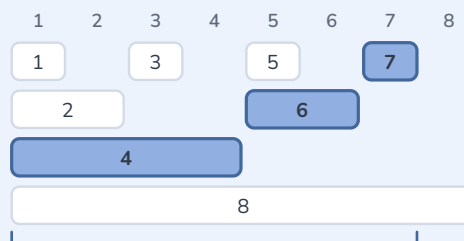


Fenwick and segment trees

Chapter 4.2 · Overall level: Advanced



Chapter 2.1 answered range totals with one subtraction. But what if the data keeps changing between the questions, a reading corrected here, a score updated there? *Can every update and every question both stay fast?*

Two small structures make that possible. A **Fenwick tree** stores cleverly chosen partial sums and needs only a few lines of code; a **segment tree** is a little longer but works for minimum, maximum and any other operation that can be combined in order.

4.2.1 When prefix sums stop working

A prefix array p answers $\text{sum}(l, r) = p[r] - p[l - 1]$ in $O(1)$. Now suppose $a[3]$ changes from 4 to 7. Every prefix that contains position 3, $p[3]$ to $p[n]$, is now 3 too small, and fixing them costs $O(n)$. The plain array has the opposite problem: an update is one assignment, but a sum reads up to n values. With 200,000 updates and 200,000 questions, either choice can reach 4×10^{10} steps.

Structure	Change one value	Sum of a range
Plain array	$O(1)$	$O(n)$
Prefix sums (Chapter 2.1)	$O(n)$	$O(1)$
Fenwick tree or segment tree (this chapter)	$O(\log n)$	$O(\log n)$

The middle ground stores totals of **blocks** of different lengths. A prefix is then a few blocks, and one position belongs to only a few blocks. Choosing the blocks well is the whole trick, and it uses binary numbers.

4.2.2 Binary numbers and the lowest set bit

Every positive integer is a sum of distinct powers of two; its binary digits say which ones. For example $6 = 4 + 2 = 110$ and $7 = 4 + 2 + 1 = 111$. The **lowest set bit** of i , written `lowbit(i)`, is the smallest of those powers: `lowbit(6) = 2`, `lowbit(7) = 1`, `lowbit(8) = 8`.

Computers store negative numbers in *two's complement*: $-i$ is “flip every bit of i , then add 1”.

Flipping turns the trailing zeros of i into ones; adding 1 carries through them and stops exactly at the position of the lowest set bit. So i and $-i$ agree on that bit and disagree on every bit above it, and a bitwise AND keeps exactly the lowest set bit:

```
lowbit(i) = i & (-i)
for example  6 = ...0110,   -6 = ...1010,   6 & -6 = 0010 = 2
```

i	1	2	3	4	5	6	7	8
binary	0001	0010	0011	0100	0101	0110	0111	1000
lowbit(i)	1	2	1	4	1	2	1	8
block stored at i	1..1	1..2	3..3	1..4	5..5	5..6	7..7	1..8

4.2.3 What each Fenwick index stores

A Fenwick tree (also called a binary indexed tree) is an array $T[1..n]$ in which index i stores the total of the block of length `lowbit(i)` that **ends** at i :

$$T[i] = a[i - \text{lowbit}(i) + 1] + \dots + a[i]$$

The indexing is 1-based, and this is not a matter of taste. `lowbit(0) = 0`, so index 0 would store an empty block, and the loops below, which move by `lowbit(i)`, would never leave 0. Positions in the problems are already 1-based; if a program keeps its data 0-based, it must add 1 to every index before it touches T .

For the array $a = [3, 1, 4, 2, 5, 0, 2, 1]$ this gives $T[1..8] = [3, 4, 4, 10, 5, 5, 2, 18]$. For example $T[6]$ covers 5..6, so it is $5 + 0 = 5$, and $T[4]$ covers 1..4, so it is $3 + 1 + 4 + 2 = 10$.

