AVANSAT · OLIMPIADĂ · LECȚIA 1

# STL pentru concurs: sort cu comparator, set, map, priority_queue

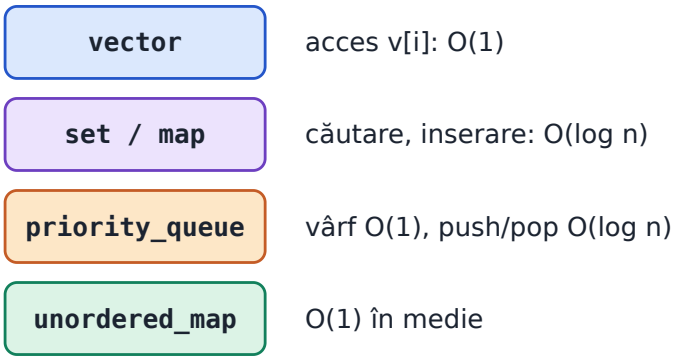
Durată: 75 min  Nivel: olimpiadă / informatică intensiv  Înainte: lecțiile x-04, xi-17

**CE VEI ȘTI DUPĂ LECȚIE**

- Sortezi orice fel de date cu un comparator scris de tine.
- Alegi containerul potrivit: vector, set, map, priority_queue (și știi cât costă fiecare operație).
- Folosești lower_bound, unique și binary_search ca să nu rescrii algoritmi cunoscuți.
- Scrii intrarea/ieșirea rapidă pentru teste mari.

## 1. De ce STL la concurs?

La olimpiadă timpul este limitat: ai 3–4 ore pentru 3–4 probleme. Cine scrie de mână o sortare sau un arbore de căutare pierde minute și face greșeli. **STL** (biblioteca standard C++) îți dă gata scrise și optimizate: vectori, mulțimi, dicționare, cozi cu priorități, sortări, căutări binare. Meseria este să știi **ce alegi** și **cât costă**.




*Ce container alegi*

| Container | Ce ține | Operații principale | Cost |
|---|---|---|---|
| `vector<T>` | șir de elemente, în ordinea introducerii | `v[i]`, `push_back`, `size` | O(1) |
| `set<T>` | elemente **distincte**, **ordonate** | `insert`, `erase`, `find`, `count`, `lower_bound` | O(log n) |
| `multiset<T>` | la fel, dar **cu dubluri** | la fel | O(log n) |
| `map<K, V>` | perechi cheie → valoare, ordonate după cheie | `m[k]`, `find`, `erase` | O(log n) |
| `priority_queue<T>` | cel mai mare element (implicit) | `push`, `top`, `pop` | O(log n); `top` O(1) |

| Container | Ce ține | Operații principale | Cost |
|---|---|---|---|
| `unordered_map/set` | ca `map` / `set`, dar fără ordine | `m[k]`, `count` | O(1) în medie |

**DE ȘTIUT**

`O(log n)` înseamnă că pentru n = 1 000 000 o operație costă în jur de 20 de pași, nu un milion. De aceea 1 000 000 de inserări într-un `set` rulează în sub o secundă (despre complexitate: lecția 2).

## 2. Citire și afișare rapidă

La $10^6$ numere, `cin` și `cout` sunt vizibil mai lente decât `scanf` și `printf`, pentru că sunt legate de ele. Două linii la începutul lui `main` rezolvă asta:

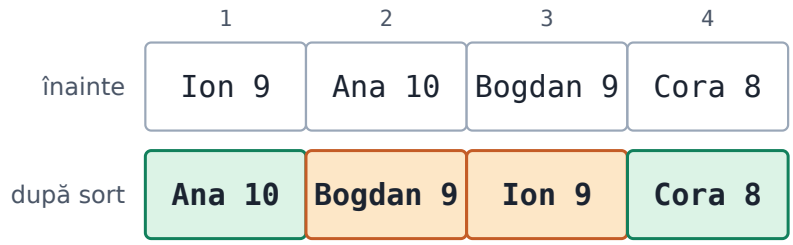
```
ios::sync_with_stdio(false);   // desface legătura cu scanf/printf
cin.tie(nullptr);              // cin nu mai golește cout înainte de fiecare citire
```

