Module 10 of 12

Segment trees and Fenwick trees

Range queries with updates in O(log n): a segment tree for sums and minimums, a Fenwick tree in ten lines with i & -i, and counting inversions by value.

Prefix sums answer a range in O(1) only while the array stands still. The moment updates and queries interleave — the shape of most data-structure problems at n, q = 2×10^5 — you need both operations in O(log n).

The lesson builds the two structures contests use for that. A **segment tree** keeps the answer for every half-split range of the array, so a point update rewrites one leaf and its ancestors and a range query collects a handful of whole nodes; it works for any associative operation, minimum and maximum included. A **Fenwick tree** keeps prefix sums in one array where cell `i` covers the `i & -i` positions ending at `i` — ten lines, and the tool for counting when it is indexed by value instead of position.

The exercises are a segment tree answering sums and minimums under point updates and an inversion count with a Fenwick tree over compressed values. The timed checkpoint adds range add with range sum on two Fenwick trees.

CONTENTS

1	Segment trees and Fenwick trees	15 min
✓	Checkpoint — Segment trees and Fenwick trees	60 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 15 MIN

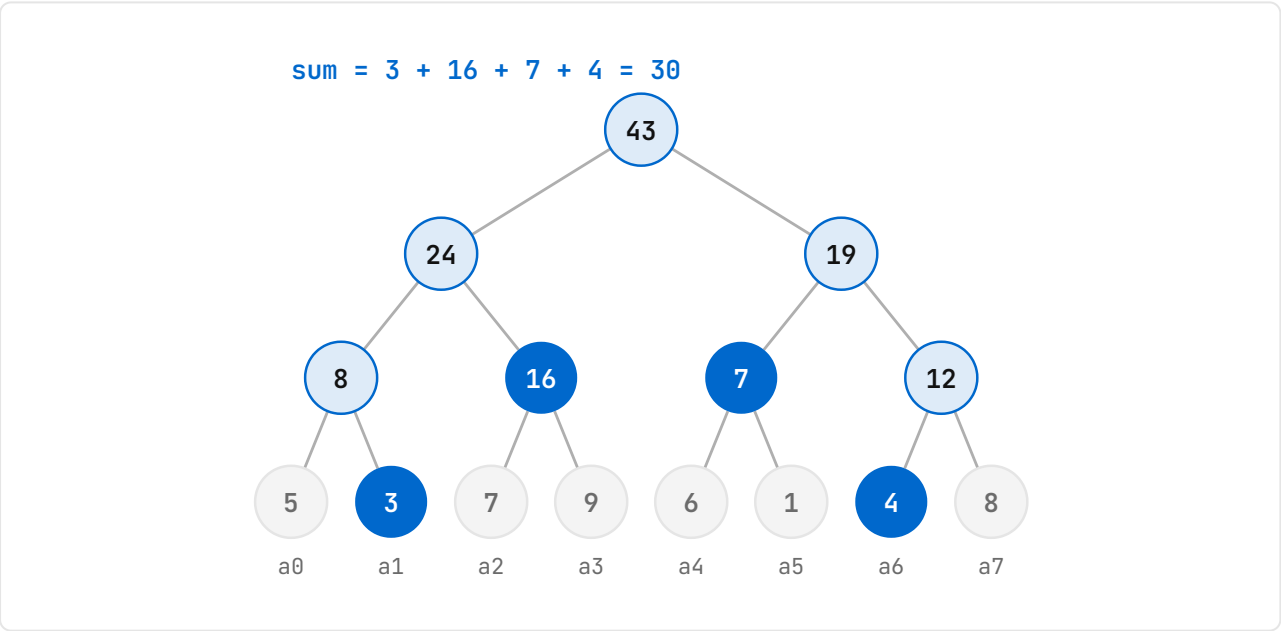
Segment trees and Fenwick trees

Prefix sums answer a range sum in O(1) — until the array changes. One update rebuilds O(n) of the prefix array, so 2×10^5 updates and queries cost 4×10^{10} steps. The fix is a structure in which an update and a query are both O(log n).