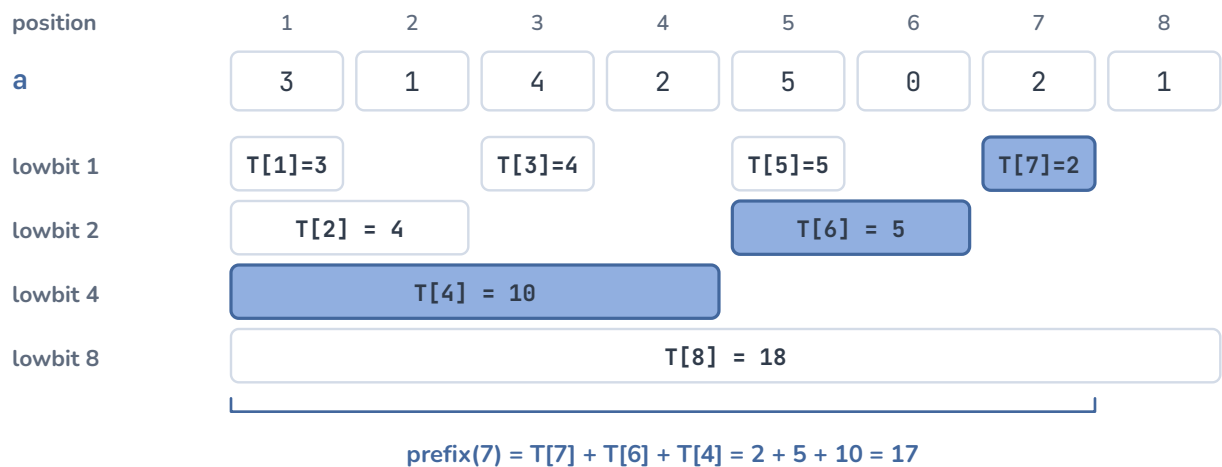


Figure 4.2.1 — A prefix query repeatedly removes the lowest set bit, collecting disjoint stored blocks. Each bar is one stored block with its exact endpoints; the highlighted bars are the ones $\text{prefix}(7)$ reads.

4.2.4 Prefix query: remove the lowest bit

To add up $a[1..i]$, take the block stored at i . It covers $i - \text{lowbit}(i) + 1 \dots i$, so what is still missing is exactly the prefix ending at $i - \text{lowbit}(i)$: continue from there until the index reaches 0.

```

prefix(7):  i = 7 (0111) → take T[7] = 2,  i ← 7 - 1 = 6
            i = 6 (0110) → take T[6] = 5,  i ← 6 - 2 = 4
            i = 4 (0100) → take T[4] = 10, i ← 4 - 4 = 0: stop

```

The total is $2 + 5 + 10 = 17$, which is indeed $3 + 1 + 4 + 2 + 5 + 0 + 2$. **Why it is correct:** each block starts right after the place where the next block ends, so the blocks are disjoint and together cover $1..i$ without gaps. **Why it is fast:** subtracting $\text{lowbit}(i)$ clears one 1-bit of i , and i has at most $\lfloor \log_2 n \rfloor + 1$ such bits, so a prefix query reads at most 18 blocks when $n = 200,000$. A range sum is two prefix queries: $\text{prefix}(r) - \text{prefix}(l - 1)$.

4.2.5 Point update: add the lowest bit

When $a[p]$ increases by d , every stored block that contains position p must increase by d , and no other. Those blocks are found by the mirror-image walk: start at p and repeatedly **add** the lowest set bit until the index passes n .