**ATENȚIE — GREȘELI FRECVENTE**

După `sync_with_stdio(false)` **nu mai amesteca** `cin` cu `scanf` (sau `cout` cu `printf`) în același program: ordinea citirilor devine imprevizibilă. Folosește doar unul dintre ele. Pe de altă parte, evită `endl` când afișezi multe linii: golește tamponul de fiecare dată. Folosește `'\n'`.

## 3. Sortare cu comparator

`sort(v.begin(), v.end())` sortează crescător. Pentru alt criteriu îi dai un **comparator**: o funcție `bool f(a, b)` care întoarce `true` dacă `a` **trebuie să fie înaintea lui** `b`.




Bogdan și Ion au 9: „Bogdan” < „Ion” alfabetic, deci Bogdan primul

*Sort cu comparator*

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

struct Elev { string nume; int nota; };

bool maiInainte(const Elev &a, const Elev &b)
{
    if (a.nota != b.nota) return a.nota > b.nota;   // nota mai mare primește locul întâi
    return a.nume < b.nume;                         // la egalitate: ordine alfabetică
}

int main()
{
    int n;
    cin >> n;
    vector<Elev> e(n);
    for (auto &x : e) cin >> x.nume >> x.nota;
    sort(e.begin(), e.end(), maiInainte);
    for (auto &x : e) cout << x.nume << ' ' << x.nota << '\n';
    return 0;
}
```

**DE CE AȘA?**

**De ce comparatorul întoarce `>` și nu `>=`?** `sort` cere o ordine „strictă”: pentru două elemente egale, `maiInainte(a, b)` trebuie să dea `false` în **ambele** sensuri. Cu `>=`, ar întoarce `true` și pentru `a, a`, iar la valori egale `sort` poate încerca să iasă din vector și programul se oprește cu eroare (segmentation fault). **De ce `const Elev &`?** Ca să nu se copieze structura la fiecare comparație: sunt n·log n comparații.

**SFAT**

Pentru criterii simple, folosește `pair` sau `tuple`: un `pair<int,int>` se compară automat întâi după `first`, apoi după `second`. Dacă salvezi `(-nota, nume)`, un simplu `sort` fără comparator dă exact ordinea de mai sus.

**Lambda (comparator „pe loc”).** Același lucru fără o funcție separată:

```cpp
sort(e.begin(), e.end(), [](const Elev &a, const Elev &b) {
    return a.nota != b.nota ? a.nota > b.nota : a.nume < b.nume;
});
```

## 4. Dicționarul: `map`

`map<K, V>` ține perechi (cheie, valoare), **ordonate după cheie**. Cea mai folosită schemă la concurs: **frecvențe** pentru valori mari (unde un vector de frecvență nu încape).

```cpp
#include <iostream>
#include <map>
using namespace std;