A segment tree: one node per range

- The root covers the whole array; each node splits its range in half between two children; the leaves are single elements.
- Every node stores the answer for its range — a sum here, but minimum, maximum and GCD work the same way (any associative operation).
- One array holds it: node `v` has children `2v` and `2v + 1`, and `4n` slots is always enough.

A segment tree: a point update climbs, a range query collects



1. Each node stores the sum of its range. The leaves are a = [5, 3, 7, 2, 6, 1, 4, 8]; each parent adds its two children, and the root holds 36, the whole array.
2. Point update a3 = 9: the leaf changes from 2 to 9. Only its ancestors cover position 3, one per level.
3. Recompute node [2..3] from its children: 7 + 9 = 16.
4. Recompute node [0..3] from its children: 8 + 16 = 24.
5. Recompute node [0..7] from its children: 24 + 19 = 43. 4 nodes changed in all — O(log n).
6. Query sum of a1..a6. The recursion splits the nodes that are only partly inside — [0..7], [0..3], [0..1], [4..7], [6..7] — and stops at nodes fully inside or fully outside.
7. The whole nodes it collects tile the range exactly: [1] = 3, [2..3] = 16, [4..5] = 7, [6] = 4. Their sum is 30, from 4 nodes instead of 6 leaves.

- **Update:** change the leaf, then recompute each ancestor from its two children — one node per level, O(log n).
- **Query `[l, r]`:** start at the root. A node fully inside the range gives its value, one fully outside gives the identity (0 for a sum, +∞ for a minimum), and a partial one asks both children. At most four nodes per level are visited.

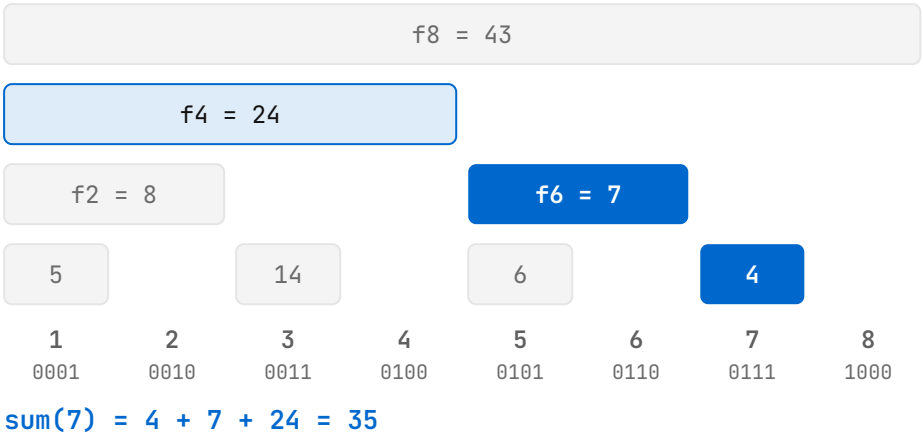
```
void update(int v, int lo, int hi, int pos, ll val) {
    if (lo == hi) { t[v] = val; return; }
    int mid = (lo + hi) / 2;
    if (pos <= mid) update(2 * v, lo, mid, pos, val);
    else update(2 * v + 1, mid + 1, hi, pos, val);
    t[v] = t[2 * v] + t[2 * v + 1];
}
ll query(int v, int lo, int hi, int l, int r) {
    if (r < lo || hi < l) return 0;            // outside: the identity
    if (l <= lo && hi <= r) return t[v];       // inside: the whole node
    int mid = (lo + hi) / 2;
    return query(2 * v, lo, mid, l, r) + query(2 * v + 1, mid + 1, hi, l, r);
}
```

A Fenwick tree: prefix sums in ten lines

When the operation can be undone — sums, counts, XOR — a Fenwick (binary indexed) tree is shorter and faster.

- Cell `i` (1-indexed) holds the sum of the `i & -i` elements ending at `i`. `i & -i` is the lowest set bit: 6 = 110₂ gives 2, 8 gives 8.
- `add(i, x)` climbs `i += i & -i`, touching every cell whose range covers `i`.
- `sum(i)` descends `i -= i & -i`, collecting disjoint ranges that tile 1..i. A range is `sum(r) - sum(l - 1)`.

