

Level 0 Training

Session 6

Static Range Queries

Prefix sums, frequency arrays,
and difference arrays (partial sums)

Store the work you would repeat.
Derive the formula. Check every boundary.

How to use this reference

This book is a reference for studying **after the session**. It explains the meaning of each idea, not only the syntax. Read a little, trace a small example, write the code yourself, and test what happens at the boundaries.

A study routine you can repeat

1. **Explain the state.** Say what each variable or stored entry means. “The total of the first five values” is more useful than “the fifth cell.”
2. **Trace before running.** Use paper to record the input, intermediate values, and output. Write indices beside values; they are not the same thing.
3. **Change one thing.** Try a negative value, a repeated value, a smallest allowed input, or an endpoint. Predict the new result before compiling.
4. **Solve independently.** Attempt the practice, then use its hint. Read a solution only after a genuine attempt. Later, close the book and rebuild the solution from its idea.

How examples are labelled

A **complete program** has headers and its own `main`. Save or compile one program at a time. A **fragment** demonstrates one smaller idea; its placement is stated. A **wrong example** is deliberately incorrect and is never a template to copy.

Pause and predict questions have answers nearby; cover the answer first. **Optional extensions** are a second-reading route. Main topics and practice solutions start on fresh pages so you can return to them easily.

Language and editable sources

The complete programs use C++23. Session 5 additionally labels the examples that require GNU PBDS extension headers. Session 6 uses only standard headers. Printed complete listings are loaded directly from the supplied `.cpp` files. Each standalone handout and the collected book share the same chapter source.

Study in stages

Session 6 assumes you can use loops, vectors, strings, functions, and Big-O notation. The final optional example also uses `map`. Learn ordinary prefix sums first, then frequency arrays, then difference arrays and two-dimensional prefixes. Return to the earlier sessions when a prerequisite feels unfamiliar.

The Level 0 learning path

Session	Topics
1	Intro to problem solving: data types, conditions, and loops.
2	Arrays and strings.
3	Functions and complexity analysis.
4	STL, part 1: vector, pair, deque, queue, priority queue, and stack.
5	STL, part 2: set, multiset, GNU ordered set, map (ordered), and unordered map.
6	Static range queries: prefix sums, frequency arrays, and partial sums / difference arrays.

The six-session series is complete

This booklet contains **Session 6: Static Range Queries**. Its section numbers begin with 6 to match the complete collected book, which now contains all six sessions.

What Session 6 will help you do

Avoid summing the same values repeatedly. Count bounded values without a map. Apply many interval additions before rebuilding an array. Answer rectangle queries. Combine these ideas while explaining each stored entry, each endpoint, and the total work.

A naming distinction: partial sums

A **partial sum** can simply mean a running prefix total. In this training's range-update topic, we also cover the **difference-array technique**: record changes at boundaries, then take running sums to reconstruct values. The C++ function `std::partial_sum` computes running totals; it does not record interval updates for you.

Conventions

Tasks specify valid input; programs do not print prompts or request corrections. Most range tasks use **one-based inclusive endpoints**. A padded vector still has ordinary zero-based C++ indexing; we choose to reserve index 0. Frequency-array indices instead represent values or letter IDs. Each exception is stated explicitly.

Contents

How to use this reference	i
The Level 0 learning path	ii
6 Static Range Queries	1
6.1 Repeated questions about an unchanged array	2
6.2 Indices, endpoints, and the extra zero	4
6.3 Building a prefix sum one value at a time	5
6.4 Answering a range sum by cancellation	7
6.5 Safe totals, boundaries, and reusable helpers	9
6.6 Count the cost and recognize the limitations	11
6.7 Turn a condition into a range count	12
6.8 Fixed-length windows without repeated scans	15
6.9 Prefix and suffix information around a position	17
6.10 Frequency arrays: use a value as an index	19
6.11 Negative values, offsets, and domain safety	21
6.12 Distinct values, modes, and tie-breaking	23
6.13 Small alphabets and anagram checks	25
6.14 Count equal pairs without a nested loop	27
6.15 Prefix sums over the value domain	29
6.16 Letter counts inside substrings	31
6.17 Differences: remember changes instead of values	33
6.18 A range addition changes only two boundaries	35
6.19 Overlapping updates and cancellation	37
6.20 Apply additions to a nonzero starting array	38
6.21 Processing phases: updates first, sums afterwards	40
6.22 Coverage counts on an integer line	42
6.23 Combine differences, a condition, and a prefix	44
6.24 Two-dimensional prefixes: totals from a corner	46
6.25 Rectangle queries and marked-cell counts	47
6.26 Fresh summaries for multiple test cases	50
6.27 Debug by checking meanings and small counterexamples	52
6.28 Optional extension: C++ numeric-library helpers	54
6.29 Optional extension: weighted and modular sums	56
6.30 Optional extension: sparse values and compression	58
6.31 Optional extension: rectangle additions with four markers	60
6.32 Optional challenge: count target-sum subarrays	62
6.33 Practice problems: choose the stored information	64
6.34 Trace, representation, and complexity drills	76
6.35 Hints before solutions	82
6.36 Explained practice solutions	86
6.37 Explained trace and complexity drills	109
6.38 Quick reference, glossary, and readiness checklist	115
6.39 Technical notes and references	119
6.40 Completing Level 0: turn the tools into a method	123

SESSION 6

Static Range Queries

Prefix sums, frequency arrays, and difference arrays (partial sums)

In earlier sessions, you learned to store information and choose a suitable container. This session asks a new question: *Which work am I repeating, and what could I store to avoid repeating it?*

You will build small summaries of an array. One summary remembers cumulative totals. Another remembers how often each value appears. A third remembers where a change starts and stops. None of these requires a new container: vectors and loops are enough.

Three meanings, not three formulas to memorize

Tool	Meaning of one entry	Typical job
Prefix sum	Total of an initial part of a sequence.	Answer many interval sums.
Frequency array	Number of occurrences of one value.	Count bounded values or letters.
Difference array	Change that begins at a boundary.	Apply many interval additions in a batch.

Your route through the chapter

Stage A: cumulative information. Start with a slow scan, derive a prefix sum, and use subtraction to answer queries. Transform conditions into 0/1 values.

Stage B: counts by value. Build a frequency array, then distinguish a range of values from a range of positions. Combine small-alphabet counts with position prefixes.

Stage C: delayed changes and grids. Record range additions at two boundaries, reconstruct the final values, and extend prefix sums to rectangles.

Stage D: practise and explain. Solve the core problems first. Optional sections introduce numeric-library helpers, weighted and modular sums, compressed domains, two-dimensional updates, and target-sum subarrays.

The sentence to write before the code

“Entry `pref[i]` means ...” or “Entry `diff[i]` means ...” Complete this sentence before writing an update. A formula with the wrong meaning can compile successfully and still solve the wrong problem.

6.1 Repeated questions about an unchanged array

What is a range query?

A **range** is a consecutive part of a sequence. A **query** asks for information without necessarily changing that sequence. For example, “What is the sum from position 2 through position 4?” is a range-sum query.

Throughout the core examples, the range $[l, r]$ includes **both** endpoints. On the array below, $[2, 4]$ contains $-2, 7, 0$, so its sum is 5.

Position	1	2	3	4	5
Value	4	-2	7	0	3

Static means the array does not change between these queries. It does not mean every query must be given in advance. Once a suitable summary has been built, we can read and answer one query at a time. The distinction between static queries and later updates is important.^[6-1]

Start with a solution you already understand

Read the array. For every query, start a total at zero and visit the positions from 1 through r . This is a correct algorithm, and it is a useful reference when testing a faster one.

Contract. Read n q , then n signed values, then q ranges. Here $1 \leq n \leq 2000$, $0 \leq q \leq 2000$, $|a_i| \leq 10^9$, and $1 \leq l \leq r \leq n$. Print one sum per query.

Complete program: scan each requested range

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> a(n + 1, 0);
    for (int i = 1; i <= n; ++i) std::cin >> a[i];
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        long long sum = 0;
        for (int i = l; i <= r; ++i) sum += a[i];
        std::cout << sum << '\n';
    }
}
```

Sample input	Sample output
5 3	5
4 -2 7 0 3	12
2 4	7
1 5	
3 3	

Where the repeated work appears

For query $[1, 5]$, the loop adds all five values. A later query $[2, 5]$ adds four of those same values again. With many overlapping queries, much of the work is repeated.

A query of length $r - l + 1$ takes that many additions. The total query work is the sum of the requested lengths, which can be as large as nq . If both n and q are 200000, the upper bound is 40000000000 additions. This is a work count, not a measured number of seconds.

Preprocessing: pay once, reuse many times

Preprocessing means doing useful work before answering queries. Rather than adding the middle section again, we will store totals measured from a common starting point. Subtracting two such totals will leave only the requested section.

Pause and predict: 1

Do we need a prefix array when the only task is to read n numbers and print their total once?

Check your prediction. No. A running total is enough and needs only one extra variable. Prefix storage becomes useful when we need to revisit totals for different boundaries.

6.2 Indices, endpoints, and the extra zero

A prefix index measures a length

Our mathematical input positions are $1, 2, \dots, n$. C++ remains zero-based: we allocate $n + 1$ entries and choose `a[1]` through `a[n]` as the real sequence. Entry `a[0]` is padding.

Define `pref[i]` as **the total of the first i values**. Entry `pref[0] = 0` represents no input values, not the first value. There are $n + 1$ possible prefix lengths, from zero through n .

```
// Inside main: positions 1 through n hold the real sequence.
std::vector<long long> a(n + 1, 0);
std::vector<long long> pref(n + 1, 0);
```

For five inputs, `pref[5]` is valid in this six-entry vector. It would not be valid in a vector of size five: the allocated size, not the variable's name, determines valid indices.

Translate, do not mix, interval conventions

All three formulas below use $P[k]$ = the sum of the first k values.

Requested interval	Elements included	Answer
One-based inclusive $[l, r]$	Positions l through r .	$P[r] - P[l - 1]$
Zero-based inclusive $[L, R]$	Indices L through R .	$P[R + 1] - P[L]$
Zero-based half-open $[L, R)$	Indices L through $R - 1$.	$P[R] - P[L]$

For 4, -2, 7, 0, 3, one-based $[2, 4]$, zero-based $[1, 3]$, and zero-based $[1, 4)$ all identify -2, 7, 0. The endpoints differ because their meanings differ.

A half-open interval $[L, L)$ is empty and has sum zero. An inclusive interval $[l, l]$ contains one element and is **not** empty.

Strings remain zero-based

For a statement using one-based character positions, position i is `text[i - 1]`. Its prefix row can still use index i , the number of characters processed.

Pause and predict: 2

There are four input values. What does `pref[4]` contain, and how many entries must the prefix vector have?

Check your prediction. It contains the total of all four values. The prefix vector needs five entries, including `pref[0]` for the empty prefix.

6.3 Building a prefix sum one value at a time

Define the stored information exactly

Let $a_1, a_2, \dots, a_n$ be the input. Define

$$P[0] = 0, \quad P[i] = a_1 + a_2 + \dots + a_i \quad (1 \leq i \leq n).$$

To get the first i values, take the first $i - 1$ values and add the new value a_i . Therefore,

$$P[i] = P[i - 1] + a_i.$$

This is a recurrence: it builds the next result from one already known. It is just a loop, not recursion.

Trace before writing the loop

i	New value	Calculation	$P[i]$
0	none	Empty prefix.	0
1	4	$0 + 4$	4
2	-2	$4 + (-2)$	2
3	7	$2 + 7$	9
4	0	$9 + 0$	9
5	3	$9 + 3$	12

Contract. Read $0 \leq n \leq 200000$ and n values with $|a_i| \leq 10^9$. Print $P[0], P[1], \dots, P[n]$ on one line. When $n = 0$, print just 0.

Complete program: build and print the prefixes

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    for (int i = 0; i <= n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << pref[i];
    }
    std::cout << '\n';
}
```

Sample input	Sample output
5 4 -2 7 0 3	0 4 2 9 9 12

Why this loop is correct

Before iteration i , `pref[i - 1]` contains the total of the previous $i - 1$ inputs. Adding the next input makes `pref[i]` the total of the first i inputs. The initial value `pref[0] = 0` makes the first iteration correct; the same reasoning then applies to every later iteration.

This statement is a **loop invariant**: something we can explain at the boundary of every iteration. It gives a reason for the code to work, instead of relying only on one sample.

Do we also need to store the original array?

Not here. We use each input value immediately to build its prefix entry. A separate original array is useful only when later work needs the original values directly. Even then, one value can be recovered as $a_i = P[i] - P[i - 1]$, provided the prefixes are still those of the unchanged array.

Pause and predict: 3

For inputs 6, -4, 2, what are all four prefix entries? Must prefix sums always increase?

Check your prediction. They are 0, 6, 2, 4. A negative input decreases the prefix total. Prefix sums are nondecreasing only when all input values are nonnegative.

6.4 Answering a range sum by cancellation

Derive the subtraction

A prefix ending at r contains everything we want *and* everything before l :

$$P[r] = \underbrace{a_1 + \cdots + a_{l-1}}_{\text{unwanted beginning}} + \underbrace{a_l + \cdots + a_r}_{\text{wanted range}},$$

$$P[l-1] = a_1 + \cdots + a_{l-1}.$$

Subtracting cancels the unwanted beginning:

$$\text{sum}(l, r) = P[r] - P[l-1].$$

The subtraction is justified by the *meaning* of the two entries. It is not a pattern to apply to every kind of prefix summary.