int main()
{
    int n, x;
    cin >> n;
    map<int, int> frecv;                // valoare -> de câte ori apare
    for (int i = 0; i < n; i++)
    {
        cin >> x;
        frecv[x]++;                     // dacă x nu există, se creează cu 0
    }
    for (auto &p : frecv)               // parcurgere în ordinea crescătoare a cheii
        cout << p.first << " apare de " << p.second << " ori\n";
    return 0;
}
```

| Pas | Citit | `frecv` după pas |
|---|---|---|
| 1 | 3 | {3: 1} |
| 2 | 1 | {1: 1, 3: 1} |
| 3 | 3 | {1: 1, 3: 2} |
| 4 | 2 | {1: 1, 2: 1, 3: 2} |
| 5 | 1 | {1: 2, 2: 1, 3: 2} |
| 6 | 3 | {1: 2, 2: 1, 3: 3} |

**DE CE AȘA?**

**De ce funcționează `frecv[x]++` și când `x` nu există?** Operatorul `[]` al lui `map` **creează** elementul cu valoarea implicită (0 pentru `int`) dacă nu îl găsește, apoi îl returnează ca să poată fi mărit. **Capcană:** același `[]` creează chei și când doar *citești* `frecv[y]`. Pentru o simplă verificare folosește `frecv.count(y)` sau `frecv.find(y) != frecv.end()`, ca să nu umpli dicționarul cu chei nedorite.

## 5. Mulțimea: `set` și `lower_bound`

`set<int>` păstrează valorile **distincte**, **sortate**. În plus, găsește rapid „cel mai apropiat element”:

| Apel | Rezultat |
|---|---|
| `s.lower_bound(x)` | iteratorul spre **primul** element `≥ x` |
| `s.upper_bound(x)` | iteratorul spre primul element `> x` |
| `s.end()` | „nu există” (după ultimul element) |




*set*

```cpp
#include <iostream>
#include <set>
using namespace std;

int main()
{
    int n, q, x;
    cin >> n;
    set<int> s;
    for (int i = 0; i < n; i++) { cin >> x; s.insert(x); }
    cout << s.size() << '\n';            // câte valori distincte
    cin >> q;
    while (q--)
    {
        cin >> x;
        auto it = s.lower_bound(x);      // primul element >= x
        if (it == s.end()) cout << "nu exista\n";
        else cout << *it << '\n';
    }
    return 0;
}
```

Mulțimea este {4, 7, 10} (trei valori distincte). Interogările: pentru 5 primul element ≥ 5 este 7; pentru 7 este chiar 7; pentru 11 nu există niciunul.

> **DE CE AȘA?**
>
> **De ce `s.lower_bound(x)` și nu `lower_bound(s.begin(), s.end(), x)`?** Varianta-membru folosește structura arborescentă a mulțimii și costă O(log n). Funcția generală din `<algorithm>` merge și pe `set`, dar parcurge cu iteratori pas cu pas: **O(n)**. Pe `vector` sortat, în schimb, se folosește varianta generală (secțiunea 7).

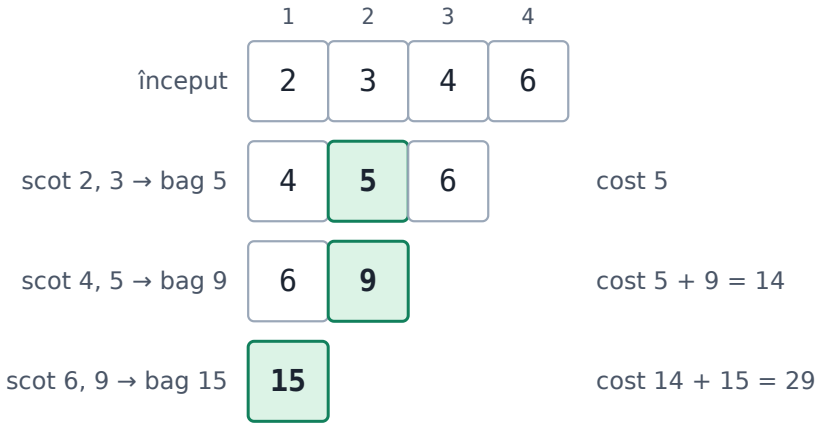
## 6. Coada cu priorități: `priority_queue`

`priority_queue<T>` ține elementele astfel încât **vârful** este mereu cel **mai mare**. Pentru cel mai mic, se declară cu `greater`:

```
priority_queue<int> mx;                                  // max-heap: top() = cel mai mare
priority_queue<int, vector<int>, greater<int>> mn;       // min-heap: top() = cel mai mic
```

**Problema funiilor.** Ai n funii de lungimi date. Legarea a două funii de lungimi a și b costă a + b și produce o funie de lungime a + b. Care este costul minim ca să obții o singură funie?

**Idee greedy:** lega mereu cele **mai scurte două**.




costul total al legării celor 4 funii este 29

*Funii*

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int main()
{
    int n;
    cin >> n;
    priority_queue<long long, vector<long long>, greater<long long>> pq;   // min-heap
    for (int i = 0; i < n; i++) { long long x; cin >> x; pq.push(x); }
    long long cost = 0;
    while (pq.size() > 1)
    {
        long long a = pq.top(); pq.pop();     // cele mai scurte două funii
        long long b = pq.top(); pq.pop();
        cost += a + b;                        // costul legării lor
        pq.push(a + b);
    }
    cout << cost;
    return 0;
}
```

**DE CE AȘA?**

**De ce `long long`?** Cu n = $10^5$ funii de câte $10^9$, costul total depășește cu mult $2 \cdot 10^9$ (limita lui `int`). La concurs, când suma poate crește, alegi `long long` de la început. **De ce greedy este corect?** Funia legată prima „se plătește” de fiecare dată când este refolosită într-o legare ulterioară; deci cele mai scurte trebuie folosite cât mai devreme (aceeași idee ca la codul Huffman).