A Fenwick tree: add climbs with i += i & -i, sum descends with i -= i & -i



1. Each cell i covers the i & -i positions ending at i: cell 6 (110 in binary) covers 5..6, cell 8 covers 1..8. Odd cells hold one value; the bars are the tree.
2. add(3, +7) at i = 3: f3 becomes 7 + 7 = 14. Then i += i & -i = 1 gives 4.
3. add(3, +7) at i = 4: f4 becomes 17 + 7 = 24. Then i += i & -i = 4 gives 8.
4. add(3, +7) at i = 8: f8 becomes 36 + 7 = 43. Then i += i & -i = 8 gives 16, past n = 8: done after 3 cells.
5. sum(7) at i = 7: take f7 = 4, which covers position 7. Then i -= i & -i gives 6.
6. sum(7) at i = 6: take f6 = 7, which covers 5..6. Then i -= i & -i gives 4.
7. sum(7) at i = 4: take f4 = 24, which covers 1..4. Then i -= i & -i gives 0: the cells tile 1..7 and the prefix sum is 35.

```
struct BIT {
    int n; vector<ll> f;
    BIT(int n) : n(n), f(n + 1, 0) {}
    void add(int i, ll x) { for (; i ⩽ n; i += i & -i) f[i] += x; }
    ll sum(int i) { ll s = 0; for (; i > 0; i -= i & -i) s += f[i]; return s;
};
```

The classic trick is to index the tree by **value** instead of position. Scanning left to right, "how many earlier elements are greater than x" is `seen - sum(rank(x))`, which counts inversions in O(n log n). Compress values to ranks 1..m first with sort, `unique` and `lower_bound`.

More practice

- Range Sum Query - Mutable
- Count of Smaller Numbers After Self
- Reverse Pairs
- Count of Range Sum
- Create Sorted Array Through Instructions

EXERCISE 1

Point updates, range sums and minimums

Read an array of `n` integers and answer `q` operations, each one of:

- `1 i v` — set `a[i] = v`;
- `2 l r` — print the sum and the minimum of `a[l..r]`, separated by a space.

Positions are 1-indexed. Build a **segment tree** that stores both the sum and the minimum of every node's range: each operation in O(log n).

Constraints: `1 ≤ n, q ≤ 2 × 10^5`, `|a[i]|, |v| ≤ 10^9`, `1 ≤ l ≤ r ≤ n`. **Input:** `n q`, the `n` values, then `q` lines. **Output:** one line per type-2 operation.

```
5 5
5 3 7 2 6
2 1 5
2 2 4
1 4 9
2 2 4
2 4 4
```

prints

```
23 2
12 2
19 3
9 9
```

INPUT 1

```
5 5
5 3 7 2 6
2 1 5
2 2 4
1 4 9
2 2 4
2 4 4
```

OUTPUT 1

```
23 2
12 2
19 3
9 9
```

INPUT 2

```
1 3
-1000000000
2 1 1
1 1 1000000000
2 1 1
```

OUTPUT 2

```
-1000000000 -1000000000
1000000000 1000000000
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int n;
vector<ll> sm, mn;

void build(int v, int lo, int hi, const vector<ll>& a) {
    if (lo == hi) {
        sm[v] = mn[v] = a[lo];
        return;
    }
    int mid = (lo + hi) / 2;
    build(2 * v, lo, mid, a);
    build(2 * v + 1, mid + 1, hi, a);
    sm[v] = sm[2 * v] + sm[2 * v + 1];
    mn[v] = min(mn[2 * v], mn[2 * v + 1]);
}

void update(int v, int lo, int hi, int pos, ll val) {
    if (lo == hi) {
        sm[v] = mn[v] = val;
        return;
    }
    int mid = (lo + hi) / 2;
    if (pos <= mid) update(2 * v, lo, mid, pos, val);
    else update(2 * v + 1, mid + 1, hi, pos, val);
    sm[v] = sm[2 * v] + sm[2 * v + 1];
    mn[v] = min(mn[2 * v], mn[2 * v + 1]);
}

pair<ll, ll> query(int v, int lo, int hi, int l, int r) {
    if (r < lo || hi < l) return {0, LLONG_MAX};
    if (l <= lo && hi <= r) return {sm[v], mn[v]};
    int mid = (lo + hi) / 2;
    auto [s1, m1] = query(2 * v, lo, mid, l, r);
    auto [s2, m2] = query(2 * v + 1, mid + 1, hi, l, r);
    return {s1 + s2, min(m1, m2)};
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int q;
    cin >> n >> q;
    vector<ll> a(n);
    for (auto& x : a) cin >> x;
    sm.assign(4 * n, 0);
    mn.assign(4 * n, LLONG_MAX);
```

```
    build(1, 0, n - 1, a);
    while (q--) {
        int type;
        ll x, y;
        cin >> type >> x >> y;
        if (type == 1) {
            update(1, 0, n - 1, static_cast<int>(x) - 1, y);
        } else {
            auto [s, m] = query(1, 0, n - 1, static_cast<int>(x) - 1, static_cast<ir
            cout << s << ' ' << m << '\n';
        }
    }
    return 0;
}
```

