AVANSAT · OLIMPIADĂ · LECȚIA 7

# Arbori indexați binar (Fenwick): sume pe interval cu actualizări

Durată: 70 min  Nivel: olimpiadă / informatică intensiv  Înainte: lecțiile co-03, xi-10

**CE VEI ȘTI DUPĂ LECȚIE**

- Înțelegi de ce sumele parțiale nu ajung când valorile se modifică.
- Folosești operația lowbit (i & -i) și explici ce acoperă fiecare t[i].
- Implementezi update și query în O(log n).
- Numeri inversiunile unui șir cu un arbore Fenwick.

## 1. Problema: sume cu modificări

Ai un șir de n numere și m operații de două feluri:

- **update:** `v[i] += x`;
- **query:** suma elementelor `v[l..r]`.

Cu **sume parțiale** (lecția co-03) query costă O(1), dar un update obligă la recalcularea a până la n sume: O(n). Fără sume parțiale, query costă O(n). În ambele cazuri, $10^5$ operații pe n = $10^5$ înseamnă $10^{10}$: prea mult.

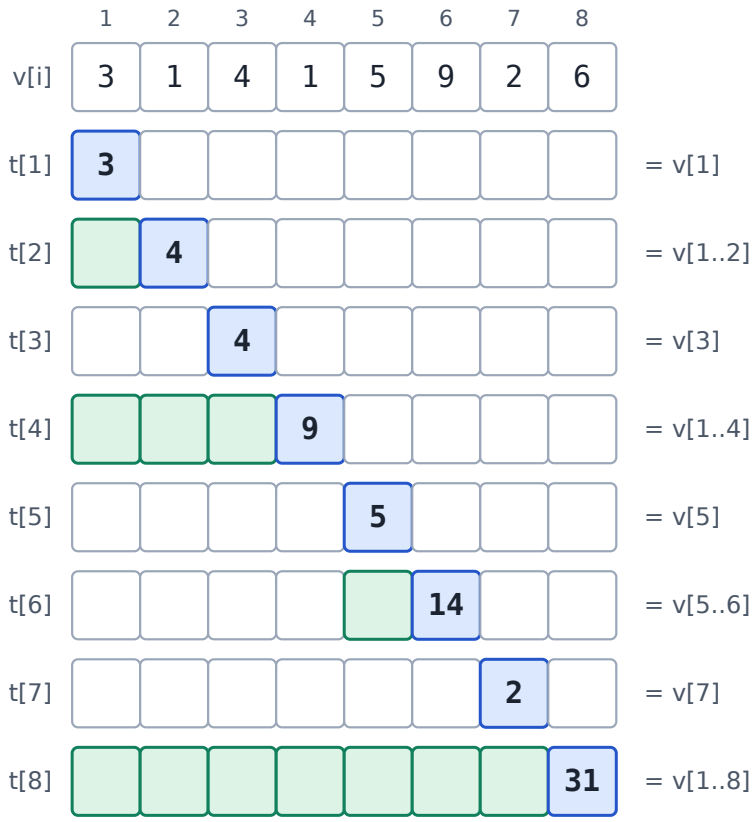
Arborele indexat binar (**Fenwick**, sau **BIT**, *Binary Indexed Tree*) face **ambele** operații în **O(log n)**, cu un singur vector și cod foarte scurt.

## 2. Ideea: fiecare t[i] acoperă o bucată de lungime putere a lui 2

Notăm **lowbit(i)** = cea mai mică putere a lui 2 din scrierea binară a lui i = „cel mai din dreapta bit de 1". În C++: `i & -i`.

| i | binar | lowbit(i) |
|---|---|---|
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 1 |
| 4 | 0100 | 4 |
| 6 | 0110 | 2 |
| 8 | 1000 | 8 |

Vectorul Fenwick `t` este definit astfel: **`t[i]` = suma celor lowbit(i) elemente care se termină la poziția i**, adică `v[i − lowbit(i) + 1 .. i]`.




albastru = poziția i; verde = elementele acoperite de t[i]

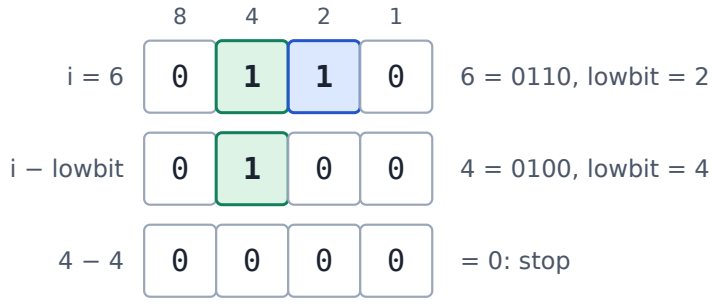
*Ce acoperă fiecare t[i]*

**DE CE AȘA?**

**De ce `i & -i` dă lowbit?** În complement față de 2, `-i` este `~i + 1`: toți biții se inversează, iar „+1" propagă un transport până la primul 0 de la dreapta, adică până la ultimul 1 al lui i. Rezultatul: biții de la dreapta lui acel 1 rămân 0, el rămâne 1, iar cei din stânga sunt inversați. Deci `i & -i` păstrează **doar** ultimul bit de 1. De exemplu, `6 = 0110`, `-6 = …1010`, `6 & -6 = 0010 = 2`.