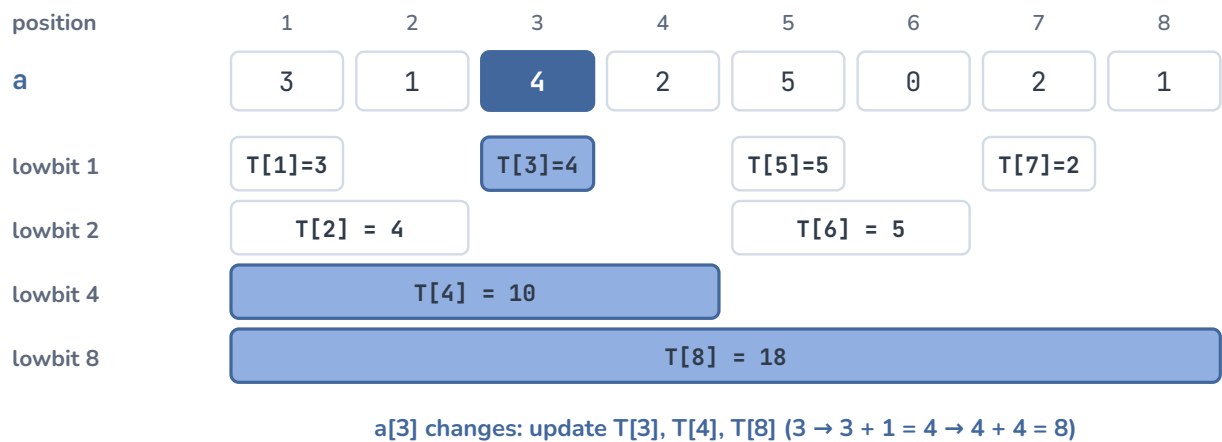


Figure 4.2.2 — An update at position 3 walks up through the blocks that contain it: 3, then $3 + 1 = 4$, then $4 + 4 = 8$.

Why exactly these blocks. The block at p contains p . Every index strictly between p and $p + \text{lowbit}(p)$ has a smaller lowest bit than p , so its block is too short to reach back to p ; the index $j = p + \text{lowbit}(p)$ has a larger lowest bit, and its block reaches back past the start of p 's block. So j is the next block containing p , and the reasoning repeats from j . Each step at least doubles the lowest bit, so there are at most about $\log_2 n$ steps. For example, adding 3 to $a[3]$ makes $T[3] = 7$, $T[4] = 13$ and $T[8] = 21$, and $\text{prefix}(7)$ becomes $2 + 5 + 13 = 20$.

Assignments become additions

Fenwick updates *add*. A practice problem that *assigns* $a[p] = x$ must keep a copy of the current values and add the difference: $\text{add}(p, x - a[p])$, then $a[p] \leftarrow x$. Adding x itself would leave the old value inside every block. To build, start from zeros and add every $a[i]$.

Pseudocode

Fenwick tree: $O(\log n)$ per update and per query

```

T[1..n] ← 0                                (1-based; index 0 is never used)
add(i, d):                                  (a[i] increases by d)
    while i ≤ n:
        T[i] ← T[i] + d
        i ← i + lowbit(i)
prefix(i):                                  (a[1] + ... + a[i])
    total ← 0
    while i > 0:
        total ← total + T[i]
        i ← i - lowbit(i)
    return total

for i from 1 to n: add(i, a[i])
for each operation:
    "1 p x": add(p, x - a[p]); a[p] ← x
    "2 l r": output prefix(r) - prefix(l - 1)

```

4.2.6 Code in four languages

Each program solves the first practice problem, **Live range totals**. The first line holds n and q , the second the n starting values, then q lines “1 p x ” (assign $a[p] = x$) or “2 l r ” (print the sum of positions l to r). Positions start at 1.

Sample input	Sample output
4 4	7
3 1 4 2	10
2 2 4	3
1 3 7	
2 2 4	
2 1 1	

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n, q = int(data[0]), int(data[1])
    fen = [0] * (n + 1)      # 1-based: fen[i] sums a[i - lowbit(i) + 1 .. i]

    def add(i, delta):       # a[i] += delta
        while i <= n:
            fen[i] += delta
            i += i & -i

    def prefix(i):           # a[1] + ... + a[i]
        total = 0
        while i > 0:
            total += fen[i]
            i -= i & -i
        return total

    a = [0] + [int(x) for x in data[2:2 + n]]
    for i in range(1, n + 1):
        add(i, a[i])
    out = []
    for k in range(2 + n, 2 + n + 3 * q, 3):
        t, x, y = int(data[k]), int(data[k + 1]), int(data[k + 2])
        if t == 1:           # assignment a[x] = y becomes an addition
            add(x, y - a[x])
            a[x] = y
        else:
            out.append(prefix(y) - prefix(x - 1))
    print("\n".join(map(str, out)))

main()
```

C++17

```

#include <bits/stdc++.h>
using namespace std;

int n;
vector<long long> fen;      // 1-based: fen[i] sums a[i - lowbit(i) + 1 .. i]

void add(int i, long long delta) {          // a[i] += delta
    for (; i <= n; i += i & -i) fen[i] += delta;
}

long long prefix(int i) {                   // a[1] + ... + a[i]
    long long total = 0;
    for (; i > 0; i -= i & -i) total += fen[i];
    return total;
}