EXERCISE 2

Count inversions with a Fenwick tree

Read `n` integers and print the number of **inversions**: pairs `i < j` with `a[i] > a[j]`.

Compress the values to ranks, then scan left to right with a **Fenwick tree** indexed by rank: before recording `a[j]`, the earlier elements greater than it are `j - sum(rank(a[j]))`. O(n log n) in all.

Constraints: `1 ≤ n ≤ 2 × 10^5`, `1 ≤ a[i] ≤ 10^9` (values may repeat; equal values are not an inversion). **Input:** `n`, then the values. **Output:** the count.

```
5
2 4 1 3 5
```

prints `3` — the pairs (2, 1), (4, 1) and (4, 3).

INPUT 1

```
5
2 4 1 3 5
```

OUTPUT 1

```
3
```

INPUT 2

```
1
7
```

OUTPUT 2

```
0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

struct BIT {
    int n;
    vector<ll> f;
    BIT(int n) : n(n), f(n + 1, 0) {}
    void add(int i, ll x) {
        for (; i ⩽ n; i += i & -i) f[i] += x;
    }
    ll sum(int i) {
        ll s = 0;
        for (; i > 0; i -= i & -i) s += f[i];
        return s;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<ll> a(n);
    for (auto& x : a) cin >> x;
    vector<ll> vals = a;
    sort(vals.begin(), vals.end());
    vals.erase(unique(vals.begin(), vals.end()), vals.end());
    BIT bit(static_cast<int>(vals.size()));
    ll inversions = 0;
    for (int j = 0; j < n; ++j) {
        int r = static_cast<int>(lower_bound(vals.begin(), vals.end(), a[j]) - vals.
        inversions += j - bit.sum(r);   // earlier elements strictly greater than a[
        bit.add(r, 1);
    }
    cout << inversions << '\n';
    return 0;
}
```

QUIZ

1. A segment tree over `n = 8` values answers `sum(1, 6)` (0-indexed). Which nodes does the query take whole?

A. `[1]`, `[2..3]`, `[4..5]`, `[6]`

B. `[0..7]` minus `[0]` and `[7]`

C. All eight leaves from 1 to 6

D. `[0..3]` and `[4..7]`

2. What is `12 & -12`?

A. `4`

B. `12`

C. `8`

D. `-12`

3. In a Fenwick tree with `n = 8`, which cells does `add(5, x)` change?

A. 5, 6, 8

B. 5, 4, 0

C. 5 only

D. 1, 2, 4, 8

4. Why can't a Fenwick tree answer range **minimum** as `query(r) - query(l - 1)`, the way it answers sums?

A. A minimum cannot be undone: knowing the minimum of `1..r` and of `1..l-1` does not give the minimum of `l..r`

B. Fenwick trees only store non-negative values

C. `i & -i` is undefined for minimums

D. It can; the subtraction just needs `long long`

ANSWER KEY · SEGMENT TREES AND FENWICK TREES

1. A The range splits into the largest nodes that fit inside it: leaf 1 (its sibling 0 is outside), node `[2..3]`, node `[4..5]` and leaf 6. A segment tree never subtracts, so `[0..7]` minus the ends is not how it works; taking every leaf would be O(n).