For our running example, $P = 0, 4, 2, 9, 9, 12$:

Query	Calculation	Answer
[2, 4]	$P[4] - P[1] = 9 - 4$	5
[1, 5]	$P[5] - P[0] = 12 - 0$	12
[3, 3]	$P[3] - P[2] = 9 - 2$	7

Contract. Read n q , the array, and then the queries. Now $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, $|a_i| \leq 10^9$, and $1 \leq l \leq r \leq n$. Print one sum per query. All values stay unchanged.

Complete program: build once, answer with two entries

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Sample input	Sample output
5 3	5
4 -2 7 0 3	12
2 4	7
1 5	
3 3	

Why the first position is not a special case

When $l = 1$, the formula subtracts $P[0] = 0$. We need no `if (l == 1)` branch. We also never read index -1 , because every allowed l is at least 1.

Why a single position works

For $l = r$, the answer is $P[l] - P[l - 1] = a_l$. A one-element range uses the same formula as a large range. Boundary cases should confirm the meaning, not require memorized exceptions.

Pause and predict: 4

A student uses `pref[r] - pref[l]`. Which input value has been removed by mistake?

Check your prediction. The value at position 1. That expression sums positions $l + 1$ through r ; for $l == r$, it incorrectly returns zero.

6.5 Safe totals, boundaries, and reusable helpers

Choose the type for the total, not only one input

With $n \leq 200000$ and $|a_i| \leq 10^9$, an absolute prefix total can be as large as $2 \cdot 10^{14}$. It does not fit the usual 32-bit `int`; it does fit the 64-bit `long long` used by our compiler. Signed overflow is undefined behavior in C++23, not a safe way to wrap a sum.^[6-3]

Use `long long` for both stored prefixes and returned sums. Changing only the final output variable does not repair an earlier overflowing addition.

```
// Wrong example: the addition may already overflow as int.
int a = 1000000000, b = 1000000000, c = 1000000000;
long long total = a + b + c;
```

```
// Correct fragment: widen before the arithmetic.
long long total = 1LL * a + b + c;
```

The complete programs read large summands as `long long`, so their cumulative additions are already wide.

A function should preserve the same contract

Session 3 taught us to pass a read-only collection by `const` reference. That matters here: copying an n -entry vector into every query helper would turn a constant-time lookup into linear work per call.

The next program has the same input/output contract as the range-sum program. It stores the original sequence to demonstrate a separate construction helper.

Pause and predict: 5

A prefix vector uses `long long`, but `rangeSum` returns `int`. Is that safe for every stated input?

Check your prediction. No. The query result can exceed the range of `int`. The return type must also preserve the wide result.

Complete program: build and query through helpers

```

#include <iostream>
#include <vector>

// a[0] is padding; real elements occupy indices 1..n.
std::vector<long long> buildPrefix(
    const std::vector<long long>& a) {
    const int n = static_cast<int>(a.size()) - 1;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i)
        pref[i] = pref[i - 1] + a[i];
    return pref;
}

// Contract: 1 <= l <= r < pref.size().
long long rangeSum(const std::vector<long long>& pref,
    int l, int r) {
    return pref[r] - pref[l - 1];
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> a(n + 1, 0);
    for (int i = 1; i <= n; ++i) std::cin >> a[i];
    const auto pref = buildPrefix(a);
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << rangeSum(pref, l, r) << '\n';
    }
}

```

Sample input**Sample output**

3 2	3000000000
1000000000 1000000000 1000000000	2000000000
1 3	
2 3	

A small boundary checklist

Test $[1, 1]$, $[n, n]$, $[1, n]$, and an interior range. Then test an all-negative array and a total above $2^{31} - 1$. The mathematics works with negative values; sortedness is not required.

The helper assumes its range is valid. It is appropriate for a contest task with guaranteed valid input. A general application accepting untrusted indices would need validation before using the helper.

6.6 Count the cost and recognize the limitations

Preprocessing, query time, and total time

Initializing $n + 1$ entries and building the prefix array take $O(n)$ time. A query uses two indexed reads and one subtraction: $O(1)$ time. Reading and answering q queries therefore gives total time $O(n + q)$, not just $O(1)$.

The prefix vector occupies $O(n)$ storage. The stream-built version does not keep a second input array. The helper version keeps both, but $2n$ entries are still $O(n)$ storage. This describes growth, not equal byte usage.

Approach	Preparation	One sum	Total
Scan each range	$O(n)$	$O(r - l + 1)$	$O(n + nq)$ worst case
Prefix sum	$O(n)$	$O(1)$	$O(n + q)$

An update makes later prefixes stale

If a_2 changes by $+5$, then every $P[i]$ with $i \geq 2$ also changes by $+5$. Editing only $a[2]$ does not update the prefix vector. A plain prefix array is not automatically a fast data structure for interleaved updates and queries.

Our later difference-array method handles **a batch of additions followed by reconstruction**. It does not make arbitrary online range-sum queries between those additions constant-time. Dynamic range-query structures are outside this session.

Why prefix minimums cannot be subtracted

Consider $a = 1, 5, 4$. The minimum of the first two values is 1, and the minimum of the first three values is also 1. The minimum of positions $[2, 3]$ is 4. Subtracting the prefix minimums gives 0, not 4.

A minimum summary may have forgotten the values needed after a boundary is excluded. A sum supports cancellation by subtraction; a minimum does not support that same cancellation. This is about lost information, not a C++ limitation.

Likewise, multiplying prefixes and dividing them is not a generally safe substitute: zeros prevent division, and products can overflow quickly. Derive the operation you need instead of changing the name of a prefix array.

Pause and predict: 6

You rebuild the whole prefix array inside the query loop. Is the total time still $O(n + q)$?

Check your prediction. No. The $O(n)$ construction runs q times, so total work is $O(nq)$ in the usual nonempty-query setting. Build once before the loop when the array is static.

6.7 Turn a condition into a range count

Counting is summing zeros and ones

Suppose a query asks how many **positive** values appear in positions $[l, r]$. Create a conceptual sequence b : let $b_i = 1$ when $a_i > 0$, otherwise let $b_i = 0$. The desired count is the sum of the corresponding b -values.

Position	1	2	3	4	5
Original value	4	-2	7	0	3
Positive?	1	0	1	0	1
Prefix count	1	1	2	2	3

There is also a leading prefix entry 0. In $[2, 4]$, the count is $2 - 1 = 1$, because only the value 7 is positive. Zero is **not** positive.

Contract. Use the same bounds and input format as ordinary range sums. Print the positive-value count for each query. An `int` count is sufficient because it is at most 200000 in this task.

Complete program: prefix counts of positive values

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<int> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + (value > 0);
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Sample input

```
5 3
4 -2 7 0 3
2 4
1 5
4 4
```

Sample output

```
1
3
0
```

The comparison `value > 0` produces a `bool`. In this integer addition, `false` contributes 0 and `true` contributes 1. You may write an `if` instead if it feels clearer.

The same idea works for text

Now a query asks for the number of vowels in a substring. A character contributes 1 exactly when it is one of a, e, i, o, or u. Do not use the character's numeric code as its contribution.

Contract. Read a nonempty lowercase ASCII word of length at most 200000, then $0 \leq q \leq 200000$, then q one-based inclusive ranges. Print the vowel count in each. The string is zero-based, so position i uses `text[i - 1]`.

Complete program: count vowels in requested substrings

```
#include <iostream>
#include <string>
#include <vector>

bool isVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i'
           || c == 'o' || c == 'u';
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string text;
    int q = 0;
    std::cin >> text >> q;
    const int n = static_cast<int>(text.size());
    std::vector<int> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i)
        pref[i] = pref[i - 1] + isVowel(text[i - 1]);
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Sample input	Sample output
education 3	2
1 4	4
3 8	0
2 2	

What information can one prefix count answer?

For a fixed predicate, let the range count be c . Test $c > 0$ for “any”, $c = 0$ for “none”, or $c = r - l + 1$ for “all”. A changed predicate, such as exceeding a query-dependent threshold x , generally needs a different summary.

Pause and predict: 7

An interval has length 6 and its prefix-count difference is 4. What does that tell us about the number of elements that fail the fixed condition?

Check your prediction. Exactly $6 - 4 = 2$ elements fail it. Each element contributes either 0 or 1, so the two counts partition the interval.

6.8 Fixed-length windows without repeated scans

Every window is one range query

A **window of length k** is a consecutive block containing exactly k values. If its first position is l , its last position is $r = l + k - 1$. The valid starting positions are 1 through $n - k + 1$.

To find the maximum-sum window, calculate each window's sum from the prefixes and keep the best. There are $n - k + 1$ windows, and each sum takes constant time.

Contract. Read n k , then n signed values. Bounds are $1 \leq k \leq n \leq 200000$, $|a_i| \leq 10^9$. Print the greatest window sum and its one-based starting position. On a tie, choose the earliest start.

Why zero is not a safe initial best

For $-5, -2, -7, -1, -3$ with $k = 2$, every possible window sum is negative. Initializing `best = 0` would keep a value that belongs to no window. Instead, initialize from the first valid window.

Starting position	1	2	3	4
Length-2 sum	-7	-9	-8	-4

Complete program: best length- k window

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    long long best = pref[k];
    int start = 1;
    for (int l = 2; l + k - 1 <= n; ++l) {
        const int r = l + k - 1;
        const long long sum = pref[r] - pref[l - 1];
        if (sum > best) {
            best = sum;
            start = l;
        }
    }
    std::cout << best << ' ' << start << '\n';
}
```

Sample input	Sample output
5 2 -5 -2 -7 -1 -3	-4 4

The first window has sum `pref[k]`. The loop begins at start 2 because the first one is already the initial answer. Updating only when `sum > best` preserves the earlier start when two sums are equal.

Reason and cost

After processing a start position, `best` is the largest sum among the windows considered so far, and `start` is the earliest position achieving it. Since every valid start is visited, the final answer satisfies both the maximum and tie-breaking rules. Total time is $O(n)$; prefix storage is $O(n)$.

A rolling-window solution can remove the departing value and add the entering value instead. That is another valid $O(n)$ approach. Here we deliberately reuse the range-sum method you have just learned.

Pause and predict: 8

How many length- k windows exist when $k = n$? Which boundary must the loop allow when $k = 1$?

Check your prediction. There is one window when $k = n$. When $k = 1$, every single element is a window, including the one starting at n .

6.9 Prefix and suffix information around a position

Do not accidentally include the pivot

A **balance position** has equal sums strictly to its left and strictly to its right. The value at the position itself belongs to neither side.

Using our prefix definition, the left sum at position i is $P[i - 1]$, and the right sum is $P[n] - P[i]$. The difference between i and $i - 1$ is essential: $P[i]$ includes the pivot and $P[i - 1]$ does not.

For 2, 1, 5, 1, 2, position 3 is balanced because both sides sum to 3. The pivot value 5 is irrelevant to that comparison.

Contract. Read $1 \leq n \leq 200000$ and the signed array, with $|a_i| \leq 10^9$. Print the smallest balance position, or -1 if none exists.

Complete program: find the first balance position

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    int answer = -1;
    for (int i = 1; i <= n; ++i) {
        const long long left = pref[i - 1];
        const long long right = pref[n] - pref[i];
        if (left == right) {
            answer = i;
            break;
        }
    }
    std::cout << answer << '\n';
}
```

Sample input

Sample output

5

3

2 1 5 1 2

Empty sides are valid totals

At position 1, the left side is empty and has sum 0. At position n , the right side is empty and has sum 0. A one-element sequence is therefore balanced at position 1, whatever its single value is.

A suffix array is the same idea from the other end

A **suffix** ends at the final element. If we define $S[i] = a_i + \dots + a_n$, we can build it backwards with $S[n+1] = 0$ and $S[i] = a_i + S[i+1]$. A suffix vector needs $n+2$ entries to use that padding directly.

```
// Inside main, after reading a[1..n].
std::vector<long long> suffix(n + 2, 0);
for (int i = n; i >= 1; --i)
    suffix[i] = a[i] + suffix[i + 1];
// The sum strictly after i is suffix[i + 1].
```

For sums, the total minus a prefix already gives the needed suffix sum, so the complete program does not allocate both arrays. Learn to recognize both directions without storing redundant information automatically.

Pause and predict: 9

For the array 0, 2, -2, 0, is position 1 balanced even though there are three values on its right?

Check your prediction. Yes. The empty left side sums to 0, and the right side sums to $2 - 2 + 0 = 0$. Balance compares sums, not lengths.

6.10 Frequency arrays: use a value as an index

The index now answers a different question

An ordinary sequence array stores $a[\text{position}] = \text{value}$. A **frequency array** stores $\text{freq}[\text{value}] = \text{number of occurrences}$. Its index represents a possible value, not an input position.

For the values 3, 0, 3, 5, 1, 3:

Possible value	0	1	2	3	4	5
Frequency	1	1	0	3	0	1

$\text{freq}[3] = 3$ means the value 3 appeared three times. It does not describe the third input item. A zero frequency represents an allowed value that did not appear.

Why the domain matters

The **domain** is the set of possible input values. If values are integers in $[0, U]$, an array with $U + 1$ entries can represent all of them. Start every entry at zero, then increment the entry for each value. The filled-vector constructor supplies the required initial zeros.^[6-2]

Contract. Read n upper q , then n values, then q queried values. Here $0 \leq n, q, U \leq 200000$; every input and query value lies in $[0, U]$. Print each queried frequency. Queries do not change the counts.

Complete program: exact frequency queries

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0, q = 0;
    std::cin >> n >> upper >> q;
    std::vector<int> freq(upper + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value];
    }
    while (q--) {
        int value = 0;
        std::cin >> value;
        std::cout << freq[value] << '\n';
    }
}
```

Sample input	Sample output
6 5 4	3
3 0 3 5 1 3	0
3	1
2	1
0	
5	

Trace one increment carefully

Before processing the third occurrence of 3, `freq[3]` is 2. Executing `++freq[3]` changes it to 3. Other frequency entries stay unchanged. After all input, the sum of all frequency entries must be n .