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int q;
    cin >> n >> q;
    vector<long long> a(n + 1);
    fen.assign(n + 1, 0);
    for (int i = 1; i <= n; i++) { cin >> a[i]; add(i, a[i]); }
    while (q--) {
        int t, x; long long y;
        cin >> t >> x >> y;
        if (t == 1) { add(x, y - a[x]); a[x] = y; } // assignment as an addition
        else cout << prefix(y) - prefix(x - 1) << '\n';
    }
}

```

JavaScript (Node)

```

const data = require("fs").readFileSync(0, "utf8")
    .trim().split(/\s+/).map(Number);
const n = data[0], q = data[1];
const fen = new Float64Array(n + 1), a = new Float64Array(n + 1); // 1-based
function add(i, delta) {          // a[i] += delta
    for (; i <= n; i += i & -i) fen[i] += delta;
}
function prefix(i) {              // a[1] + ... + a[i]
    let total = 0;
    for (; i > 0; i -= i & -i) total += fen[i];
    return total;
}
for (let i = 1; i <= n; i++) { a[i] = data[1 + i]; add(i, a[i]); }
const out = [];
for (let k = 2 + n; k < 2 + n + 3 * q; k += 3) {
    const t = data[k], x = data[k + 1], y = data[k + 2];
    if (t === 1) { add(x, y - a[x]); a[x] = y; } // assignment as an addition
    else out.push(prefix(y) - prefix(x - 1));
}
console.log(out.join("\n"));

```

Java 17

```

import java.io.*;

public class Main {
    static int n;
    static long[] fen;      // 1-based: fen[i] sums a[i - lowbit(i) + 1 .. i]

    static void add(int i, long delta) {    // a[i] += delta
        for (; i <= n; i += i & -i) fen[i] += delta;
    }

    static long prefix(int i) {             // a[1] + ... + a[i]
        long total = 0;
        for (; i > 0; i -= i & -i) total += fen[i];
        return total;
    }

