele de la pozițiile i și j dacă
nu sunt în ordinea potrivită
```

```
void swap(tip_element *el1, tip_element *el2){
        tip_element tmp;
        tmp = *el1;
        *el1 = *el2;
        *el2 = tmp;
}

void bubble_sort(tip_element a[], int n) {
```

```
        int i, j;
        for (i = 0; i < n - 1; i++)
                for (j = n - 1; j>i;j--)
                        if (a[j].cheie < a[j - 1].cheie)
                        /* daca elementele nu sunt in ordinea potrivita */
                        swap(&a[j], &a[j - 1]);
}
```

```
//implementare geeks for geeks
// Optimized implementation of Bubble sort
#include <stdbool.h>
#include <stdio.h>

void swap(int* xp, int* yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n){
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
                swapped = true;
            }
        }

        // If no two elements were swapped by inner loop,
        // then break
        if (swapped == false)
            break;
    }
}

// Function to print an array
void printArray(int arr[], int size){
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

int main(){
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Analiza Bubble Sort

- Ca în cazul sortării prin inserție și selecție avem tot două cicluri (for)
- Ciclul exterior se execută de (n - 1) ori
- Ciclul interior se execută de (n - 1 - i) ori
- Comparațiile de elemente (C) se execută doar în ciclul interior

C este același pentru **toate cazurile**:

$$C = \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Pentru calculul numărului de atribuiri de elemente avem pentru **cazul cel mai defavorabil**, câte 3 (date de funcția swap) pentru fiecare iterație a ciclului interior:

$$M_{max} = 3 * C == \frac{3n(n-1)}{2} = O(n^2)$$

Pentru cazul cel mai favorabil, când elementele sunt deja sortate nu avem atribuiri de elemente:

$$M_{mi} = 0$$

Pentru cazul mediu, avem următoarea probabilitate

$$P = \frac{1+2+3+\cdots n-1-i}{n-i} = \frac{n-i-1}{2}$$

Numărul de mutări mediu devine:

$$M_{med} = 3 * \sum_{i=0}^{n-2} \frac{(n-1-i)}{2} = \frac{3n(n-1)}{4} = O(n^2)$$

Avantaje

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

-> Stabil doar daca nu punem ">=";

Dezavantaje

- Pentru cazul mediu sortarea prin interschimbare face de doua ori mai multe comparații față de sortarea prin inserție și același număr de atribuiri de elemente
- Pentru cazul mediu sortarea prin interschimare face același număr de comparații cu sortarea prin selecție, dar de n ori mai multe atribuiri de elemente

Concluzii

O comparație a algorimilor prezentați (în variantele de implementare propuse)

Număr de **comparații de elemente (C)**

Sortare prin/ Valoare	Insertie	Selectie	Interschimbare
Min	O(n)	$O(n^2)$	$O(n^2)$
Med	$O(n^2)$	$O(n^2)$	$O(n^2)$
Max	$O(n^2)$	$O(n^2)$	$O(n^2)$

O comparație a algorimilor prezentați (în variantele de implementare propuse)

Număr de **atribuiri de elemente (M)**

Sortare prin/ Valoare	Insertie	Selectie	Interschimbare
Min	O(n)	O(n)	0
Med	$O(n^2)$	O(n)	$O(n^2)$
Max	$O(n^2)$	O(n)	$O(n^2)$

O comparație a algorimilor prezentați (în variantele de implementare propuse)

Sortare prin/ Valoare	Insertie	Selectie	Interschimbare
Memorie auxiliară	O(1)	O(1)	O(1)
Sortare stabilă	Da	Depinde de implementare	Da
Observații	Poate fi folosită online		Cod simplu

Important

```
-> Sortarea "in situ" == sortarea in zona de memorie alocata tabloului
```

Sortari in situ

Sortari simple

```
-> Insertion sort
-> Selection sort
-> Bubble sort
```

Sortari stabile

```
-> Insertion sort
```

Sortari instabile

```
-> Selection sort
```