Cost and a useful comparison with maps

Initializing the domain costs $O(U + 1)$. Reading the values costs $O(n)$, and each exact frequency query costs $O(1)$. Total time is $O(U + n + q + 1)$, with $O(U + 1)$ storage. We retain $+1$ here because a domain with $U = 0$ still has one possible value.

For a compact domain, direct indexing is simple. For arbitrary names or very large sparse integer values, the maps from Session 5 are often a better representation. Do not allocate a billion-entry vector merely because the largest key is a billion.

Pause and predict: 10

There are only five inputs, but they may have values from 0 through 100. Is `vector<int> freq(n + 1, 0)` large enough?

Check your prediction. No. The frequency index is a value, so it may be 100. The vector needs 101 entries even though only five observations are read.

6.11 Negative values, offsets, and domain safety

A negative number is not a valid direct index

If values lie in $[-1000, 1000]$, we can shift the entire domain by a fixed offset. Map a value x to index $x - \text{LOW}$, where $\text{LOW} = -1000$.

Value	-1000	-2	0	7	1000
Stored index	0	998	1000	1007	2000

The required size is $\text{HIGH} - \text{LOW} + 1 = 2001$. The inverse conversion is $x = \text{index} + \text{LOW}$.

Contract. Read n q , then the values, then the queried values. All values and queries lie in $[-1000, 1000]$; $0 \leq n, q \leq 200000$. Print each frequency.

Complete program: bounded signed values

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    constexpr int LOW = -1000;
    constexpr int HIGH = 1000;
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<int> freq(HIGH - LOW + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value - LOW];
    }
    while (q--) {
        int value = 0;
        std::cin >> value;
        std::cout << freq[value - LOW] << '\n';
    }
}
```

Sample input	Sample output
5 3	2
-2 0 -2 1000 -1000	1
-2	0
-1000	
7	

Use the same mapping everywhere

Every insertion and every query must use `value - LOW`. Applying an offset while building counts but forgetting it during lookup reads a different entry.

The offset is a translation of a **known finite domain**. It does not reduce the number of different possible indices. Shifting $[-10^9, 10^9]$ still leaves more than two billion possible values; it does not solve the memory problem. Later, coordinate compression will keep only the distinct values actually needed.

Do not hide an invalid input by changing its value

Taking an absolute value, using `x % size`, or clamping a value into the valid domain can merge different keys. These are not general fixes for an out-of-bounds frequency access. Either the contract guarantees a valid value, or your application must validate or represent it differently.

Pause and predict: 11

Why would indexing `freq[abs(value)]` be wrong for counts of signed values?

Check your prediction. It merges $-x$ and $+x$ into the same entry. They are different values unless the task explicitly asks to count absolute values.

6.12 Distinct values, modes, and tie-breaking

A frequency table can answer several questions

A value is present exactly when its count is positive. Counting positive entries gives the number of **distinct values**. The value with greatest frequency is a **mode**. If several values tie, the problem must specify which one to report.

Contract. Read n upper, then n values in $[0, U]$, with $1 \leq n \leq 200000$ and $0 \leq U \leq 200000$. Print the distinct count on the first line. Print the smallest most-frequent value and its count on the second line.

Complete program: a small frequency report

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0;
    std::cin >> n >> upper;
    std::vector<int> freq(upper + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value];
    }
    int best = 0, distinct = 0;
    for (int value = 0; value <= upper; ++value) {
        if (freq[value] > 0) ++distinct;
        if (freq[value] > freq[best]) best = value;
    }
    std::cout << distinct << '\n';
    std::cout << best << ' ' << freq[best] << '\n';
}
```

Sample input

7 5
4 2 4 2 5 0 5

Sample output

4
2 2

Why the smallest tied value survives

The scan visits values in increasing order. It changes **best** only when a *strictly larger* frequency is found. Therefore, if a later value has the same frequency, the earlier and smaller value stays selected.

Starting **best** = 0 is safe even when the value 0 is absent: its count is zero, and some input value must have positive frequency because $n \geq 1$. If empty input were allowed, the problem would need a separate rule for reporting a mode.

Frequency order is not input order

Scanning `freq[0]`, `freq[1]`, and so on reports values in numeric order. It does not preserve which occurrence came first. To find the *first input value with total frequency one*, you must keep or revisit the original order after building counts.

This also explains when a frequency table is insufficient: it remembers multiplicity but forgets arrangement. Repeated values can be printed in sorted order by printing each possible value `freq[value]` times, but that reconstructs a sorted sequence, not the original one.

Pause and predict: 12

If a scan from small values to large values updates the mode on `>=` instead of `>`, which tied value will remain?

Check your prediction. The largest tied value, because every later equal frequency replaces the earlier answer. That violates a smallest-on-tie requirement.

6.13 Small alphabets and anagram checks

Give each allowed character a small ID

For lowercase ASCII English letters, $c - 'a'$ maps 'a' to 0, 'b' to 1, and 'z' to 25. We explicitly use this alphabet; the method is not a general Unicode text-processing technique.

Two words are **anagrams** when their letters can be rearranged to match. The arrangement may differ, but every letter must have the same multiplicity. Merely checking that both words contain the same distinct letters is not enough.

Compare counts or keep one balance table

One approach builds two 26-entry arrays and compares them. Another adds the first word's counts and subtracts the second word's counts into one **balance** array. Every entry must end at zero.

For **aabcc** and **cbaca**, the first word contributes counts $a : 2, b : 1, c : 2$, and the second removes exactly those counts. For **aabb** and **abbb**, the final balances include $a : +1, b : -1$, so the words are not anagrams.

Contract. Read two nonempty lowercase ASCII words, each of length at most 200000. Print YES if they are anagrams, otherwise NO.

Complete program: compare letter multiplicities

```

#include <iostream>
#include <string>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string a, b;
    std::cin >> a >> b;
    std::vector<int> balance(26, 0);
    for (char c : a) ++balance[c - 'a'];
    for (char c : b) --balance[c - 'a'];
    bool same = true;
    for (int count : balance) {
        if (count != 0) same = false;
    }
    std::cout << (same ? "YES" : "NO") << '\n';
}

```

Sample input

aabcc cbaca

Sample output

YES

The counters are signed because subtraction can produce negative values. A separate length check is a valid early rejection, but it is not necessary for correctness here: if every letter balance is zero, the lengths must also match.

Time and space

For word lengths n and m , processing the text takes $O(n + m)$ time and checking the 26 counts takes $O(26)$. The count table uses $O(26) = O(1)$ extra space for this fixed alphabet. The program also stores the two input strings, so its total explicit storage is $O(n + m)$.

Pause and predict: 13

Why does a set of letters incorrectly accept **aabb** and **abbb** as equivalent?

Check your prediction. Both sets are just $\{a, b\}$. A set discards repeated copies, but anagram checking must preserve their counts.

6.14 Count equal pairs without a nested loop

Specify what counts as a pair

We want the number of pairs of **different positions** (i, j) with $i < j$ and $a_i = a_j$. Equal-valued occurrences are separate input items. The order restriction avoids counting both (i, j) and (j, i) .

If a value occurs f times, it contributes $f(f - 1)/2$ such pairs. Alternatively, count pairs as the values arrive. When the next value is x , each earlier occurrence of x makes exactly one new pair whose right endpoint is the current position.

Count first, insert second

For 3, 0, 3, 5, 3, 0:

Position	Value	Earlier equal values	New total
1	3	0	0
2	0	0	0
3	3	1	1
4	5	0	1
5	3	2	3
6	0	1	4

Contract. Read n upper, then values in $[0, U]$, with $0 \leq n, U \leq 200000$. Print the number of equal pairs.

Complete program: count pairs during input

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0;
    std::cin >> n >> upper;
    std::vector<long long> freq(upper + 1, 0);
    long long pairs = 0;
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        pairs += freq[value];
        ++freq[value];
    }
    std::cout << pairs << '\n';
}
```

Sample input

6 5
3 0 3 5 3 0

Sample output

4

Every valid pair is counted once, when its second occurrence is read. Incrementing the frequency *before* adding it to the answer would also count the current occurrence paired with itself.

The answer needs a wider type than the count

With 200000 identical values, there are 19999900000 pairs, which exceeds a signed 32-bit integer. The answer uses `long long`. The frequency vector also uses `long long` to make any later formula such as $f * (f - 1) / 2$ naturally wide.

Total time is $O(n + U + 1)$, including initialization. Storage is $O(U + 1)$. Compared with checking every pair explicitly, we have summarized the only fact the next step needs: the number of matching earlier occurrences.

Pause and predict: 14

When four copies of the same value arrive, how many new pairs does each copy create?

Check your prediction. They create 0, 1, 2, 3 new pairs, for a total of 6. The fourth copy pairs with the previous three, not with itself.

6.15 Prefix sums over the value domain

A different kind of range

So far, a range sum usually described **positions**. Now suppose a query asks: “How many input values lie between L and R , inclusive?” These endpoints are **values**. For example, among 3, 0, 3, 5, 1, 3, four values lie in $[1, 3]$: one copy of 1 and three copies of 3.

The answer is $\text{freq}[L] + \dots + \text{freq}[R]$. That is another static range-sum problem, this time over the frequency array.

Make room for the value zero

The frequency array is indexed from 0 through U . Define a new prefix array C with

$$C[0] = 0, \quad C[v + 1] = C[v] + \text{freq}[v].$$

Thus, $C[t]$ is the number of input values **strictly smaller than t** . It contains $U + 2$ entries, and a value-range query is

$$\#\{i : L \leq a_i \leq R\} = C[R + 1] - C[L].$$

This is exactly the zero-based inclusive formula from the indexing table. We have not changed the prefix idea; we have changed which axis is being prefixed.

Contract. Read n upper q , the values, then value ranges low high . Bounds are $0 \leq n, U, q \leq 200000$, $0 \leq a_i \leq U$, and $0 \leq L \leq R \leq U$. Print one count per query.

Complete program: count observations in a value interval

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0, q = 0;
    std::cin >> n >> upper >> q;
    std::vector<int> freq(upper + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value];
    }
    std::vector<int> pref(upper + 2, 0);
    for (int value = 0; value <= upper; ++value)
        pref[value + 1] = pref[value] + freq[value];
    while (q--) {
        int low = 0, high = 0;
        std::cin >> low >> high;
        std::cout << pref[high + 1] - pref[low] << '\n';
    }
}
```

Sample input	Sample output
6 5 3	4
3 0 3 5 1 3	1
1 3	1
0 0	
4 5	

Read the table with its meaning attached

For our example, $\text{freq} = 1, 1, 0, 3, 0, 1$ and $C = 0, 1, 2, 2, 5, 5, 6$. Query $[1, 3]$ gives $C[4] - C[1] = 5 - 1 = 4$. Query $[0, 0]$ gives $C[1] - C[0] = 1$, with no negative index.

The build costs $O(n + U + 1)$; each query costs $O(1)$. Two domain-sized arrays still use $O(U + 1)$ storage. Bounds outside $[0, U]$ are not accepted by this program's contract; a broader application must handle them deliberately.

Pause and predict: 15

Does $C[4] - C[1]$ count original positions 2 through 4?

Check your prediction. No. It counts all observations whose *values* are 1, 2, or 3, wherever they occurred. The prefix axis is possible values, not input positions.

6.16 Letter counts inside substrings

Why one whole-word frequency table is insufficient

Consider `aabb` and `bbaa`. Both words have two `a`s and two `b`s. Yet their first two positions have different letter counts. A whole-word frequency table cannot answer every substring query because it has forgotten positions.

For a fixed small alphabet, preserve both pieces of information. Let $P[i][c]$ be the number of occurrences of letter ID c in the first i characters. There are $n + 1$ prefix rows and 26 columns.

Copy yesterday's counts, then add today's letter

To build row i , copy row $i - 1$, then increment the column for `text[i - 1]`. Every other count stays the same. For `abaca`, the `a` column is 0, 1, 1, 2, 2, 3; the `b` column is 0, 0, 1, 1, 1, 1.

For a query asking about letter c in $[l, r]$, subtract the corresponding columns:

$$\text{count}(l, r, c) = P[r][c] - P[l - 1][c].$$

Contract. Read a nonempty lowercase ASCII word of length at most 100000, then $0 \leq q \leq 200000$. Each query is `l r c`, with a valid one-based inclusive range and a lowercase letter. Print that letter's count in the range.

