# Tutoriat SD 1: Sortări, Complexitate și Containere STL

**Cupins:**

- Analiza complexității (Big O, cazuri medii, best-case și worst-case)
- Algoritmii de sortare și clasificarea lor
- Containere STL din C++ (vector, stack, queue) și avantajele folosirii lor în comparație cu implementările clasice.

## 1. Analiza Complexității Algoritmice

1.1. Notarea Big O și alte notații
În analiza algoritmilor se folosesc notații precum:

- Big O (O): descrie complexitatea maximă (cazul cel mai defavorabil).
- Omega (Ω): oferă o limită inferioară (cazul cel mai bun).
- Theta (Θ): descrie o limită asimptotică strânsă (cazul mediu când timpul de execuție este similar în toate cazurile)

Big O (O):

- $f(n) \leq C * g(n)$, oricare $n \geq n0$
- Descrie limita superioară a timpului de execuție al algoritmului.

Ω (Omega)

- $f(n) \geq C * g(n)$, oricare $n \geq n0$
- Descrie limita inferioară a timpului de execuție al algoritmului.

θ (Theta):

- $C1 * g(n) \leq f(n) \leq C2 * g(n)$ pentru $n \geq n0$
- Descrie atât limita superioară, cât și limita inferioară a timpului de execuție al algoritmului.

### 1.2. Exemple de Complexități

- O(n): Timp de execuție liniar (ex.: parcurgerea unui vector).
- O(n log n): Timp de execuție optim pentru multi algoritmi de sortare prin comparație (ex.: Merge Sort, Quick Sort – în medie).
- $O(n^2)$: Timp de execuție pătratic (ex.: Bubble Sort, Selection Sort, Insertion Sort în cel mai rău caz).

**2. Algoritmi de Sortare**: Sortarea se poate clasifica în mai multe categorii:

### 2.1. Sortări Elementare

Acestea sunt algoritmi simpli de implementat, însă, în general, nu sunt foarte eficienți pentru seturi mari de date.

Bubble Sort

- Caz cel mai bun: O(n) (dacă se verifică că vectorul este deja sortat)
- Caz mediu și cel mai rău: O($n^2$)

Selection Sort

- Toate cazurile: O($n^2$)

Insertion Sort

- Caz cel mai bun: O(n) (vector deja sortat)
- Caz mediu și cel mai rău: O($n^2$)

### 2.2. Sortări prin Comparație

Aceste algoritmi compară elementele între ele și, în medie, ating complexitatea de O(n log n):

Quick Sort

- Caz mediu: O(n log n)
- Caz cel mai rău: O($n^2$) – însă acest scenariu poate fi evitat prin alegerea inteligentă a pivotului (ex.: mediana din 3, mediana medianelor sau pivot aleator)

Merge Sort

- Toate cazurile: O(n log n)
- Notă: Necesită memorie suplimentară pentru vectorul auxiliar

Heap Sort

- Toate cazurile: O(n log n)

### 2.3. Sortări prin Numărare

Aceste metode nu se bazează pe comparații directe, ci pe organizarea datelor într-o structură auxiliară:

Counting Sort

- Complexitate: O(n + k), unde k este intervalul de valori

Radix Sort

- Complexitate: O(n · k), unde k reprezintă numărul de cifre sau componente

Bucket Sort

- Complexitate medie: O(n + k) – eficiența depinde de distribuția datelor

### 2.5. Recomandări

În funcție de date:

- Numere naturale (ex. sub 10^6): Counting Sort poate fi foarte eficient.
- Șiruri de caractere / numere în baze diferite (ex. sub 10^18): Radix Sort este adesea preferat.
- Numere întregi: Bucket Sort poate oferi performanțe bune.

ATENȚIE: Utilizarea structurilor de date auxiliare (ex.: vector auxiliar în Merge Sort) influențează și complexitatea spațială a algoritmului.