## 3. Query: suma pe prefix

Suma primelor i elemente se obține adunând blocuri: `t[i]`, apoi blocul care se termină chiar înainte de el, adică `t[i − lowbit(i)]`, și tot așa până la 0.




query(6) = t[6] + t[4]

*query(6)*

De exemplu, `query(7) = t[7] + t[6] + t[4] = 2 + 14 + 9 = 25`, care chiar este 3+1+4+1+5+9+2 = 25. La fiecare pas dispare un bit de 1, deci sunt cel mult $\log_2 n$ pași.

Suma pe intervalul `[l, r]` este `query(r) − query(l − 1)`, ca la sumele parțiale.

## 4. Update: adaugă x la poziția i

Elementul `v[i]` se află în `t[i]` și în toate blocurile mai mari care îl conțin. Acestea sunt `i`, `i + lowbit(i)`, apoi mai departe în același fel, până se depășește n.

Pentru `update(3, +5)`: se modifică `t[3]`, `t[4]` (3 + 1), `t[8]` (4 + 4). Sunt exact blocurile care conțin poziția 3 (vezi figura de mai sus: `t[3]`, `t[4]` și `t[8]` o acoperă).

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

int n;
long long t[100005];                        // arborele Fenwick, indexat de la 1

void adauga(int i, long long x)             // v[i] += x
{
    for (; i <= n; i += i & -i)             // sărim la următorul bloc care conține i
        t[i] += x;
}

long long prefix(int i)                     // suma v[1..i]
{
    long long s = 0;
    for (; i > 0; i -= i & -i)              // ștergem cel mai din dreapta bit de 1
        s += t[i];
    return s;
}

int main()
{
    int q;
    cin >> n >> q;
    for (int i = 1; i <= n; i++)
    {
        long long x;
        cin >> x;
        adauga(i, x);                       // construim arborele element cu element
    }
    while (q--)
    {
        int tip;
        cin >> tip;
        if (tip == 1)
        {
            int i;
            long long x;
            cin >> i >> x;
            adauga(i, x);
        }
        else
        {
            int l, r;
            cin >> l >> r;
            cout << prefix(r) - prefix(l - 1) << '\n';
        }
    }
    return 0;
}
```

| Operația | Ce se întâmplă | Rezultat |
|---|---|---|
| `2 1 7` | prefix(7) − prefix(0) = 25 − 0 | 25 |
| `1 3 5` | t[3], t[4], t[8] cresc cu 5 | – |

| Operația | Ce se întâmplă | Rezultat |
|---|---|---|
| `2 1 7` | prefix(7) = t[7] + t[6] + t[4] = 2 + 14 + 14 | 30 |
| `2 3 4` | prefix(4) − prefix(2) = 14 − 4 | 10 |
| `2 1 8` | prefix(8) = t[8] = 36 | 36 |

**DE CE AȘA?**

**De ce indexăm de la 1?** Pentru că lowbit(0) = 0: bucla `i += i & -i` ar rămâne pe loc pentru totdeauna. **De ce `long long` ?** Suma a $10^5$ valori de până la $10^9$ depășește `int`. **Construirea.** Cu n apeluri `adauga` costă O(n log n); există și o variantă O(n) (fiecare `t[i]` se adaugă la `t[i + lowbit(i)]`), dar rar contează.

**ATENȚIE — GREȘELI FRECVENTE**

Nu confunda **valoarea** `v[i]` cu `t[i]`. Dacă ai nevoie de `v[i]` nu poți citi `t[i]`: ține și vectorul `v`, sau calculează `prefix(i) − prefix(i − 1)`.

## 5. Aplicație: numărarea inversiunilor

O **inversiune** este o pereche de poziții `i < j` cu `v[i] > v[j]`. Verificarea tuturor perechilor este $O(n^2)$. Cu Fenwick, parcurgem șirul de la stânga la dreapta: pentru fiecare element `v[j]` numărăm câte elemente **deja văzute** sunt strict mai mari, apoi îl marcăm ca văzut.

Valorile pot fi mari, așa că le înlocuim cu **rangul** lor în șirul sortat (de la 1 la numărul valorilor distincte): rangurile păstrează ordinea.

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

int m;
int t[100005];

void adauga(int i)                          // am văzut o valoare cu rangul i
{
    for (; i <= m; i += i & -i) t[i]++;
}

int prefix(int i)                           // câte valori văzute au rangul <= i
{
    int s = 0;
    for (; i > 0; i -= i & -i) s += t[i];
    return s;
}

int main()
{
    int n;
    cin >> n;
    vector<int> v(n);
    for (auto &x : v) cin >> x;

    vector<int> sortat = v;                 // rangurile: poziția în șirul sortat, fără
dubluri
    sort(sortat.begin(), sortat.end());
    sortat.erase(unique(sortat.begin(), sortat.end()), sortat.end());
    m = sortat.size();

    long long inversiuni = 0;
    for (int j = 0; j < n; j++)
    {
        int rang = lower_bound(sortat.begin(), sortat.end(), v[j]) - sortat.begin() + 1;
        inversiuni += j - prefix(rang);     // văzute până acum: j; cele <= v[j]:
prefix(rang)
        adauga(rang);
    }
    cout << inversiuni;
    return 0;
}
```