Complete program: one prefix column per letter

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string text;
    int q = 0;
    std::cin >> text >> q;
    const int n = static_cast<int>(text.size());
    std::vector<std::vector<int>> pref(
        n + 1, std::vector<int>(26, 0));
    for (int i = 1; i <= n; ++i) {
        pref[i] = pref[i - 1];
        ++pref[i][text[i - 1] - 'a'];
    }
    while (q--) {
        int l = 0, r = 0;
        char c = 'a';
        std::cin >> l >> r >> c;
        const int id = c - 'a';
        std::cout << pref[r][id] - pref[l - 1][id] << '\n';
    }
}
```

Sample input	Sample output
abacaba 3	2
2 6 a	2
1 7 b	0
4 4 a	

Count the row-copy work and the memory

Copying a row visits 26 counts, so construction takes $O(26n)$ time and storage is $O(26n)$. A query about one letter costs $O(1)$. For an alphabet of size σ , write $O(\sigma n + q)$, rather than silently assuming copying a row of arbitrary size is constant-time.

With 100000 characters and four-byte `int` entries, the raw counts alone occupy about 10.4 million bytes. A vector of vectors also has row objects and allocation overhead. This is comfortable for many tasks but should not be extended blindly to hundreds of thousands of distinct keys.

For a large alphabet, storing the sorted occurrence positions of each value is another option. A pair of binary searches can count positions in a requested interval without allocating one full prefix column per possible value. That builds on Session 5's map-of-vectors pattern; it is not needed for the small-alphabet program here.

Pause and predict: 16

What goes wrong if a new row starts as all zeros and we increment only the current letter, without copying the previous row?

Check your prediction. The row remembers only that one character, not the entire prefix. Subtracting such rows does not produce a substring frequency.

6.17 Differences: remember changes instead of values

An inverse view of a sequence

A prefix sum accumulates values. A **difference array** records changes between consecutive values. For a one-based sequence, use the conceptual boundary value $a_0 = 0$, then define

$$D[1] = a_1, \quad D[i] = a_i - a_{i-1} \quad (2 \leq i \leq n).$$

For $a = 4, -2, 7, 0, 3$, the differences are $4, -6, 9, -7, 3$. An entry is a *change*, so it can be negative even when many original values are positive.

Running sums undo the differences

Starting from zero, add the changes in order:

Position	1	2	3	4	5
Difference	4	-6	9	-7	3
Running value	4	-2	7	0	3

Algebraically, all the middle terms cancel:

$$D[1] + \cdots + D[i] = a_1 + (a_2 - a_1) + \cdots + (a_i - a_{i-1}) = a_i.$$

The prefix sum of a difference array therefore reconstructs the original values. This relationship is what makes boundary updates useful.^[6-1]

Contract. Read $0 \leq n \leq 200000$ and signed values with $|a_i| \leq 10^9$. Print the differences on one line, then the reconstructed sequence on another. For $n = 0$, print two empty lines.

Complete program: take differences, then reconstruct

```

#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> diff(n + 1, 0);
    long long previous = 0;
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        diff[i] = value - previous;
        previous = value;
    }
    for (int i = 1; i <= n; ++i) {
        if (i > 1) std::cout << ' ';
        std::cout << diff[i];
    }
    std::cout << '\n';
    long long current = 0;
    for (int i = 1; i <= n; ++i) {
        current += diff[i];
        if (i > 1) std::cout << ' ';
        std::cout << current;
    }
    std::cout << '\n';
}

```

Sample input

5
4 -2 7 0 3

Sample output

4 -6 9 -7 3
4 -2 7 0 3

Do not confuse three related quantities

For the same input, $P[i]$ is the total through position i , a_i is the actual value at that position, and $D[i]$ is its change from the previous value. These are three different meanings, even though loops involving addition and subtraction connect them.

The syllabus label **partial sums** is sometimes used for the range-update topic because reconstruction takes running totals. In this book, *difference array* names the stored boundary changes; *prefix sum* names the cumulative operation. The standard-library function `std::partial_sum` is discussed separately later.

Pause and predict: 17

For $a = 5, 5, 5$, what is the difference array? Why are the last two entries not 5?

Check your prediction. It is 5, 0, 0. After the first position, the value does not change, so the next differences are zero.

6.18 A range addition changes only two boundaries

Describe the intended effect before the code

Suppose every position starts at zero. Add x to every position in $[l, r]$. The final extra value should be zero before l , equal to x from l through r , and zero again after r .

Rather than visiting every covered position, record two changes:

$$D[l] += x, \quad D[r + 1] -= x.$$

At l , turn the addition on. At $r + 1$, turn it off. Because r is included, the cancellation belongs *after* it, not at r .

A one-update trace

With $n = 6$, adding 4 to $[2, 5]$ gives the following boundary changes. Position 7 is a sentinel beyond the real output.

Index	1	2	3	4	5	6	7
Change	0	4	0	0	0	-4	0
Running value	0	4	4	4	4	0	0

Only positions 2 through 5 retain the addition. If the update ended at n , its cancellation would be stored at $n + 1$.

Contract. Read n `updates`, then `updates` triples `l r delta`. Start with n zeros. Bounds are $1 \leq n \leq 200000$, $0 \leq u \leq 200000$, $|\Delta| \leq 10^9$, and valid one-based inclusive ranges. Print all final values on one line.

Complete program: record additions, then reconstruct

```

#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, updates = 0;
    std::cin >> n >> updates;
    std::vector<long long> diff(n + 2, 0);
    for (int j = 0; j < updates; ++j) {
        int l = 0, r = 0;
        long long delta = 0;
        std::cin >> l >> r >> delta;
        diff[l] += delta;
        diff[r + 1] -= delta;
    }
    long long current = 0;
    for (int i = 1; i <= n; ++i) {
        current += diff[i];
        if (i > 1) std::cout << ' ';
        std::cout << current;
    }
    std::cout << '\n';
}

```

Sample input

```

5 3
2 4 3
1 2 -1
4 5 2

```

Sample output

```

-1 2 3 5 2

```

Why the vector has $n + 2$ entries

Real positions are 1 through n . Index 0 is unused padding. Index $n + 1$ can receive a cancellation. A vector with $n + 2$ entries has valid indices 0 through $n + 1$, so every permitted $r + 1$ is safe.

The reconstruction loop stops at n . The sentinel is not another answer to print. We update it so that the boundary formula remains uniform even when the range reaches the end.

Pause and predict: 18

What happens if an inclusive update $[2, 4]$ cancels at `diff[4]` instead of `diff[5]`?

Check your prediction. Position 4 loses the addition. The effect remains active only on positions 2 and 3, so the right endpoint is incorrectly excluded.

6.19 Overlapping updates and cancellation

Add boundary contributions; do not replace them

Different updates may start or finish at the same index. Use $+=$ and $-=$; a plain assignment would overwrite earlier contributions.

Consider $n = 5$ and three updates:

$$[2, 4] += 3, \quad [1, 2] += -1, \quad [4, 5] += 2.$$

The first update contributes $+3$ at index 2 and -3 at index 5. The second contributes -1 at index 1 and $+1$ at index 3. The third contributes $+2$ at index 4 and -2 at index 6.

Index	1	2	3	4	5	6
After first update	0	3	0	0	-3	0
After second update	-1	3	1	0	-3	0
After third update	-1	3	1	2	-3	-2
Running extra value	-1	2	3	5	2	0

Only the first five running values are printed: $-1, 2, 3, 5, 2$.

Explain each position another way

Position 1 receives only -1 . Position 2 receives $3 - 1 = 2$. Position 3 receives only 3 . Position 4 receives $3 + 2 = 5$. Position 5 receives only 2 . This direct check agrees with the reconstructed values.

Why overlapping additions work

For any one update, its contribution after reconstruction is x exactly inside its interval and zero outside. Addition lets us combine all these independent contributions. At any position, the running total is the sum of the updates that cover that position.

This also explains why a negative update works without a special rule. Adding $-x$ means its start marker is negative and its cancellation is positive. An update followed by the opposite update on the same interval has net effect zero.

Addition is not assignment

These two boundary operations implement “add x to the range.” They do not implement “set every element in the range to x .” For assignments, the last applicable operation can replace earlier values; that behavior does not follow from summing independent boundary contributions.

Pause and predict: 19

An update adds 7 to $[1, 3]$, then another adds -7 to $[1, 3]$. What remains in the difference array for those updates?

Check your prediction. The markers at both boundaries cancel to zero, so the final extra value is zero everywhere.

6.20 Apply additions to a nonzero starting array

Two valid models; choose one

When the initial values are not all zero, there are two natural representations:

Model A: differences of the complete array. Initialize $D[i] = a_i - a_{i-1}$, add the update markers, then reconstruct. The running sum directly gives the final value.

Model B: differences of the added amounts only. Keep the initial array unchanged. Start a separate difference vector at zero, record only updates, and reconstruct the total extra amount. The final value is the original value plus that extra amount.

Both work. The next program uses Model B because it keeps the two meanings visible.

Contract. Read n updates, the initial values, then l r Δ updates. Use $1 \leq n \leq 200000$, $0 \leq u \leq 200000$, $|a_i|, |\Delta| \leq 10^9$. Print the final array after all additions.

Complete program: original values plus accumulated additions

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, updates = 0;
    std::cin >> n >> updates;
    std::vector<long long> a(n + 1, 0);
    for (int i = 1; i <= n; ++i) std::cin >> a[i];
    std::vector<long long> diff(n + 2, 0);
    for (int j = 0; j < updates; ++j) {
        int l = 0, r = 0;
        long long delta = 0;
        std::cin >> l >> r >> delta;
        diff[l] += delta;
        diff[r + 1] -= delta;
    }
    long long added = 0;
    for (int i = 1; i <= n; ++i) {
        added += diff[i];
        if (i > 1) std::cout << ' ';
        std::cout << a[i] + added;
    }
    std::cout << '\n';
}
```

Sample input

```
5 2
4 -2 7 0 3
2 4 3
1 2 -1
```

Sample output

```
3 0 10 3 3
```

Trace the sample by its two layers

The initial array is $4, -2, 7, 0, 3$. The updates add 3 to $[2, 4]$ and -1 to $[1, 2]$. Their combined extra amounts are $-1, 2, 3, 3, 0$. Adding the layers gives $3, 0, 10, 3, 3$.

The common double-counting mistake

If you initialize `diff` from the original array *and* print `a[i] + running`, you count the initial value twice. In Model A, `running` already contains the initial value. In Model B, `added` contains only changes. The variable name is a useful reminder, but the definition is what guarantees correctness.

With up to 200000 updates of magnitude 10^9 , one final value can have magnitude about $2 \cdot 10^{14} + 10^9$. Use `long long` for updates, differences, running additions, and the initial values.

Pause and predict: 20

A program initializes the difference vector to zero and ignores the initial array while printing. What does it print instead of final values?

Check your prediction. Only the accumulated extra amounts. It must add those amounts to the initial values, or instead initialize differences from the initial array.

6.21 Processing phases: updates first, sums afterwards

Separate the two jobs

Suppose all range additions arrive first, then many range-sum queries ask about the **final** array. Use a difference array for the update phase and a prefix array for the query phase.

Record boundaries → **Reconstruct values** → **Prefix totals**

The final two steps can share one left-to-right loop: calculate the extra amount at position i , add it to the original value, then extend the prefix total.

Contract. Read n updates q , the initial values, then the updates, then the queries. Use $1 \leq n \leq 200000$, $0 \leq u, q \leq 200000$, $|a_i| \leq 10^9$, and $|\Delta| \leq 10^6$. Every query refers to the array *after every update*. Print one range sum per query.

Complete program: difference updates followed by static queries

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, updates = 0, q = 0;
    std::cin >> n >> updates >> q;
    std::vector<long long> a(n + 1, 0);
    for (int i = 1; i <= n; ++i) std::cin >> a[i];
    std::vector<long long> diff(n + 2, 0);
    for (int j = 0; j < updates; ++j) {
        int l = 0, r = 0;
        long long delta = 0;
        std::cin >> l >> r >> delta;
        diff[l] += delta;
        diff[r + 1] -= delta;
    }
    std::vector<long long> pref(n + 1, 0);
    long long added = 0;
    for (int i = 1; i <= n; ++i) {
        added += diff[i];
        pref[i] = pref[i - 1] + a[i] + added;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Sample input	Sample output
5 2 3	19
4 -2 7 0 3	13
2 4 3	3
1 2 -1	
1 5	
2 4	
1 1	

Reason and cost

At each reconstructed position, `added` equals the sum of update amounts covering that position. Therefore `a[i] + added` is its final value. Adding that to the preceding prefix constructs the correct final-array prefix. Ordinary subtraction then answers each query.

Recording one update takes $O(1)$. Recording all u updates takes $O(u)$, reconstruction and prefix building take $O(n)$, and all queries take $O(q)$. Total time is $O(n + u + q)$, with $O(n)$ storage.

The type bound changes when we sum updated values

A final value can have magnitude up to $10^9 + 200000 \cdot 10^6$. Summing 200000 such values gives a conservative bound $4.02 \cdot 10^{16}$, which fits the tested 64-bit `long long`. The smaller update bound here is intentional. A type that fits one updated value may still be too small for a sum of many updated values.