## 3. Containerii STL și Implementările Clasice

Containerele din STL sunt clase ce gestionează automat memoria și oferă funcționalități standard, economisind timp la dezvoltare și reducând posibilitatea apariției erorilor.

### 3.1. Vector

Vectorul din STL este echivalentul unui tablou dinamic:

- Alocarea memoriei: La declarare se alocă un spațiu inițial; ulterior, la atingerea capacității, vectorul se realocă de obicei dublându-și mărimea.
- Operația `push_back` are o complexitate amortizată O(1), însă realocările pot costa O(n) la anumite inserări.

Exemplu clasic vs STL:

```
// Implementare clasică a unui vector
class Vector {
    int N;          // numărul de elemente
    int capacitate;  // capacitatea curentă
    int v;
public:
    Vector(int N, int val, int capacitate) {
        this->N = N;
        this->capacitate = capacitate;
        v = new int[capacitate]();
        for (int i = 0; i < N; ++i)
            v[i] = val;
    }
    // Metode precum Push_Back, Pop_Back, Resize, etc.
    void Push_Back(int val) {
        if (N == capacitate)
            Resize();
        v[N++] = val;
```

```
    }
    // ...
    ~Vector() { delete[] v; }
};
```

În STL, vectorul se comportă similar, dar beneficiază de o implementare optimizată și mult mai robustă.

### 3.2. Stack & Queue

Aceste structuri de date permit gestionarea datelor în mod specific:

Stack (Stivă)

- Principiu: LIFO (Last In, First Out)
- Operații de bază: `push` (adăugare), `pop` (eliminare) și `top` (extragere element de la vârf)
- Complexitate: toate operațiile sunt O(1)

Queue (Coadă)

- Principiu: FIFO (First In, First Out)
- Operații de bază: `push` (adăugare), `pop` (eliminare) și `front` (extragere element din față)
- Complexitate: toate operațiile sunt O(1)

**Implementare Clasică vs STL**

- Implementare clasică a unei stive:

```
void SolveNormal() {
   int T;
   cin >> T;
   const int Nmax = 1000;
   int Stiva[Nmax];
   int MarimeStiva = 0;
   while (T--) {
      char query[10];
      cin >> query;
      if (query[1] == 'u') { // operația "push"
         int x;
         cin >> x;
         if (MarimeStiva < Nmax)
            Stiva[MarimeStiva++] = x;
```

```
        } else if (query[0] == 'p') { // operația "pop"
            if (MarimeStiva > 0)
                MarimeStiva--;
        } else { // operația "top"
            if (MarimeStiva > 0)
                cout << Stiva[MarimeStiva - 1] << '\n';
        }
    }
}
```

- Implementare folosind STL:

```
include <stack>
void SolveSTL() {
    int T;
    cin >> T;
    stack<int> Stiva;
    while (T--) {
        string query;
        cin >> query;
        if (query[1] == 'u') {
            int x;
            cin >> x;
            Stiva.push(x);
        } else if (query[0] == 'p') {
            if (!Stiva.empty())
                Stiva.pop();
        } else {
            if (!Stiva.empty())
                cout << Stiva.top() << '\n';
        }
    }
}
```

Un procedeu similar se aplică și pentru coadă:

- Implementare clasică a unei cozi: se folosește un tablou cu două indicatoare (Left și Right) pentru a simula operațiile FIFO.
- Implementare STL:

```
include <queue>
```

```
void SolveSTL() {
    int T;
    cin >> T;
    queue<int> Coada;
    while (T--) {
        char query[10];
        cin >> query;
        if (query[1] == 'u') {
            int x;
            cin >> x;
            Coada.push(x);
        } else if (query[0] == 'p') {
            if (!Coada.empty())
                Coada.pop();
        } else {
            if (!Coada.empty())
                cout << Coada.front() << '\n';
        }
    }
}
```

Toate aceste operații în cadrul STL se execută în O(1), iar folosirea lor reduce semnificativ riscul de erori legate de gestionarea manuală a memoriei.