    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken(); n = (int) in.nval;
        in.nextToken(); int q = (int) in.nval;
        long[] a = new long[n + 1];
        fen = new long[n + 1];
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            a[i] = (long) in.nval;
            add(i, a[i]);
        }
        StringBuilder out = new StringBuilder();
        for (int k = 0; k < q; k++) {
            in.nextToken(); int t = (int) in.nval;
            in.nextToken(); int x = (int) in.nval;
            in.nextToken(); long y = (long) in.nval;
            if (t == 1) { add(x, y - a[x]); a[x] = y; } // assign via addition
            else out.append(prefix((int) y) - prefix(x - 1)).append('\n');
        }
        System.out.print(out);
    }
}

```

Cost and number sizes. Each of the $n + q$ operations walks at most about 18 indices, so the whole run is $O((n + q) \log n)$ with one array of $n + 1$ numbers. Sums can reach $200,000 \times 10^6 = 2 \times 10^{11}$, beyond 32 bits: C++ and Java use 64-bit integers. In JavaScript the sums are ordinary numbers (exact below 2^{53}) and the bitwise $\&$ is applied only to indices; applied to a 40-bit sum it would silently cut it to 32 bits.

4.2.7 Segment trees: any operation that combines in order

Now ask for the **minimum** of a range instead of the sum. Prefix minima do not help: knowing the minimum of $a[1..r]$ and of $a[1..l-1]$ says nothing about $a[l..r]$, because there is no “subtraction” that removes the first part from a minimum. A Fenwick prefix query has the same problem. We need blocks that can cover any range directly.

A **segment tree** stores one block for every node of a balanced binary tree. The root covers $1..n$; a node covering $lo..hi$ has two children covering $lo..mid$ and $mid+1..hi$, with $mid = (lo + hi) \div 2$; leaves cover single positions. Each node stores the combined value of its block. The combining operation must be **associative**, so that blocks can be grouped in any way, and it needs a **neutral element** that changes nothing: 0 for sums, $+\infty$ for minima, $-\infty$ for maxima.

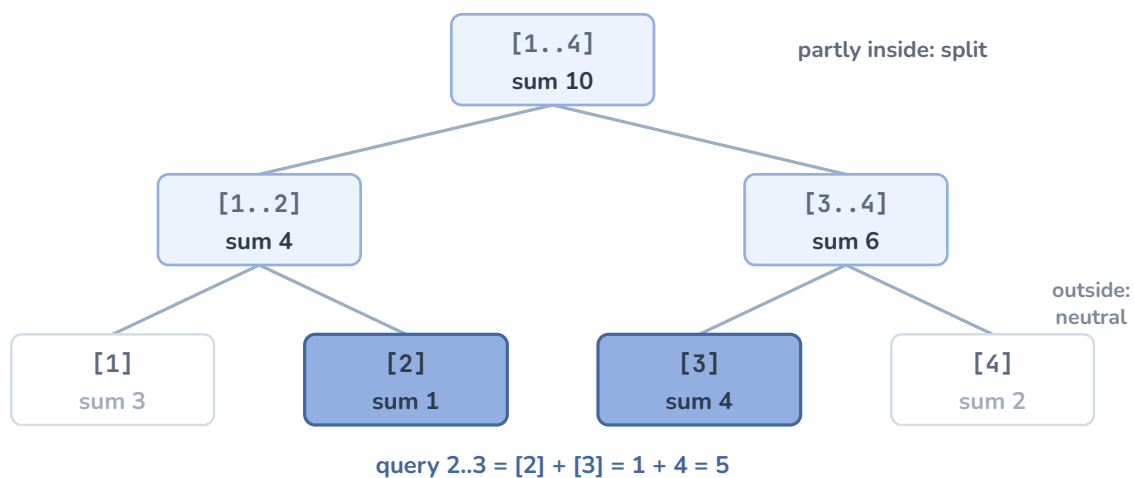


Figure 4.2.3 — A range answer combines disjoint node intervals using the same associative operation. Every node shows its interval and its value; for minima the same tree would store 1, 1, 2 in the top three nodes.

Query $l..r$. Start at the root. A node whose block lies completely inside $l..r$ returns its stored value; a node completely outside returns the neutral element; any other node asks both children and combines their answers. **Update $a[p] = x$.** Replace the leaf for p , then recompute the value of every ancestor from its two children. A segment tree can assign directly, so no difference trick is needed.

Why it is correct. The nodes that return their own value are disjoint and their blocks together are exactly $l..r$, in left-to-right order; associativity lets the answers be combined in any grouping. **Why it is fast.** On each level at most two nodes are only partly inside the range (the ones containing l and r), so at most four nodes per level are visited, about $4 \log_2 n$ in total. An update touches one node per level.

Pseudocode

Segment tree over $a[1..n]$: $O(\log n)$ per query and update

```

NEUTRAL  $\leftarrow +\infty$            combine(x, y)  $\leftarrow \min(x, y)$            (for range minimum)
t[1..4n]           (node 1 is the root; node k has children 2k and 2k + 1)

update(k, lo, hi, p, x):           (a[p]  $\leftarrow$  x)
    if lo = hi: t[k]  $\leftarrow$  x; return
    mid  $\leftarrow$  (lo + hi) div 2
    if p  $\leq$  mid: update(2k, lo, mid, p, x)
    else:       update(2k + 1, mid + 1, hi, p, x)
    t[k]  $\leftarrow$  combine(t[2k], t[2k + 1])