This is not an interleaved-update data structure

A query that belongs between two updates must see only the earlier updates. This program would instead include all updates. Do not reorder an input's semantics merely to make a difference-array solution fit. The batch approach applies only when final-state queries are requested.

Pause and predict: 21

Recording u range additions costs $O(u)$. Is $O(u)$ the whole cost when the program must print all n final values?

Check your prediction. No. Initialization, reconstruction, and output also take $O(n)$, so the total is $O(n + u)$.

6.22 Coverage counts on an integer line

Every interval adds one active item

Imagine n numbered time slots and several activities. An activity covering slots $[l, r]$ contributes 1 to each covered slot. A difference array can record these contributions with $+1$ at l and -1 at $r + 1$.

After reconstruction, **active** is the number of covering intervals at the current slot. We can use it to find the busiest slot, count uncovered slots, or count slots covered at least k times.

Contract. Read n intervals threshold, then the inclusive intervals. Bounds are $1 \leq n \leq 200000$, $0 \leq m \leq 200000$, $1 \leq k \leq 200001$. Print the number of slots with coverage at least k .

Complete program: count sufficiently covered slots

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, intervals = 0, threshold = 0;
    std::cin >> n >> intervals >> threshold;
    std::vector<int> diff(n + 2, 0);
    for (int j = 0; j < intervals; ++j) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        ++diff[l];
        --diff[r + 1];
    }
    int active = 0, answer = 0;
    for (int i = 1; i <= n; ++i) {
        active += diff[i];
        if (active >= threshold) ++answer;
    }
    std::cout << answer << '\n';
}
```

Sample input

```
6 3 2
1 3
2 5
5 6
```

Sample output

```
3
```

For the sample intervals $[1, 3]$, $[2, 5]$, and $[5, 6]$, coverage is 1, 2, 2, 1, 2, 1. Slots 2, 3, and 5 meet the threshold 2, so the answer is 3.

Shared endpoints and discrete slots

Here the intervals $[1, 3]$ and $[3, 5]$ both include slot 3. The activity ending at slot 3 must stop at boundary 4. Counting integer slots in $[l, r]$ gives $r - l + 1$.

For continuous time intervals written $[l, r)$, an end marker instead belongs at r , and elapsed length is $r - l$. Do not silently use a discrete inclusive formula for a continuous-time duration problem.

What changes for threshold zero?

This program's stated threshold is positive. Mathematically, every numbered slot is covered at least zero times, so a problem allowing $k = 0$ would count all n slots, even with no activities. Make such domain decisions explicit in the statement.

Pause and predict: 22

There are no intervals and the threshold is 1. Should the answer be n , 1, or 0?

Check your prediction. It is 0. Every slot has coverage zero, and none meets a positive threshold.

6.23 Combine differences, a condition, and a prefix

A complete pipeline: busy slots inside a requested interval

Now many queries ask: “How many slots in $[l, r]$ have coverage at least k ?” The coverage intervals and the fixed threshold are known before any query.

A direct solution might scan every activity for every slot in every query. Instead, choose the stored information in three stages:

Boundary changes tell us where each activity begins and stops contributing. Their running sum gives **coverage**. The condition $\text{coverage} \geq k$ becomes a **0/1 indicator**. Prefix sums of that indicator answer the queries.

For coverage 1, 2, 2, 1, 2, 1 and $k = 2$:

Slot	1	2	3	4	5	6
Coverage	1	2	2	1	2	1
Meets threshold?	0	1	1	0	1	0
Prefix of good slots	0	1	2	2	3	3

There is also a leading zero prefix entry. Query $[3, 5]$ gives $3 - 1 = 2$: slots 3 and 5 qualify.

Contract. Read n intervals threshold q , then the intervals, then the queries. Use the coverage bounds from the previous section and $0 \leq q \leq 200000$. All endpoints are valid one-based inclusive positions. The threshold stays the same for all queries.

Complete program: count good slots in each query range

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, intervals = 0, threshold = 0, q = 0;
    std::cin >> n >> intervals >> threshold >> q;
    std::vector<int> diff(n + 2, 0);
    for (int j = 0; j < intervals; ++j) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        ++diff[l];
        --diff[r + 1];
    }
    std::vector<int> good(n + 1, 0);
    int active = 0;
    for (int i = 1; i <= n; ++i) {
        active += diff[i];
        good[i] = good[i - 1] + (active >= threshold);
    }
    while (q-- > 0) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << good[r] - good[l - 1] << '\n';
    }
}
```

Sample input	Sample output
6 3 2 3	3
1 3	2
2 5	0
5 6	
1 6	
3 5	
6 6	

Why the final subtraction answers the right question

`good[i]` does not store the coverage at slot `i`. It stores the number of qualifying slots among the first `i` slots. Thus `good[r] - good[l - 1]` removes exactly the qualifying slots before the requested range.

It would be wrong to prefix the raw coverage values and subtract them here: that would count total activity-slot incidences, not how many slots meet a threshold.

Complexity and a changed-query warning

The whole program takes $O(n + m + q)$ time and $O(n)$ storage. We never store all queries. If every query supplies a different threshold, however, a single `good` prefix built for one k does not answer them all. The representation must match the actual question.

Pause and predict: 23

Coverage in three slots is 2, 2, 0, and $k = 2$. Why is the qualifying-slot count 2 rather than 4?

Check your prediction. Each qualifying slot contributes one, regardless of how many activities cover it. The raw coverage sum is 4, but there are only two qualifying slots.

6.24 Two-dimensional prefixes: totals from a corner

From an initial segment to an initial rectangle

A matrix has rows and columns. Define $P[r][c]$ as the sum of every cell in rows 1 through r and columns 1 through c . It is a rectangle whose upper-left corner is always $(1, 1)$.

Use a zero row and a zero column: $P[0][c] = P[r][0] = 0$. These represent empty rectangles and let the same formula handle the first row and first column.

Derive the construction formula by overlap

To build $P[r][c]$, combine the rectangle above it and the rectangle to its left, then add the current cell. The upper-left overlap was included in *both* rectangles, so subtract it once:

$$P[r][c] = a[r][c] + P[r-1][c] + P[r][c-1] - P[r-1][c-1].$$

This is **inclusion-exclusion**: add the desired pieces, then remove any duplicated overlap. The current cell is in neither earlier rectangle, so it must also be added.

Trace a small rectangular matrix

Input matrix a					Padded prefix P					
$r \backslash c$	1	2	3	4	$r \backslash c$	0	1	2	3	4
1	2	1	-1	3	0	0	0	0	0	0
2	0	4	2	-2	1	0	2	3	2	5
3	5	-3	1	2	2	0	2	7	8	9
					3	0	7	9	11	14

For example, $P[2][2] = 4 + 3 + 2 - 2 = 7$. Directly summing its four cells gives $2 + 1 + 0 + 4 = 7$, confirming the meaning.

Why row-major construction works

When we visit rows from top to bottom and columns from left to right, the upper entry, left entry, and upper-left entry have already been computed. There is no dependency on a future cell.

The table needs $(R+1)(C+1)$ entries for an R -by- C matrix. Preparing it takes $O(RC)$ time and $O(RC)$ storage for positive dimensions. A large single dimension is not the only concern: the *product* of the dimensions determines memory.

Pause and predict: 24

Why is the upper-left prefix subtracted when building a cell, rather than added?

Check your prediction. It is already counted in both the upper rectangle and the left rectangle. Subtracting one copy leaves it counted exactly once.

6.25 Rectangle queries and marked-cell counts

Cut away everything above and to the left

A query rectangle has upper-left corner (r_1, c_1) and lower-right corner (r_2, c_2) , with both endpoints included.

Start with $P[r_2][c_2]$. Subtract the rows above the rectangle, $P[r_1 - 1][c_2]$, and the columns to its left, $P[r_2][c_1 - 1]$. Their common upper-left region has now been subtracted twice, so add it back once:

$$\begin{aligned} \text{rectSum} = & P[r_2][c_2] - P[r_1 - 1][c_2] \\ & - P[r_2][c_1 - 1] + P[r_1 - 1][c_1 - 1]. \end{aligned}$$

For rows 2 through 3 and columns 2 through 4 of the previous matrix, this gives $14 - 5 - 7 + 2 = 4$. The six cells are 4, 2, -2, -3, 1, 2, which also sum to 4.

Contract. Read rows cols q, the matrix in row order, then queries r1 c1 r2 c2. Use $1 \leq R, C \leq 500$, $0 \leq q \leq 200000$, and $|a_{r,c}| \leq 10^9$. All corners form valid nonempty rectangles. Print one sum per query.

Complete program: static rectangle sums

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0, q = 0;
    std::cin >> rows >> cols >> q;
    std::vector<std::vector<long long>> pref(
        rows + 1, std::vector<long long>(cols + 1, 0));
    for (int r = 1; r <= rows; ++r) {
        for (int c = 1; c <= cols; ++c) {
            long long value = 0;
            std::cin >> value;
            pref[r][c] = value + pref[r - 1][c]
                + pref[r][c - 1] - pref[r - 1][c - 1];
        }
    }
    while (q--) {
        int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
        std::cin >> r1 >> c1 >> r2 >> c2;
        const long long answer = pref[r2][c2]
            - pref[r1 - 1][c2] - pref[r2][c1 - 1]
            + pref[r1 - 1][c1 - 1];
        std::cout << answer << '\n';
    }
}
```

Sample input	Sample output
3 4 3	4
2 1 -1 3	14
0 4 2 -2	-1
5 -3 1 2	
2 2 3 4	
1 1 3 4	
1 3 1 3	

Check the formula at the boundaries

If $r_1 = 1$, the term for rows above uses row 0 and is zero. If $c_1 = 1$, the term for columns to the left uses column 0 and is zero. A one-cell rectangle still uses the same formula. Test a one-row matrix and a one-column matrix too; square-only tests can hide swapped dimensions.

Construction and queries together take $O(RC + q)$ time. Each rectangle sum takes $O(1)$, regardless of its area. The prefix matrix uses $O(RC)$ storage.

Counting marked cells is the same rectangle sum

Suppose a grid contains # for a marked cell and . for an unmarked cell. Give each marked cell the value 1 and each other cell the value 0. A rectangle's sum is then its number of marked cells.

Contract. Read `rows cols q`, then `rows` strings of length `cols`, using only # and .. Use the same dimension and query bounds. Print the marked-cell count for each rectangle.

Complete program: count marked cells inside rectangles

```

#include <iostream>
#include <string>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0, q = 0;
    std::cin >> rows >> cols >> q;
    std::vector<std::vector<int>>> pref(
        rows + 1, std::vector<int>(cols + 1, 0));
    for (int r = 1; r <= rows; ++r) {
        std::string row;
        std::cin >> row;
        for (int c = 1; c <= cols; ++c) {
            const int marked = (row[c - 1] == '#');
            pref[r][c] = marked + pref[r - 1][c]
                + pref[r][c - 1] - pref[r - 1][c - 1];
        }
    }
    while (q--) {
        int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
        std::cin >> r1 >> c1 >> r2 >> c2;
        std::cout << pref[r2][c2] - pref[r1 - 1][c2]
            - pref[r2][c1 - 1] + pref[r1 - 1][c1 - 1]
            << '\n';
    }
}

```

Sample input**Sample output**

3 4 2

4

#..#

2

.##.

...#

1 2 3 4

2 1 2 4

The row string is zero-based, so the character for one-based column c is `row[c - 1]`. The largest count is $RC \leq 250000$, so `int` is sufficient for this counting table. Ordinary numeric matrix sums use `long long` instead.

6.26 Fresh summaries for multiple test cases

A summary belongs to one particular input

A prefix array summarizes one array, not every array the program will ever read. A frequency table counts one case's observations, and a difference table contains one case's update markers. Reusing old entries without clearing them mixes unrelated cases.

The simplest approach is to create local vectors inside `solve()`. Each call constructs the needed size and starts at the requested initial value.

Contract. Read $1 \leq T \leq 200000$. Each case contains n q , its values, and its range queries. Every case has $n \geq 1$, values of magnitude at most 10^9 , and valid one-based ranges. Across all cases, $\sum n \leq 200000$ and $\sum q \leq 200000$. Print query answers in input order without extra separators.

Complete program: independent range-query cases

```
#include <iostream>
#include <vector>

void solve() {
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int tests = 0;
    std::cin >> tests;
    while (tests--) solve();
}
```

Sample input	Sample output
2	6
3 1	-7
1 2 3	-7
1 3	
1 2	
-7	
1 1	
1 1	

Account for the sum of all case sizes

The total time is $O(T + \sum n + \sum q)$. Since each case has at least one element, $T \leq \sum n$, but showing the case loop makes its work explicit. Peak vector storage is $O(\max n)$, because one case's local vector is destroyed before the next call.

For frequency arrays, a full reset of a domain of size $U + 1$ per case adds $O(T(U + 1))$ work if the same fixed domain is always allocated. A small total number of observations does not erase that initialization cost. For sparse large domains, a suitable map or a carefully managed list of touched indices may be preferable. Keep the local-vector version until the constraints require another design.

Pause and predict: 25

Case 1 has ten occurrences of value 7. Case 2 contains only value 3. What should `freq[7]` mean in case 2?

Check your prediction. Zero. The two cases are independent, so case 1's observations must not survive in case 2's counts.

6.27 Debug by checking meanings and small counterexamples

Match a symptom to a possible cause

Symptom	First thing to inspect
Single-element sums become zero.	Did you subtract <code>pref[l]</code> instead of <code>pref[l - 1]</code> ?
First-position queries fail.	Is the empty prefix present and initialized?
Last-position updates crash.	Is <code>diff[r + 1]</code> allocated when <code>r == n</code> ?
Overlaps lose earlier additions.	Did you assign boundary markers instead of accumulating them?
Final values are too large.	Did reconstruction already include the original array before you added it again?
Negative tests fail, positive tests pass.	Check signed counters, wide totals, and an incorrectly zero-initialized maximum.
A frequency access crashes.	Is the domain sized by values, and are all offsets applied?
Only later test cases fail.	Look for summaries or running values not reset between cases.
Rectangles fail near the top or left.	Check the padding and the added-back overlap term.

Some code can look plausible but mean the wrong thing

```
// Wrong example: reserve changes capacity, not vector size.
std::vector<long long> pref;
pref.reserve(n + 1);
pref[0] = 0; // No element exists at index 0.
```

Use a sized vector, such as `vector<long long> pref(n + 1, 0)`. This is the same size-versus-capacity distinction from Session 4.