## 7. Pe vector sortat: `unique`, `lower_bound`, `binary_search`

Când nu ai nevoie să inserezi între timp, e mai simplu și mai rapid: **sortezi o dată** și cauți binar.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int n, q, x;
    cin >> n;
    vector<int> v(n);
    for (auto &a : v) cin >> a;
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());          // elimină dublurile
    cout << v.size() << '\n';
    cin >> q;
    while (q--)
    {
        cin >> x;
        int mai_mici = lower_bound(v.begin(), v.end(), x) - v.begin();   // câte sunt < x
        bool exista = binary_search(v.begin(), v.end(), x);
        cout << mai_mici << ' ' << (exista ? "da" : "nu") << '\n';
    }
    return 0;
}
```

După sortare și `unique` rămân valorile 1 3 5 9. Pentru x = 5: `lower_bound` arată poziția 2 (două valori mai mici: 1 și 3) și `binary_search` găsește 5.

> **DE CE AȘA?**
>
> **De ce `unique` trebuie urmat de `erase`?** `unique` doar **mută** elementele distincte la început și întoarce un iterator spre noul „sfârșit"; vectorul își păstrează lungimea. `erase` taie coada rămasă. Fără `erase`, `v.size()` ar rămâne 7. **De ce funcționează `unique` doar după `sort`?** Elimină dublurile **vecine**; după sortare, valorile egale sunt una lângă alta.

## 8. Ce structură pentru ce problemă?

| Problema | Alegerea | De ce |
|---|---|---|
| frecvențe pentru valori până la $10^6$ | `vector<int> f(1000001)` | acces O(1), simplu |
| frecvențe pentru valori până la $10^9$ | `map<int,int>` | un vector nu încape |
| câte valori distincte | `set` sau `sort + unique` | aceeași complexitate O(n log n) |
| „cel mai apropiat element ≥ x" cu inserări | `set` + `lower_bound` | ordonat, O(log n) |
| „cel mai mic element, apoi îl scot" | `priority_queue` cu `greater` | vârf O(1), scoatere O(log n) |

| Problema | Alegerea | De ce |
|---|---|---|
| maximul curent dintr-un flux de numere | `priority_queue` | la fel |
| cod rapid, ordinea nu contează | `unordered_map` | O(1) în medie |

> **DE ȘTIUT**
>
> `unordered_map` poate fi ținta unor teste „împotriva hash-ului" pe unele platforme de concurs (timpul crește la O(n)). Pe olimpiadele din România rar se întâmplă, dar dacă ai nevoie de siguranță, `map` este alegerea sigură.

## 9. Greșeli frecvente

| Greșeala | Ce se întâmplă | Corect |
|---|---|---|
| comparator cu `<=` sau `>=` | blocaj sau eroare la `sort` | `<` sau `>` |
| `map[x]` doar pentru a verifica existența | creează chei noi, `size()` crește | `count(x)` sau `find(x)` |
| `v.erase(unique(...))` fără `sort` | dublurile nevecine rămân | `sort` mai întâi |
| `priority_queue<int>` când vrei minimul | `top()` dă maximul | `greater<int>` |
| iterator folosit după `erase` | comportament nedefinit | `it = s.erase(it)` |
| `endl` în buclă de $10^6$ linii | program lent | `'\n'` |
| `cin` și `scanf` amestecate după `sync_with_stdio(false)` | citiri în altă ordine | un singur stil |

## 10. La olimpiadă

În problemele de olimpiadă STL este permis aproape peste tot (verifică regulamentul concursului: la unele etape pentru clasele mici se cere rezolvare fără STL). Observă în enunț indiciile: „în ordinea ..." (sortare), „valori distincte" (set), „cel mai mic / mare, repetat" (priority_queue), „frecvența" (map sau vector de frecvență), „valori până la $10^9$" (map, nu vector).

## 11. Exersează pe hârtie

**1.** Ce afișează programul de la secțiunea 4 (map) pentru intrarea `5` / `7 7 7 2 2`?

**2.** După `set<int> s; s.insert(4); s.insert(2); s.insert(4); s.insert(9);`, câte elemente are `s` și în ce ordine sunt?

**3.** Pe `set` cu elementele {4, 7, 10}, ce întoarce `lower_bound(8)`? Dar `lower_bound(10)`? Dar `upper_bound(10)`?

**4.** Rulează în minte problema funiilor pentru lungimile 1, 2, 3. Ce cost obții?

**5.** Scrie un comparator care ordonează numerele crescător după **ultima cifră**, iar la egalitate descrescător după valoare.

**6.** De ce comparatorul `bool f(int a, int b) { return a <= b; }` este greșit?

**7.** Într-un `priority_queue<int>` (max-heap) pui 5, 1, 8, 3. Ce afișează `top()`? După un `pop()`, ce afișează?

**8.** Ce valoare are `v.size()` după `v = {3,1,3,2,1}; sort(...); unique(...);` **fără** `erase`? Dar cu `erase`?

**9.** Care cost este mai mare: pentru $10^6$ numere, `map<int,int>` sau `vector<int>` de frecvență (valorile sunt până la $10^6$)? Care e mai rapid și de ce?

**10.