2. A 12 is `1100` in binary; in two's complement `-12` is `...0100`, so the AND keeps only the lowest set bit, `100` = 4. Fenwick cell 12 therefore covers the four positions 9..12.

3. A Climb with `i += i & -i`: 5 (lowest bit 1) → 6 (lowest bit 2) → 8 → 16, past n. Those are exactly the cells whose ranges contain position 5. Descending (5 → 4 → 0) is what `sum(5)` does.

4. A Range answers from prefixes need an inverse operation — subtraction for sums, XOR for XOR. Minimum has none: if the prefix minimum sits before `l`, it tells you nothing about `l..r`. That is the segment tree's job.

CHECKPOINT · 60 MIN

Checkpoint — Segment trees and Fenwick trees

A timed checkpoint on the module: segment trees (build, point update, range query, the identity for an outside node), Fenwick trees (`i & -i`, the climb and the descent, ranges as two prefixes), counting with a tree indexed by value, and choosing between them and plain prefix sums.

How it works. Sixty minutes from the moment you start the timer: ten questions and one program. 70% on the questions and the program accepted clears the module.

Before you start, make sure you can answer these from memory:

- Which nodes a query for `[l, r]` takes whole, and what an outside node returns for a sum and for a minimum.
- What `i & -i` is for 12, and which cells `add(5, x)` and `sum(7)` touch when n = 8.
- Why a minimum query cannot be answered as `sum(r) - sum(l - 1)`.
- How compressing values to ranks turns "earlier elements greater than x" into a prefix count.

The program is range add with range sum: two Fenwick trees, every value in `long long`.

EXERCISE

Range add, range sum

Answer `q` operations on an array of `n` integers:

- `1 l r v` — add `v` to every `a[i]` with `l ≤ i ≤ r`;
- `2 l r` — print `a[l] + … + a[r]`.

Positions are 1-indexed. Use **two Fenwick trees** `B1`, `B2`: a range add is `B1.add(l, v)`, `B1.add(r+1, -v)`, `B2.add(l, v·(l-1))`, `B2.add(r+1, -v·r)`, and the prefix sum up to `i` is `B1.sum(i)·i - B2.sum(i)` plus the original prefix.

Constraints: `1 ≤ n, q ≤ 2 × 10^5`, `|a[i]|, |v| ≤ 10^6`. **Input:** `n q`, the values, then `q` lines. **Output:** one line per type-2 operation.

```
5 6
1 2 3 4 5
2 1 5
1 2 4 10
2 1 5
2 3 3
1 1 5 -1
2 4 5
```

prints `15`, `45`, `13` and `17` on four lines.