```
// Wrong example for an inclusive update:
diff[l] = delta; // Overwrites earlier starts.
diff[r] -= delta; // Stops before the right endpoint.
```

The correct markers are `diff[l] += delta` and `diff[r + 1] -= delta`, using an adequately sized vector.

Useful invariants to check on paper

For numeric prefixes, verify $P[0] = 0$, $P[i] - P[i - 1] = a_i$, and $P[n] = \sum a_i$. For a frequency table, all counts are nonnegative and their sum is the number of observations. For an update-only difference array, reconstruct and compare against direct interval additions on a tiny array.

With both endpoints recorded, the sum of *all* update markers through the sentinel $n + 1$ is zero: each $+\Delta$ has a corresponding $-\Delta$. The running added amount therefore returns to zero after the last real position. This is a check for update markers, not a requirement that the final real values sum to zero.

Differential testing: compare two ways to answer

Write the simple direct solution for tiny inputs. Generate many short arrays and ranges, then compare its output with the prefix or difference solution. For updates, explicitly add to every covered cell in the reference solution. For rectangles, explicitly visit every requested cell.

When a mismatch appears, keep that input and reduce it. A failing three-element example is easier to understand than a large random file. Include $l = r$, $l = 1$, $r = n$, zero updates, repeated updates, negative updates, exact cancellation, and rectangular rather than only square grids.

The source package contains an automated Python checker using these direct reference calculations. It also checks the printed samples and large type-boundary cases. Passing those checks is evidence about tested inputs, not a substitute for explaining the algorithm.

6.28 Optional extension: C++ numeric-library helpers

Partial sum is a library operation, not a range-update table

In `<numeric>`, `std::partial_sum` writes running totals. With input 4, −2, 7, it writes 4, 2, 9. To keep our leading empty prefix, write its results starting at `pref.begin() + 1`. Allocate the output entries first.^[6-4]

Its accumulator uses the **input iterator's value type**. Reading from `vector<int>` and writing into `vector<long long>` does not by itself make the internal accumulation wide. Use `long long` input values when totals need that width.^[6-4]

Related helpers

`std::adjacent_difference` writes the first input value, then consecutive differences. Taking partial sums of that result reconstructs the input. Both ranges in the example have sufficient output storage.^[6-5]

`std::accumulate` produces one final total rather than all prefixes. The initial value determines its accumulator type; 0LL requests a `long long` accumulator.^[6-6]

Contract. Read $0 \leq n \leq 200000$ and values of magnitude at most 10^9 . Print all padded prefix entries, the consecutive differences, and the final total on three lines. The differences line is empty when $n = 0$.

Complete program: running totals, differences, and one final sum

```
#include <iostream>
#include <numeric>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> a(n);
    for (long long& value : a) std::cin >> value;
    std::vector<long long> pref(n + 1, 0), diff(n, 0);
    std::partial_sum(a.begin(), a.end(), pref.begin() + 1);
    std::adjacent_difference(a.begin(), a.end(), diff.begin());
    for (int i = 0; i <= n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << pref[i];
    }
    std::cout << '\n';
    for (int i = 0; i < n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << diff[i];
    }
    std::cout << '\n';
    std::cout << std::accumulate(a.begin(), a.end(), 0LL)
                << '\n';
}
```

Sample input	Sample output
5	0 4 2 9 9 12
4 -2 7 0 3	4 -6 9 -7 3
	12

Prefer the explicit loop while learning the idea

The helpers shorten the syntax but do not remove the need to understand the output size, the numeric type, or the meaning of each result. Calling `partial_sum` alone does not mark any interval additions; those boundary changes must still be recorded by your algorithm.

Pause and predict: 26

Why is a `long long` destination not enough to make `partial_sum` safe when the input is a `vector<int>` with a very large total?

Check your prediction. The internal running accumulator has the input value type, so it can overflow before a value is written to the wider destination. Use wide input values or an explicitly wide accumulation loop.

6.29 Optional extension: weighted and modular sums

Weights relative to the requested range

Suppose query $[l, r]$ asks for

$$a_l + 2a_{l+1} + 3a_{l+2} + \cdots + (r - l + 1)a_r.$$

The weight of position i is $i - l + 1$, not simply i . Build two prefix arrays: ordinary totals P and globally weighted totals $W[i] = \sum_{j=1}^i ja_j$. Then

$$\begin{aligned} \sum_{i=l}^r (i - l + 1)a_i &= \sum_{i=l}^r ia_i - (l - 1) \sum_{i=l}^r a_i \\ &= (W[r] - W[l - 1]) - (l - 1)(P[r] - P[l - 1]). \end{aligned}$$

The derivation shifts each global weight down by $l - 1$, making the first queried weight equal to 1.

Contract. Read n q , the array, then valid range queries. Use $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, and $|a_i| \leq 10^6$. Print the relative-weighted sum per query.

Complete program: two prefixes for relative weights

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> pref(n + 1, 0), weighted(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
        weighted[i] = weighted[i - 1] + i * value;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        const long long sum = pref[r] - pref[l - 1];
        const long long weightedSum = weighted[r] - weighted[l - 1];
        std::cout << weightedSum - (l - 1LL) * sum << '\n';
    }
}
```

Sample input

```
5 3
4 -2 7 0 3
2 5
1 3
4 4
```

Sample output

```
24
21
0
```

Since `value` is already `long long`, `i * value` is wide. The tighter 10^6 input bound keeps the weighted totals and intermediate products within the tested 64-bit range. Build time is $O(n)$; query time is $O(1)$; storage is $O(n)$.

A sum modulo a positive integer

For a sum modulo M , store normalized prefixes in $[0, M)$. A C++ remainder can be negative, so the difference also needs normalization.^[6-8]

Normalize each signed input with $(\text{value} \% \text{mod} + \text{mod}) \% \text{mod}$. Prefixes then lie in $[0, M)$, so adding one M to their difference before taking the remainder gives a nonnegative answer.

Contract. Read n q mod , the array, then the valid range queries. Use $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, $|a_i| \leq 10^9$, and $1 \leq M \leq 10^9$. Print each range sum modulo M as a nonnegative residue.

Complete program: normalized modular range sums

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    long long mod = 0;
    std::cin >> n >> q >> mod;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        value = (value % mod + mod) % mod;
        pref[i] = (pref[i - 1] + value) % mod;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        const long long answer =
            (pref[r] - pref[l - 1] + mod) % mod;
        std::cout << answer << '\n';
    }
}
```

Sample input

```
4 3 5
-3 8 -9 2
1 4
2 3
3 3
```

Sample output

```
3
4
1
```

For example, -1 modulo 5 is reported as 4 , because -1 and 4 differ by a multiple of 5 . When $M = 1$, every answer is 0 . A modulus of zero is not allowed. Reducing arithmetic modulo M does not excuse overflow in a product that happens before the reduction; our bounds keep the operations here safe.

6.30 Optional extension: sparse values and compression

Keep order without reserving every possible key

If only a few different values appear but their magnitudes are large, direct indexing wastes space. **Coordinate compression** assigns each distinct value its position in a sorted list of distinct keys.

For values $-10^9, 7, 7, 10^9, 50$, the sorted distinct keys are $-10^9, 7, 50, 10^9$, with IDs 0, 1, 2, 3. We can count the two occurrences of 7 in `freq[1]` and build a prefix over the four frequencies.

A query boundary need not appear in the keys. `lower_bound(low)` finds the first key at least `low`; `upper_bound(high)` finds the first key greater than `high`. Their positions define a half-open interval of compressed IDs. Binary-search bounds require a suitably ordered range, so sorting happens before these lookups.^[6-7]

Contract. Read n q , the values, then value intervals `low high`. Use $0 \leq n, q \leq 200000$, values and endpoints in $[-10^9, 10^9]$, and $L \leq R$. Print how many input occurrences lie in each value interval.

Complete program: frequencies over compressed keys

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> a(n);
    for (long long& value : a) std::cin >> value;
    std::vector<long long> keys = a;
    std::sort(keys.begin(), keys.end());
    keys.erase(std::unique(keys.begin(), keys.end()), keys.end());
    const int distinct = static_cast<int>(keys.size());
    std::vector<int> freq(distinct, 0), pref(distinct + 1, 0);
    for (long long value : a) {
        const auto it = std::lower_bound(keys.begin(), keys.end(), value);
        const int id = static_cast<int>(it - keys.begin());
        ++freq[id];
    }
    for (int i = 0; i < distinct; ++i)
        pref[i + 1] = pref[i] + freq[i];
    while (q--) {
        long long low = 0, high = 0;
        std::cin >> low >> high;
        const auto first = std::lower_bound(keys.begin(), keys.end(), low);
        const auto last = std::upper_bound(keys.begin(), keys.end(), high);
        const int left = static_cast<int>(first - keys.begin());
        const int right = static_cast<int>(last - keys.begin());
        std::cout << pref[right] - pref[left] << '\n';
    }
}
```

Sample input	Sample output
5 3	3
-10000000000 7 7 10000000000 50	0
7 50	1
8 49	
-10000000000 -10000000000	

Meaning and cost

`left` is the first compressed ID allowed by the lower boundary. `right` is the first ID excluded by the upper boundary. Subtracting `pref[right] - pref[left]` counts all occurrences in that half-open ID interval. Empty intervals and an empty input list give zero naturally.

Sorting and assigning frequencies take $O(n \log(n + 1))$ time; each query takes $O(\log(d + 1))$ for d distinct keys. Storage is $O(n + d)$, rather than proportional to the largest numeric key. Keeping $+1$ avoids a meaningless logarithm of zero for an empty key list.

Compression does not preserve distance

The difference between IDs 1 and 2 is one, but the difference between original values 7 and 50 is 43. Compression preserves equality and sorted order, not numeric gaps, lengths, or area. A geometric interval-length problem must retain original coordinate distances.

For *only* raw occurrence counts in value intervals, sorting all input values, keeping duplicates, and subtracting two bound positions is an even simpler alternative. Compression becomes useful when other data is attached to each distinct key or a compact indexed domain is needed.

Pause and predict: 27

A query is $[8, 49]$, but neither boundary appears among keys $-10^9, 7, 50, 10^9$. Should the boundaries be replaced with their nearest keys?

Check your prediction. No. Use the lower and upper bounds as defined. They both identify the boundary before 50 here, so the compressed interval is empty and its count is zero.

6.31 Optional extension: rectangle additions with four markers

Turn one-dimensional start/stop markers into corners

For a zero-filled grid, add x to every cell in rectangle $[r_1, r_2] \times [c_1, c_2]$. Store four changes:

$$\begin{aligned} D[r_1][c_1] &+= x, & D[r_2 + 1][c_1] &-= x, \\ D[r_1][c_2 + 1] &-= x, & D[r_2 + 1][c_2 + 1] &+= x. \end{aligned}$$

A start marker contributes to every cell down and to its right after two-dimensional prefix reconstruction. The two negative markers stop that effect beyond the bottom and beyond the right edge. The final positive marker restores the region that both negative markers canceled.

For one rectangle, its reconstructed contribution is x inside both coordinate ranges and zero outside. Multiple rectangle additions then combine by ordinary addition.

Contract. Read `rows cols updates`, then `r1 c1 r2 c2 delta` updates. Use $1 \leq R, C \leq 500$, $0 \leq u \leq 200000$, and $|\Delta| \leq 10^9$. All rectangles are valid and inclusive. Print the final grid row by row.