Pe șirul `3 5 1 4 2` (rangurile coincid cu valorile), pas cu pas:

| j | v[j] | văzute | ≤ v[j] | strict mai mari | total |
|---|---|---|---|---|---|
| 0 | 3 | 0 | 0 | 0 | 0 |
| 1 | 5 | 1 | 1 | 0 | 0 |
| 2 | 1 | 2 | 0 | 2 | 2 |
| 3 | 4 | 3 | 2 | 1 | 3 |

| j | v[j] | văzute | ≤ v[j] | strict mai mari | total |
|---|---|---|---|---|---|
| 4 | 2 | 4 | 1 | 3 | 6 |

Total: **6** inversiuni: (3,1), (3,2), (5,1), (5,4), (5,2), (4,2).

> **DE CE AȘA?**
>
> **De ce `j - prefix(rang)` ?** Până la pasul j am marcat exact j elemente. `prefix(rang)` le numără pe cele cu rang ≤ rangul curent (care **nu** formează inversiune, nici cele egale). Restul, `j - prefix(rang)`, sunt strict mai mari. **De ce rang și nu valoarea?** Fenwick este un vector indexat; valori până la $10^9$ nu încap. Comprimarea păstrează doar ordinea, care este tot ce contează pentru comparații. **Complexitate:** sortare O(n log n) plus n căutări binare și n operații Fenwick: O(n log n).

## 6. Greșeli frecvente

| Greșeala | Ce se întâmplă | Corect |
|---|---|---|
| indexare de la 0 | buclă infinită (lowbit(0) = 0) | indexare de la 1 |
| `i += i & -i` fără `i <= n` | acces în afara vectorului | condiția `i <= n` |
| `int` pentru sume mari | depășire | `long long` |
| `prefix(l - 1)` cu `l = 0` | index negativ | indexare de la 1, deci `l >= 1` |
| citești `v[i]` din `t[i]` | valoare incorectă | ține vectorul v separat |
| nu comprimi valorile mari | `t` imens sau acces invalid | rang cu sort + lower_bound |
| Fenwick pentru minim/maxim cu update oarecare | greșit: se pot actualiza doar creșteri/scăderi consecvente | folosește arbore de intervale (lecția co-08) |

## 7. La olimpiadă

Fenwick este soluția „scurtă și rapidă” pentru: sume pe prefix cu modificări, inversiuni, numărarea elementelor mai mici până la un punct, „al k-lea cel mai mic element” (cu căutare binară peste biți), și variante 2D. Când problema cere **minim/maxim pe interval** sau **update pe interval cu query pe interval**, treci la arborele de intervale (co-08).