** Cum numeri câți elevi distincți (după nume) apar într-o listă de $10^5$ nume? Ce container folosești?

## 12. Mini-grile

**1.** Ce cost are `insert` într-un `set` cu n elemente? a) O(1) b) O(log n) c) O(n) d) O($n^2$)

**2.** Un comparator pentru `sort` trebuie să returneze `true` când: a) `a` și `b` sunt egale b) `a` trebuie să fie înaintea lui `b` c) `a` este mai mare decât `b`, mereu d) `b` este înaintea lui `a`

**3.** `priority_queue<int, vector<int>, greater<int>>` are în vârf: a) cel mai mare element b) cel mai mic element c) primul introdus d) ultimul introdus

**4.** Ce face `frecv[7]` pe un `map<int,int>` în care 7 nu există? a) eroare b) întoarce 0 fără modificări c) creează cheia 7 cu valoarea 0 d) întoarce −1

## 13. Mai departe (pentru cine vrea mai mult)

- **`multiset`** ține și dublurile; `ms.erase(ms.find(x))` șterge o singură apariție, iar `ms.erase(x)` le șterge pe toate.
- **`set` și `map` cu comparator propriu:** `set<Elev, Comparator>`.
- **Ordine totală pe puncte:** `pair<int,int>` este deja comparabil; pentru puncte în plan este cea mai folosită alegere.
- **`tuple<int,int,int>`** și `tie(a, b, c) = ...` pentru sortări pe mai multe criterii.
- **Citirea rapidă manuală** cu `getchar_unlocked` sau `fread`, utilă la $10^7$ numere.

## Rezumat

- STL îți dă structuri gata făcute: alegerea corectă și cunoașterea costurilor sunt esențiale.
- `sort` cu comparator: întoarce `true` dacă **primul** trebuie înaintea celui de-al doilea; folosește `<` / `>`, niciodată `<=`.
- `map` pentru frecvențe cu valori mari, `set` pentru valori distincte și „cel mai apropiat”, `priority_queue` pentru minim/maxim repetat.
- `lower_bound` pe vector sortat și `set.lower_bound` costă O(log n).
- Citire rapidă: `ios::sync_with_stdio(false); cin.tie(nullptr);` și `'\n'`.

## Probleme de lucru (concurs)

Probleme de pe infoarena, CSES, Kilonova sau pbinfo, în ordinea dificultății. Trimite soluțiile pe platformă și bifează ce ai terminat.

- [ ] `infoarena` **algsort** — ușoară
  sortare simplă: verifică-ți comparatorul și citirea rapidă
  https://www.infoarena.ro/problema/algsort

- [ ] `CSES` **Distinct Numbers** — ușoară
  câte valori distincte: set sau sort + unique
  https://cses.fi/problemset/task/1621

- [ ] `CSES` **Sum of Two Values** — medie
  map sau sort + doi pointeri (lecția 2)
  https://cses.fi/problemset/task/1640

- [ ] `CSES` **Concert Tickets** — medie
  multiset + lower_bound
  https://cses.fi/problemset/task/1091

## Temă (concurs)

Mai grele: încearcă-le după ce le-ai terminat pe cele de mai sus.

- [ ] `infoarena` **hashuri** — medie
  inserare / ștergere / căutare: set, unordered_set sau hash propriu
  https://www.infoarena.ro/problema/hashuri

- [ ] `infoarena` **heapuri** — medie
  heap cu ștergere: priority_queue + marcarea elementelor șterse
  https://www.infoarena.ro/problema/heapuri

## Citește și exersează în plus

- cppreference — Containers library — toate containerele STL, cu complexitatea fiecărei operații (engleză) https://en.cppreference.com/w/cpp/container
- cppreference — std::sort — ce cere un comparator corect (engleză) https://en.cppreference.com/w/cpp/algorithm/sort
- CP-Algorithms — colecția de referință pentru algoritmi de concurs (engleză) https://cp-algorithms.com/

# Răspunsuri

## Exersează pe hârtie

**1.** `2 apare de 2 ori` și `7 apare de 3 ori` (în ordinea crescătoare a cheilor).

**2.** 3 elemente, în ordinea 2, 4, 9 (dublura 4 nu se adaugă).

**3.** `lower_bound(8)` → 10 (primul ≥ 8); `lower_bound(10)` → 10; `upper_bound(10)` → `end()` (niciun element mai mare).

**4.** 1 + 2 = 3 (cost 3, rămân 3 și 3); 3 + 3 = 6 (cost 6). Total: **9**.

**5.** `bool f(int a, int b) { if (a % 10 != b % 10) return a % 10 < b % 10; return a > b; }`

**6.** Pentru două valori egale întoarce `true` în ambele sensuri; `sort` cere o ordine strictă și poate ieși din vector sau cicla.

**7.** `top()` = 8; după `pop()` → 5.

**8.** Fără `erase`: **5** (vectorul își păstrează lungimea; primele elemente sunt 1 2 3, iar restul sunt valori nedefinite). Cu `erase`: **3**.

**9.** Vectorul de frecvență este mai rapid: acces O(1), fără arbore; `map` are un factor log n și memorie dinamică. Cu valori până la $10^6$, alegi vectorul.

**10.** `set<string>`: inserezi toate numele și afișezi `s.size()` (O(n log n)); sau `sort` + `unique`.

## Mini-grile

**1.** b) O(log n). **2.** b). **3.** b) cel mai mic. **4.** c) creează cheia 7 cu valoarea 0.