INPUT 1

```
5 6
1 2 3 4 5
2 1 5
1 2 4 10
2 1 5
2 3 3
1 1 5 -1
2 4 5
```

OUTPUT 1

```
15
45
13
17
```

INPUT 2

```
1 3
0
1 1 1 1000000
1 1 1 1000000
2 1 1
```

OUTPUT 2

```
2000000
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

struct BIT {
    int n;
    vector<ll> f;
    BIT(int n) : n(n), f(n + 2, 0) {}
    void add(int i, ll x) {
        for (; i <= n; i += i & -i) f[i] += x;
    }
    ll sum(int i) {
        ll s = 0;
        for (; i > 0; i -= i & -i) s += f[i];
        return s;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    vector<ll> pre(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        ll x;
        cin >> x;
        pre[i] = pre[i - 1] + x;
    }
    BIT b1(n), b2(n);
    auto prefix = [&](int i) { return pre[i] + b1.sum(i) * i - b2.sum(i); };
    while (q--) {
        int type, l, r;
        cin >> type >> l >> r;
        if (type == 1) {
            ll v;
            cin >> v;
            b1.add(l, v);
            b1.add(r + 1, -v);
            b2.add(l, v * (l - 1));
            b2.add(r + 1, -v * r);
        } else {
            cout << prefix(r) - prefix(l - 1) << '\n';
        }
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. Updates and queries interleave, n = q = 2 × 10^5. Why is rebuilding a prefix-sum array after each update too slow?

A. Each rebuild is O(n), so 2 × 10^5 of them is about 4 × 10^10 steps — far past the ~10^8 a second allows

B. Prefix sums overflow `long long`

C. Prefix sums cannot be stored for n above 10^5

D. It is not too slow; O(n) per update is fine

2. In a recursive segment tree for a **minimum**, what should a node completely outside the query range return?

A. The largest value the type holds (+∞)

B. 0

C. The node's own minimum

D. -1

3. Why are `4n` slots allocated for a recursive segment tree with root 1?

A. The deepest index is below 2^(⌈log2 n⌉ + 1), which is always less than 4n

B. Each element appears in exactly four nodes

C. It stores four values per node

D. 2n would be enough; 4n is only tradition

4. How many nodes does a segment tree query visit, at most, per level?

A. About four

B. Exactly one

C. All nodes at that level

D. log n

5. Fenwick tree, n = 8. Which cells does `sum(7)` read?

A. 7, 6, 4

B. 7, 8

C. 1 through 7

D. 4, 6, 7, 8

6. A Fenwick tree over positions 1..n. What does `sum(r) - sum(l - 1)` give?

A. The sum of `a[l..r]`

B. The sum of `a[l+1..r]`

C. The sum of `a[l..r-1]`

D. Only `a[r] - a[l]`

7. Counting inversions with a Fenwick tree, why compress the values to ranks first?

A. The tree is indexed by value, and values up to 10^9 would need a 10^9-cell array; ranks 1..n keep it at n cells

B. Compression sorts the array, which removes the inversions

C. Fenwick trees cannot hold duplicates

D. It makes the answer fit in `int`

8. Which operation can a plain Fenwick tree answer over a range `[l, r]` with point updates?

A. XOR of `a[l..r]`

B. Maximum of `a[l..r]`

C. GCD of `a[l..r]`

D. Number of distinct values in `a[l..r]`

9. Point update **and** range maximum, n = q = 2 × 10^5. The right tool is…

A. A segment tree

B. A Fenwick tree with subtraction

C. A prefix-maximum array

D. A `std::map` of values

10. In range-add/range-sum with two Fenwick trees, adding `v` to `[l, r]` does what to `B1`?

A. `B1.add(l, v)` and `B1.add(r + 1, -v)`

B. `B1.add(l, v)` for every position in the range

C. `B1.add(r, v)` only

D. `B1.add(l, v * l)`

ANSWER KEY · CHECKPOINT — SEGMENT TREES AND FENWICK TREES

1. A The budget lesson again: q · n = 4 × 10^10. O(log n) per operation is about 2 × 10^5 · 18 ≈ 3.6 × 10^6 steps.

2. A An outside node must not change the combined answer, so it returns the identity of `min`: +∞ (`LLONG_MAX`). Returning 0 would wrongly win whenever every real value is positive.

3. A When n is not a power of two the tree is as deep as one over the next power, up to 2n leaves' worth; heap indexes then reach just under 4n. The bottom-up iterative layout is the one that fits in exactly 2n.

4. A At each level at most two nodes are partially covered (the ones holding l and r), and each sends the query to two children, so about four are visited — O(log n) in all.

5. A Descend with `i -= i & -i`: 7 → 6 → 4 → 0. Cell 7 covers 7, cell 6 covers 5..6, cell 4 covers 1..4: three disjoint ranges tiling 1..7.

6. A `sum(r)` covers 1..r and `sum(l - 1)` covers 1..l−1; the difference is exactly l..r. Using `sum(l)` would drop `a[l]`.

7. A Only the relative order matters for an inversion, so replacing each value by its rank among the distinct values keeps every comparison and shrinks the index range to at most n.

8. A XOR is its own inverse, so `prefix(r) ^ prefix(l - 1)` is the range. Maximum and GCD cannot be undone, and distinct counts need an offline trick — segment-tree territory.

9. A Maximum is associative but not invertible, which is exactly what a segment tree needs and a Fenwick range query cannot use. A prefix-maximum array breaks on the first update that lowers a value.

10. A B1 is the difference array of the additions: +v where the range starts, −v just past where it ends. Its prefix sum at i is how much was added to position i. B2 carries the `v·(l-1)` / `-v·r` corrections.

CodeKairo · Competitive Programming study plan · Module 10: Segment trees and Fenwick trees. Solve the exercises online to track your progress and earn XP.