## 8. Exersează pe hârtie

**1.** Calculează lowbit pentru 12, 16, 20, 7.

**2.** La n = 8, ce blocuri se adună la `prefix(5)` ?

**3.** La n = 8, ce poziții din `t` se modifică la `adauga(5, x)` ?

**4.** La n = 10, ce poziții se modifică la `adauga(6, x)` ?

**5.** Cu `v = 3 1 4 1 5 9 2 6`, calculează `t[6]` și `t[8]`.

**6.** Cum obții suma `v[4..7]` din Fenwick? Câte blocuri adună fiecare `prefix` ?

**7.** Câte inversiuni are `4 3 2 1`? Dar `1 2 3 4`?

**8.** Cum numeri câte elemente **mai mici** decât `x` sunt în șir, la un moment dat?

**9.** De ce Fenwick este mai bun decât sumele parțiale când sunt update-uri? Și când nu sunt?

**10.** Estimează numărul de operații pentru n = m = $2 \cdot 10^5$ cu Fenwick, față de varianta naivă.

## 9. Mini-grile

**1.** `lowbit(i)` se calculează cu: a) `i & (i - 1)` b) `i & -i` c) `i | -i` d) `i ^ -i`

**2.** Complexitatea `prefix` și `adauga` într-un Fenwick cu n elemente este: a) O(1) b) O(n) c) O(log n) d) O(n log n)

**3.** `adauga(i, x)` se oprește când: a) i = 0 b) i > n c) i = n d) t[i] = 0

**4.** Pentru valori până la $10^9$ în problema inversiunilor se folosește: a) un vector de $10^9$ elemente b) comprimarea valorilor (rang) c) doar `long long` d) recursivitate

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

- **Update pe interval, query de punct:** Fenwick peste vectorul de **diferențe** (lecția co-03).
- **Update pe interval și query pe interval:** două arbori Fenwick.
- **Fenwick 2D:** pe matrice, cu două bucle imbricate în `adauga` și `prefix`.
- **Al k-lea element:** coborâre binară pe biții lui n.
- **Arbore de intervale** (lecția următoare): mai general (minim, maxim, cmmdc, update pe interval).

## Rezumat

- Fenwick = vector `t`; `t[i]` = suma ultimelor `lowbit(i)` elemente care se termină în i.
- `lowbit(i) = i & -i`.
- `prefix(i)`: `i -= lowbit(i)` până la 0; `adauga(i, x)`: `i += lowbit(i)` până la n.
- Ambele O(log n); indexare de la 1.
- Aplicație: inversiuni cu comprimarea valorilor, O(n log 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` **aib** — ușoară
  problema clasică: update, sumă pe prefix, căutare a poziției
  https://www.infoarena.ro/problema/aib

- [ ] `CSES` **Dynamic Range Sum Queries** — ușoară
  update de punct, sumă pe interval
  https://cses.fi/problemset/task/1648

- [ ] CSES **Forest Queries II** — dificilă
  Fenwick 2D: extinderea ideii pe matrice
  https://cses.fi/problemset/task/1739

## Temă (concurs)

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

- [ ] CSES **Salary Queries** — dificilă
  normalizare + Fenwick, offline
  https://cses.fi/problemset/task/1144

- [ ] CSES **Range Update Queries** — medie
  update pe interval, citire punct: Fenwick peste diferențe
  https://cses.fi/problemset/task/1651

## Citește și exersează în plus

- CP-Algorithms — Fenwick Tree — articol de referință, cu variante (engleză) https://cp-algorithms.com/data_structures/fenwick.html
- USACO Guide — Point Update Range Sum — Fenwick și arbori de intervale, cu probleme graduate (engleză) https://usaco.guide/gold/point-update-range-sum

# Răspunsuri

## Exersează pe hârtie

**1.** lowbit(12) = 4 (1100), lowbit(16) = 16, lowbit(20) = 4 (10100), lowbit(7) = 1.

**2.** 5 → 4 → 0: `t[5] + t[4]`.

**3.** 5 → 6 → 8: se modifică `t[5]`, `t[6]`, `t[8]`.

**4.** 6 → 8 → 16 > 10: se modifică `t[6]` și `t[8]`.

**5.** `t[6] = v[5] + v[6] = 14`; `t[8] = suma tuturor = 31`.

**6.** `prefix(7) − prefix(3)`. `prefix(7)` adună `t[7], t[6], t[4]` (3 blocuri: 7 → 6 → 4 → 0); `prefix(3)` adună `t[3], t[2]` (2 blocuri: 3 → 2 → 0).

**7.** `4 3 2 1`: 6 inversiuni (toate perechile); `1 2 3 4`: 0.

**8.** `prefix(rang(x) − 1)` (valorile cu rang mai mic decât al lui x) după ce ai marcat elementele văzute.

**9.** Cu update-uri, sumele parțiale costă O(n) la fiecare modificare, Fenwick doar O(log n). Fără update-uri, sumele parțiale sunt mai bune: O(1) pe query după O(n) precalculare.

**10.** Fenwick: $2 \cdot 10^5 \cdot 18 \approx 3{,}6 \cdot 10^6$ operații. Varianta naivă: $2 \cdot 10^5 \cdot 2 \cdot 10^5 = 4 \cdot 10^{10}$ (prea lent).

## Mini-grile

**1.** b) `i & -i`. **2.** c) O(log n). **3.** b) `i > n`. **4.** b) comprimarea valorilor.