query(k, lo, hi, l, r):
    if r < lo or hi < l: return NEUTRAL           (block outside the range)
    if l  $\leq$  lo and hi  $\leq$  r: return t[k]           (block inside the range)
    mid  $\leftarrow$  (lo + hi) div 2
    return combine(query(2k, lo, mid, l, r), query(2k + 1, mid + 1, hi, l, r))

```

Build the tree by calling `update(1, 1, n, i, a[i])` for every i (after filling `t` with `NEUTRAL`), and answer a range with `query(1, 1, n, l, r)`. Four times n nodes is always enough. The recursion is only about $\log_2 n \approx 18$ calls deep, so here, unlike the tree traversals of Chapter 4.1, recursion is perfectly safe.

The neutral value must really be neutral. If an empty part returned 0 for a minimum, then on the array 3, 1, 4, 2 the query 3..4 would combine 0 from the left half with 2 and answer 0, a value that is not even in the array. Use a number larger than every possible value, such as 2×10^9 when values lie within $\pm 10^9$, or the type's maximum.

4.2.8 Finding the k -th token: a prefix threshold

Let $a[i]$ count tokens at position i , and ask: at which position is the k -th token, counting from position 1? That is the first position p with $\text{prefix}(p) \geq k$. Because the counts are **nonnegative**, prefix sums never decrease, so the positions where $\text{prefix}(p) \geq k$ form a final stretch of the array and their first element is well defined. One could binary search (Chapter 2.3) on p with a Fenwick prefix query at each step, $O(\log^2 n)$ per question. Walking down the Fenwick tree itself is faster:

```

pos  $\leftarrow$  0
for step = largest power of two  $\leq n$ , then step / 2, ..., down to 1:
    if pos + step  $\leq n$  and T[pos + step] < k:
        pos  $\leftarrow$  pos + step           (the k-th token is beyond pos + step)
        k  $\leftarrow$  k - T[pos + step]
answer pos + 1

```

Every value added to pos is a larger power of two than the current $step$, so pos is a multiple of $2 \cdot step$ and $lowbit(pos + step) = step$: the block $T[pos + step]$ is exactly positions $pos + 1 \dots pos + step$. The loop keeps the invariant “the first pos positions hold fewer than the original k tokens, and k is now the rank of the wanted token after position pos ”. If the next block holds fewer than k tokens, the target lies beyond it, so skip it and subtract its tokens; otherwise it lies inside, so keep pos and try a smaller block. Each step costs $O(1)$: $O(\log n)$ in total.

step	block tested	$T[pos + step]$	compare with k	decision	pos, k afterwards
8	1..8	18	$18 \geq 9$	stay	0, 9
4	1..4	10	$10 \geq 9$	stay	0, 9
2	1..2	4	$4 < 9$	skip 2 positions	2, 5
1	3..3	4	$4 < 5$	skip 1 position	3, 1

For counts 3, 1, 4, 2, 5, 0, 2, 1 and $k = 9$ the walk ends with $pos = 3$, so the 9th token is at position 4: the first three positions hold 8 tokens and position 4 holds the next two. With a negative count the prefix sums could go down again, “the first position with prefix $\geq k$ ” would no longer be found by skipping blocks, and the method breaks.

4.2.9 Live subtree sums

Chapter 4.1 showed that the entry order of a rooted tree turns the subtree of v into the interval $tin[v] \dots tin[v] + size[v] - 1$. The tree’s shape never changes, so compute tin and $size$ once, place the reward of v at position $tin[v]$, and maintain that flat array with a Fenwick tree. A subtree question becomes a range sum and a new reward for v becomes a point assignment at $tin[v]$.

With the tree and rewards of Chapter 4.1, the flat array is $[5, -2, 3, 1, 4, 2]$. The subtree of 2 is positions 2..4, total 2. Setting the reward of vertex 4 to 8 adds $8 - 3 = 5$ at $tin[4] = 3$, after which the subtree of 2 totals 7 and the whole tree 18.

4.2.10 Common mistakes

- **Fenwick update at index 0.** $\text{lowbit}(0) = 0$, so $\text{add}(0, d)$ never moves and loops forever. Convert 0-based positions with $+1$ before calling the tree.
- **Zero as the neutral value for a minimum.** Empty parts then return 0, and every query on positive values answers 0. Use $+\infty$ (a value above every possible element).
- **Applying an assignment as an addition.** $\text{add}(p, x)$ adds x on top of the old value. Add $x - a[p]$ and remember the new $a[p]$.
- **32-bit sums.** Totals reach 2×10^{11} ; use 64-bit integers, and in JavaScript never apply bitwise operators to sums.
- **Fenwick trees for operations without an inverse.** A prefix query can only answer a general range if the operation can be undone by subtraction. Minimum and maximum need a segment tree.
- **Threshold search on counts that can be negative.** The descent assumes nondecreasing prefix sums. With negative values there is no monotone threshold to find.

4.2.11 Where these structures stop

This chapter changes one position at a time. Adding a value to a whole range at once, while still asking for range sums, needs a segment tree with *lazy propagation*, which is beyond this chapter. (If only single values are asked for, a Fenwick tree over the difference array of Chapter 2.1 handles range additions: add v at l and $-v$ at $r + 1$, then the value at p is $\text{prefix}(p)$.) If the data never changes, plain prefix sums are simpler and faster. And an operation that is not associative cannot be split into blocks at all.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● Why can a range minimum not, in general, be computed as $P[r] - P[l - 1]$ from some prefix array P ?
2. ● What goes wrong in the prefix-threshold search if counts may be negative?
3. ● Find $\text{lowbit}(12)$ and the block that $T[12]$ stores. Which indices does $\text{prefix}(13)$ read?
4. ●● For $a = [2, 0, 1, 3, 1, 1]$ ($n = 6$), write $T[1..6]$. Then add 2 to $a[2]$: which entries change, and to what?
5. ●● Build a minimum segment tree for $a = [5, 3, 8, 6]$. What is the minimum of positions 3..4? After $a[2] = 9$, which nodes change and what is the new root?
6. ●● For the counts 3, 1, 4, 2, 5, 0, 2, 1, run the descent for $k = 11$.

Answers and explanations

Read these after attempting the exercises.

1. Minima have no subtraction: knowing $\min(a[1..r])$ and $\min(a[1..l-1])$ does not tell you the minimum of the part in between. For $[1, 5, 4]$, both prefixes up to 1 and up to 3 have minimum 1, yet the minimum of positions 2..3 is 4.
2. Prefix sums are no longer monotone, so they can reach k , drop below it and reach it again; skipping a block because its total is below k may skip the true first position.
3. $12 = 1100$, so $\text{lowbit}(12) = 4$ and $T[12]$ stores $a[9..12]$. $\text{prefix}(13)$ reads 13 (1101), then 12 (1100), then 8 (1000), then stops at 0.
4. $T[1..6] = [2, 2, 1, 6, 1, 2]$. The walk from 2 is $2 \rightarrow 4 \rightarrow 8 (> 6, \text{stop})$: $T[2]$ becomes 4 and $T[4]$ becomes 8. The others are unchanged.
5. Leaves 5, 3, 8, 6; $[1..2] = 3$, $[3..4] = 6$, $\text{root}[1..4] = 3$. The minimum of 3..4 is the node $[3..4] = 6$. After $a[2] = 9$ the leaf $[2]$, the node $[1..2]$ (now 5) and the root (now 5) change.
6. step 8: $T[8] = 18 \geq 11$, stay. step 4: $T[4] = 10 < 11$, $\text{pos} = 4$, $k = 1$. step 2: $T[6] = 5 \geq 1$, stay. step 1: $T[5] = 5 \geq 1$, stay. Answer $\text{pos} + 1 = 5$ (the first four positions hold 10 tokens).

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Live range totals

[Apply](#)

Maintain an array. Type 1 $p\ x$ assigns $a[p]=x$. Type 2 $l\ r$ asks the inclusive range sum.

Live range minimum

[Connect](#)

Maintain an array. Type 1 $p\ x$ assigns $a[p]=x$. Type 2 $l\ r$ asks the minimum value in the inclusive range.

Locate the k th token

[Reason](#)

Position i holds a_i tokens. Type 1 $p\ d$ adds d to $a[p]$, and the resulting count is guaranteed nonnegative. Type 2 k asks for the first position whose cumulative token count is at least k . Each such k is guaranteed between 1 and the current total.

Live subtree rewards

[Stretch](#)

Root a tree at 1. Type 1 $v\ x$ assigns vertex v reward x . Type 2 v asks for the sum of rewards in v 's subtree, including v .