Complete program: batch additions to a zero grid

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0, updates = 0;
    std::cin >> rows >> cols >> updates;
    std::vector<std::vector<long long>> diff(
        rows + 2, std::vector<long long>(cols + 2, 0));
    for (int j = 0; j < updates; ++j) {
        int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
        long long delta = 0;
        std::cin >> r1 >> c1 >> r2 >> c2 >> delta;
        diff[r1][c1] += delta;
        diff[r2 + 1][c1] -= delta;
        diff[r1][c2 + 1] -= delta;
        diff[r2 + 1][c2 + 1] += delta;
    }
    for (int r = 1; r <= rows; ++r) {
        for (int c = 1; c <= cols; ++c) {
            diff[r][c] += diff[r - 1][c] + diff[r][c - 1]
                - diff[r - 1][c - 1];
            if (c > 1) std::cout << ' ';
            std::cout << diff[r][c];
        }
        std::cout << '\n';
    }
}
```

Sample input	Sample output
2 3 2	4 4 0
1 1 2 2 4	4 3 -1
2 2 2 3 -1	

Reconstruction and storage

The nested loop takes the two-dimensional prefix sum of the marker table *in place*. Once a real cell is visited, that entry changes meaning: it is now the reconstructed final value. Do not append more raw update markers to that partially reconstructed representation.

The grid has two extra rows and columns so all bottom/right cancellation corners are valid. Only rows 1 through R and columns 1 through C are output. Time is $O(RC + u)$; storage is $O(RC)$. This handles additive updates in a batch, not arbitrary rectangle assignments.

6.32 Optional challenge: count target-sum subarrays

Use a prefix as a key

A **subarray** is a consecutive, nonempty range of positions. Suppose we want the number of subarrays whose sum is K , rather than answers for a supplied list of ranges.

A subarray $[l, r]$ has sum K exactly when

$$P[r] - P[l - 1] = K, \quad \text{or equivalently} \quad P[l - 1] = P[r] - K.$$

While scanning a right endpoint r , count how many *earlier* prefix totals equal $P[r] - K$. Those earlier prefixes identify all valid left boundaries. Because numeric prefixes may be negative and far apart, use a map of frequencies rather than a direct array indexed by their values.

The empty prefix is an earlier boundary

Before reading any elements, prefix total 0 has already occurred once: the empty prefix $P[0]$. Initialize `seen[0] = 1` so subarrays starting at position 1 can be counted.

Look up the target earlier prefix **before** inserting the current prefix. Otherwise, when $K = 0$, the current prefix can match itself and count an empty subarray, which this task forbids.

Contract. Read n **target**, then n values, with $0 \leq n \leq 200000$ and $|a_i|, |K| \leq 10^9$. Print the number of nonempty target-sum subarrays. Signed values and repeated prefixes are allowed.

Complete program: count matching earlier prefix totals

```
#include <iostream>
#include <map>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    long long target = 0;
    std::cin >> n >> target;
    std::map<long long, long long> seen;
    seen[0] = 1;
    long long prefix = 0, answer = 0;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        prefix += value;
        const auto it = seen.find(prefix - target);
        if (it != seen.end()) answer += it->second;
        ++seen[prefix];
    }
    std::cout << answer << '\n';
}
```

Sample input	Sample output
5 3 1 2 -1 1 2	3

Trace the sample's three matches

For 1, 2, -1, 1, 2, the prefixes are 0, 1, 3, 2, 3, 5. With target 3, matches occur at right endpoints 2, 4, and 5. The corresponding subarrays are [1, 2], [1, 4], and [4, 5]. Each has sum 3.

A set would remember only whether an earlier prefix exists, not how many earlier boundaries have that same total. With several zeros or canceling positive/negative values, repeated prefix totals can create many distinct subarrays.

Reason and cost

Before reading the next value, **seen** counts exactly the prefixes ending before the new right endpoint. Its lookup therefore counts exactly the valid left boundaries for that endpoint. Each nonempty subarray has a unique right endpoint, so it is counted once.

With `std::map`, total time is $O(n \log(n+1))$ apart from constant setup, and storage is $O(n)$ in the worst case. The answer can reach $n(n+1)/2$, so use `long long`. This optional problem combines the prefix reasoning from this session with frequency maps from Session 5.

Pause and predict: 28

For input 0, 0, 0 and target 0, how many nonempty subarrays should be counted?

Check your prediction. There are $3 + 2 + 1 = 6$: three of length 1, two of length 2, and one of length 3. Frequencies of repeated zero prefixes are essential to count all of them.

6.33 Practice problems: choose the stored information

Try each task before reading its hint or solution. Each statement gives its own input contract; all promised indices and values are valid. Print exactly the requested answer, without prompts. Unless stated otherwise, ranges are one-based and inclusive.

Suggested route

Problems 1 to 5 build prefix reasoning. Problems 6 to 11 practise value-domain counts. Problems 12 to 15 cover interval additions and reconstruction. Problems 16 to 19 combine positions, grids, and counts. Problems 20 to 22 are optional challenges based on the extension sections.

Write four things before you code

State what each stored entry means, how you build it, how you obtain an answer, and why its size and numeric type fit the constraints. Then choose one smallest case and one boundary case to test.

1. Print the empty prefix too

Read $0 \leq n \leq 200000$, followed by n integers with $|a_i| \leq 10^9$. Print $n + 1$ values: the sum of zero input values, then the first one, the first two, and so on through all inputs. Print them on one line separated by spaces. For $n = 0$, output 0.

Sample input

Sample output

4

0 3 3 -1 5

3 0 -4 6

Hint: page 82. Explained solution: page 87.

2. A sum for each interval

Read n , q , then the array, then q queries l r . Use $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, $|a_i| \leq 10^9$, and $1 \leq l \leq r \leq n$. Print the sum of positions l through r for each query. The array never changes.

Sample input

Sample output

4 3

12

5 -1 2 6

1

1 4

6

2 3

4 4

Hint: page 82. Explained solution: page 88.

3. How many are positive?

Use the same input format and bounds as Problem 2. For each interval, print how many stored values are strictly greater than zero. Zero is not positive. The query asks for a count, not the sum of the positive values.

Sample input	Sample output
6 3	3
0 2 -3 5 -1 4	2
1 6	0
2 5	
1 1	

Hint: page 82. Explained solution: page 89.

4. Choose the best fixed-length block

Read n k , then the array, with $1 \leq k \leq n \leq 200000$ and $|a_i| \leq 10^9$. Among all blocks of exactly k consecutive values, print the greatest sum and the smallest one-based starting position that achieves it.

Sample input	Sample output
6 3	5 1
2 -1 4 0 -2 5	

Hint: page 82. Explained solution: page 90.

5. Equal sums on both sides

Read $1 \leq n \leq 200000$ and the signed array, with $|a_i| \leq 10^9$. Print the first position whose strictly-left sum equals its strictly-right sum, or -1 if none exists. Do not include the value at the candidate position in either side. Empty sides have sum zero.

Sample input

Sample output

4

1

0 2 -2 0

Hint: page 82. Explained solution: page 91.

6. Count a queried value

Read **n upper q**, then n values and q queried values. Use $0 \leq n, U, q \leq 200000$, with all values and queries in $[0, U]$. Print the occurrence count of each queried value. Missing values have count zero, and queries do not change the observations.

Sample input

Sample output

5 4 3

3

0 4 1 4 4

0

4

1

3

0

Hint: page 82. Explained solution: page 92.

7. Distinct count and smallest mode

Read n upper and n values in $[0, U]$. Use $1 \leq n \leq 200000$, $0 \leq U \leq 200000$. Print the number of distinct values on the first line. On the second line, print the smallest value with maximum frequency, followed by that frequency.

Sample input

6 4
3 1 3 1 4 0

Sample output

4
1 2

Hint: page 82. Explained solution: page 93.

8. Same letters with the same multiplicities

Read two nonempty lowercase ASCII words, each with at most 200000 characters. Print YES if one can be rearranged into the other, otherwise NO. Letter counts must match, not only the set of letters.

Sample input

aabb abbb

Sample output

NO

Hint: page 82. Explained solution: page 94.

9. Equal values at different positions

Read n **upper** and values in $[0, U]$, with $0 \leq n, U \leq 200000$. Count pairs of positions (i, j) such that $i < j$ and $a_i = a_j$. Print the number of pairs once. Different occurrences are different positions even when their values match.

Sample input
Sample output

5 3

6

2 2 2 2 0

Hint: page 83. Explained solution: page 95.

10. How many observations fall in a value range?

Read n **upper** q , the observations, then queries **low high**. Use $0 \leq n, U, q \leq 200000$, $0 \leq a_i \leq U$, and $0 \leq L \leq R \leq U$. Print the number of original observations whose values are between L and R , inclusive. Query endpoints are values, not positions.

Sample input
Sample output

7 6 3

5

0 2 2 6 4 2 0

1

0 2

1

3 5

6 6

Hint: page 83. Explained solution: page 96.

11. Frequencies across zero

Read n q , the observations, then queried values. Use $0 \leq n, q \leq 200000$. Every observation and queried value lies in $[-1000, 1000]$. Print each queried frequency. Negative and positive values with the same absolute value are distinct keys.

Sample input	Sample output
4 3	2
-1000 1000 -1000 0	0
-1000	1
1	
1000	

Hint: page 83. Explained solution: page 97.

12. Add to many intervals

Start with n zeros. Read n **updates**, then u triples l r **delta**. Use $1 \leq n \leq 200000$, $0 \leq u \leq 200000$, $|\Delta| \leq 10^9$, and $1 \leq l \leq r \leq n$. Add the given amount to every covered position. Print all final values after every update has been recorded.

Sample input	Sample output
4 2	5 3 3 5
1 4 5	
2 3 -2	

Hint: page 83. Explained solution: page 98.

13. Add to an existing array

Read n updates, an initial array, then u valid interval additions. Use $1 \leq n \leq 200000$, $0 \leq u \leq 200000$, and $|a_i|, |\Delta| \leq 10^9$. Print the final array after all additions. The initial values are not necessarily zero.

Sample input

```
4 2
1 2 3 4
1 2 10
2 4 -1
```

Sample output

```
11 11 2 3
```

Hint: page 83. Explained solution: page 99.

14. Find the first busiest slot

Read n intervals, then m valid inclusive intervals on numbered slots 1 through n . Use $1 \leq n \leq 200000$ and $0 \leq m \leq 200000$. Print the maximum coverage count and the smallest slot attaining it. If there are no intervals, print 0 1.

Sample input

```
6 3
1 3
2 5
5 6
```

Sample output

```
2 2
```

Hint: page 83. Explained solution: page 100.

15. Queries after every addition is complete

Read n updates q , the initial array, then all additions, then all range-sum queries. Use $1 \leq n \leq 200000$, $0 \leq u, q \leq 200000$, $|a_i| \leq 10^9$, $|\Delta| \leq 10^6$, and valid inclusive endpoints. Every query refers to the final array after all additions. Print one sum per query.

Sample input	Sample output
3 2 2	11
2 0 -1	5
1 3 4	
2 2 -2	
1 3	
2 3	

Hint: page 83. Explained solution: page 101.

16. Sum a rectangle

Read $rows$ $cols$ q , an integer matrix in row order, and queries $r1$ $c1$ $r2$ $c2$. Use $1 \leq R, C \leq 500$, $0 \leq q \leq 200000$, and $|a_{r,c}| \leq 10^9$. Query corners form valid one-based inclusive rectangles. Print each rectangle's sum.

Sample input	Sample output
2 3 2	0
1 -2 3	9
4 5 -6	
1 2 2 3	
2 1 2 2	

Hint: page 84. Explained solution: page 102.

17. Count marked cells in a rectangle

Read `rows cols q`, then a grid of `#` and `.`, then valid rectangle queries as in Problem 16. Use the same dimension and query bounds. Each row string has exactly C characters. Print how many `#` cells lie in each rectangle.

Sample input

```
2 3 2
#.#
.##
1 1 2 3
1 2 2 2
```

Sample output

```
4
1
```

Hint: page 84. Explained solution: page 103.

18. Count one letter inside a substring

Read a nonempty lowercase ASCII word of length at most 100000, then $0 \leq q \leq 200000$. Each query is `l r c`, where $1 \leq l \leq r \leq n$ and `c` is a lowercase letter. Print how many times that letter occurs in the requested substring.

Sample input

```
banana 3
1 6 a
2 5 n
1 1 a
```

Sample output

```
3
2
0
```

Hint: page 84. Explained solution: page 104.

19. How many slots meet the fixed threshold?

Read n intervals $threshold$ q , then the coverage intervals, then query intervals. Use $1 \leq n \leq 200000$, $0 \leq m, q \leq 200000$, $1 \leq k \leq 200001$, and valid inclusive endpoints. For each query, count slots covered by at least k original intervals. The same threshold is used for all queries.

Sample input

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

Sample output

```
4
2
```

Hint: page 84. Explained solution: page 105.

20. Optional: count all target-sum subarrays

Read n $target$, then the array, with $0 \leq n \leq 200000$ and $|a_i|, |K| \leq 10^9$. Print the number of nonempty consecutive subarrays summing to K . Negative values and repeated prefix totals are allowed. Use the optional prefix-frequency-map method.

Sample input

```
4 0
0 1 -1 0
```

Sample output

```
6
```

Hint: page 84. Explained solution: page 106.

21. Optional: weights start at one inside each query

Read n q , the array, then valid intervals. Use $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, and $|a_i| \leq 10^6$. For each $[l, r]$, print $a_l + 2a_{l+1} + \dots + (r - l + 1)a_r$. Weights restart at 1 for every query.

Sample input	Sample output
4 2	17
2 3 1 4	8
2 4	
1 2	

Hint: page 84. Explained solution: page 107.

22. Optional: batch additions on a zero grid

Start with an R -by- C zero grid. Read $rows$ $cols$ $updates$, then $r1$ $c1$ $r2$ $c2$ $delta$. Use $1 \leq R, C \leq 500$, $0 \leq u \leq 200000$, $|\Delta| \leq 10^9$, and valid inclusive rectangles. Add the amount to every covered cell. Print the final grid, one row per line.

Sample input	Sample output
3 2 2	2 0
1 1 3 1 2	5 3
2 1 2 2 3	2 0

Hint: page 84. Explained solution: page 108.

6.34 Trace, representation, and complexity drills

These exercises are solved on paper before running code. Write the meaning of every index and trace the entries that change. A correct number without a correct interpretation can hide a boundary error. Drill 12 uses the optional target-sum extension.

Drill 1: A prefix is allowed to decrease

The sequence is $-3, 5, 0, -2$, in that order. Use one-based input positions and $P[k]$ = the sum of the first k values.

1. Write all five entries of P , including the empty prefix.
2. Use two entries to find the sum on $[2, 4]$.
3. A student says that a prefix array must be sorted. Explain whether that statement is true for these inputs.

Hint: page 85. Worked answer: page 109.

Drill 2: Translate endpoints before calculating

The zero-based vector is $\mathbf{a} = \{2, 5, -1, 4\}$. A query specifies the *half-open* interval $[1, 3)$. Define P by prefix length, with $P[0] = 0$.

1. Which values are included? What is the equivalent one-based inclusive interval?
2. Choose the correct formula: $P[3] - P[1]$, $P[3] - P[0]$, or $P[4] - P[1]$.
3. What would the half-open interval $[2, 2)$ contain?

Hint: page 85. Worked answer: page 109.

Drill 3: Position ranges and value ranges

An array has values 4, 1, 4, 0. All values lie in $[0, 4]$.

1. Write `freq[0]` through `freq[4]`.
2. Find the *sum at positions* $[1, 2]$.
3. Find the *number of observations with values* in $[1, 2]$.
4. Can the frequency array tell you the original position of the first 4? Explain.

Hint: page 85. Worked answer: page 110.

Drill 4: An offset is a reversible mapping

All values belong to the inclusive domain $[-3, 2]$. You choose `LOW = -3`, `HIGH = 2`, and index a frequency vector using `x - LOW`.

1. How many entries are needed?
2. Which index stores the count of -1 ?
3. Which original value corresponds to index 5?
4. Why is using `abs(x)` not a suitable replacement for this mapping?

Hint: page 85. Worked answer: page 110.

Drill 5: Trace both boundaries of every update

Start with five zeros. Apply $[1, 3] += 2$, then $[3, 5] += -1$. Use a difference array with an extra position 6 for cancellation.

1. Record every changed difference entry for each update.
2. Write $D[1]$ through $D[6]$ after both updates.
3. Reconstruct the five real values.
4. Explain why the entry at position 6 is not part of the output.

Hint: page 85. Worked answer: page 111.

Drill 6: Do not add the base array twice

The initial array is $[2, 2, 5]$. Add 4 to the inclusive range $[2, 3]$.

1. Find the final values directly.
2. Show the initial full differences, then their real entries after the update.
3. Alternatively, start an all-zero array of update differences. What running changes does it reconstruct?
4. A program reconstructs the *full* differences and then adds the original values. What incorrect output would it produce?

Hint: page 85. Worked answer: page 111.

Drill 7: When an old prefix stops describing the data

The array is $[1, 2, 3]$, with prefix array $[0, 1, 3, 6]$. Increase the second array element by 5, but do not rebuild the prefix array.

1. What does the old formula return for $[2, 3]$? What is the correct new answer?
2. Which prefix entries are now stale?
3. Can collecting *all* future updates and reconstructing once answer a query that must observe only updates already performed?

Hint: page 85. Worked answer: page 112.

Drill 8: Count initialization and hidden copies

Consider two separate implementations.

- A.** For each of 100 test cases, allocate and zero a frequency vector for $U = 1000000$, read 50 values, and answer 10 exact-count queries.
- B.** Build a prefix vector of length $n + 1$, but pass it *by value* to a helper for each of q queries.
1. What work does A repeatedly perform even before reading the observations?
 2. What is B's total query cost, despite the helper using only one subtraction in its body?
 3. State a suitable parameter type to avoid B's copies.

Hint: page 85. Worked answer: page 112.

Drill 9: A count of qualifying positions is not a coverage sum

Four positions have coverage counts $[2, 0, 3, 1]$. A position qualifies when its coverage is at least 2.

1. Build the 0/1 qualification array and its padded prefix array.
2. How many qualifying positions are in $[2, 4]$?
3. What would a prefix of the raw coverage counts return for the same interval?
4. Could the same qualification prefix directly answer a query with threshold 3 instead?

Hint: page 85. Worked answer: page 113.

Drill 10: A rectangle needs four boundaries

The matrix is

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Use $P[r][c]$ for the sum of the first r rows and first c columns, with a padded zero row and column.

1. Write the two nonzero rows of P .
2. Find the sum in rows 1 through 2 and columns 2 through 3 using four prefix entries.
3. Explain what happens to terms that touch the padded row or column.

Hint: page 85. Worked answer: page 113.

Drill 11: A wide destination does not widen an earlier calculation

This deliberately wrong fragment tries to build wide totals:

```
// Wrong example: the input element type is still int.
std::vector<int> a = {1000000000, 1000000000,
                    1000000000};
std::vector<long long> pref(4, 0);
std::partial_sum(a.begin(), a.end(), pref.begin() + 1);
```

Assume a 32-bit `int`. Explain the problem and repair it. Then compute how many unordered pairs of different positions exist among 100000 equal values. Does that answer fit in a 32-bit signed `int`?
Hint: page 85. Worked answer: page 114.

Drill 12: Optional: a prefix must look only into the past

Count nonempty subarrays with sum zero in $[0, 0]$. Start with `seen[0] = 1`, representing the empty prefix before reading input.

1. Trace `answer += seen[prefix]` followed by `++seen[prefix]` for both inputs.
2. What wrong total results if you increment `seen[prefix]` *before* looking it up?
3. What correct subarrays are missed if you omit the initial empty prefix instead?

Hint: page 85. Worked answer: page 114.

6.35 Hints before solutions

A hint should help you choose the next step without replacing your attempt. After reading one, close this page and try the problem again.

1. Print the empty prefix too

Begin with one known prefix, the empty prefix. Each new input extends the preceding total.

2. A sum for each interval

Which prefix contains the full requested interval, and which prefix contains only the part before it?

3. How many are positive?

Replace each value conceptually with 1 when it is positive and 0 otherwise.

4. Choose the best fixed-length block

Every start l defines a query ending at $l + k - 1$. Initialize from an actual window, not zero.

5. Equal sums on both sides

The left side ends just before i . The right side is the whole total minus everything through i .

6. Count a queried value

There are $U + 1$ possible values, irrespective of the number of observations.

7. Distinct count and smallest mode

Scan the frequency array in increasing value order. Think carefully about $>$ versus $>=$.

8. Same letters with the same multiplicities

Add the first word's letter counts and subtract the second's. What must every balance be at the end?

9. Equal values at different positions

When the next value arrives, every earlier equal occurrence creates one new pair.

10. How many observations fall in a value range?

First count exact values. Then prefix those frequencies, leaving space for the domain value zero.

11. Frequencies across zero

Translate the whole known domain so its smallest value receives index zero.

12. Add to many intervals

Turn an addition on at l , and turn it off just after r .

13. Add to an existing array

Keep the initial values and a separate, initially zero difference array for added amounts only.

14. Find the first busiest slot

Reconstruct interval coverage, then track a maximum while scanning slots from left to right.

15. Queries after every addition is complete

Difference markers solve the update phase. A prefix of the reconstructed final values solves the query phase.

16. Sum a rectangle

A prefix rectangle begins at $(1, 1)$. Remove the top and left strips, then restore their overlap.

17. Count marked cells in a rectangle

Treat a marked cell as 1 and an unmarked cell as 0, then use a rectangle sum.

18. Count one letter inside a substring

Keep a prefix count for each of the 26 possible letters, not just a whole-word count.

19. How many slots meet the fixed threshold?

Reconstruct coverage, convert it to a good/not-good indicator, and prefix those indicators.

20. Optional: count all target-sum subarrays

At the current right endpoint, an earlier prefix must equal current prefix $- K$. Count it before storing the current prefix.

21. Optional: weights start at one inside each query

Write each relative weight as $i - (l - 1)$. Split the resulting sum into two parts.

22. Optional: batch additions on a zero grid

Use four corner markers, including one row and column beyond the rectangle, then reconstruct with a two-dimensional prefix.

Hints for the twelve drills

Use these hints after attempting the questions. Full worked answers follow the coding solutions.

Drill 1. Start at zero, then add exactly one new input value per step. To remove position 1 from a total ending at position 4, subtract the prefix ending immediately before position 2.

Drill 2. A half-open right endpoint is excluded. The prefix index counts how many original values precede a boundary, not the original index of the last included value.

Drill 3. A frequency index is a possible value. It is not an original position. Consider which information survives when you rearrange the observations.

Drill 4. An inclusive integer domain has $\text{HIGH} - \text{LOW} + 1$ values. Reverse the mapping by adding LOW to an index.

Drill 5. The left boundary receives $+v$, and position $r + 1$ receives $-v$. For a negative update, stopping it means adding a positive cancellation.

Drill 6. Full differences already encode the base values. Update-only differences encode changes and therefore still need the base added after reconstruction.

Drill 7. The prefix is a summary of a particular version of the array. A change at position 2 affects every prefix containing position 2, but not shorter prefixes.

Drill 8. A sized vector initializes all its entries. A value parameter constructs a separate vector. Count that work even when it does not appear as an explicit loop in your code.

Drill 9. Apply the threshold to each reconstructed coverage count before adding it to the second prefix. This prefix stores counts of positions, not counts of covering intervals.

Drill 10. Subtract the rectangle above the query and the rectangle to its left. Their shared corner was removed twice, so add it back once.

Drill 11. The running accumulator of `partial_sum` uses the input iterator's value type. For the pair count, evaluate $n(n - 1)/2$ using wide arithmetic before multiplication.

Drill 12. A nonempty subarray requires an earlier prefix boundary. The current prefix must not count itself as an earlier boundary. The empty prefix is earlier than every real input.

6.36 Explained practice solutions

The following programs are complete and compile independently. Each solution explains the representation, why the algorithm works, its cost, and cases worth checking. A correct program is not the only possible program; use the explanation to understand the idea rather than memorizing its exact lines.

After studying a solution

Recreate a three-element example on paper. Then close the solution and write a fresh version. Explain why every index is valid and why the computed value matches the statement.

Solution 1: Print the empty prefix too

Idea. Store $P[0] = 0$. On reading value a_i , write $P[i] = P[i - 1] + a_i$, then output indices 0 through n .

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    for (int i = 0; i <= n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << pref[i];
    }
    std::cout << '\n';
}
```

Why it works. The initial entry represents no values. If entry $i - 1$ has the first $i - 1$ values' sum, adding the next value gives exactly the sum required for entry i .

Complexity. $O(n + 1)$ time and $O(n + 1)$ storage, including the empty-prefix entry.

Check the boundaries. Try no values, one negative value, and a total above the 32-bit range. In the sample, $3 + 0 - 4 = -1$, so prefixes need not be sorted.

Solution 2: A sum for each interval

Idea. Build a padded prefix vector once. Each query prints $P[r] - P[l - 1]$. Read and answer queries one at a time.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Why it works. The first term contains every value through r . The second removes exactly the values before l , leaving each requested value once.

Complexity. $O(n + q)$ time and $O(n)$ storage. Each query itself takes $O(1)$.

Check the boundaries. Test $[1, 1]$, $[n, n]$, the whole array, and $q = 0$. The sample query $[2, 3]$ sums $-1 + 2 = 1$.

Solution 3: How many are positive?

Idea. Build prefix counts using the contribution `value > 0`. Subtract at the query boundaries exactly as for sums.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<int> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + (value > 0);
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Why it works. Each positive value contributes one and every other value contributes zero. The difference of the two prefix counts therefore equals the number of qualifying positions in the interval.

Complexity. $O(n + q)$ time and $O(n)$ storage. Counts are at most n , so `int` is sufficient under these bounds.

Check the boundaries. Test zero, all-negative input, and a single positive position. In the sample, interval $[2, 5]$ contains exactly the two positive values 2 and 5.

Solution 4: Choose the best fixed-length block

Idea. Use prefixes to evaluate each valid window. Begin with the first window as the best, and replace it only on a strictly greater sum.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, k = 0;
    std::cin >> n >> k;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    long long best = pref[k];
    int start = 1;
    for (int l = 2; l + k - 1 <= n; ++l) {
        const int r = l + k - 1;
        const long long sum = pref[r] - pref[l - 1];
        if (sum > best) {
            best = sum;
            start = l;
        }
    }
    std::cout << best << ' ' << start << '\n';
}
```

Why it works. Every allowed start is considered once. Keeping a larger sum preserves the maximum; ignoring equal later sums preserves the earliest start among ties.

Complexity. $O(n)$ time and $O(n)$ storage for the prefix vector.

Check the boundaries. Test all-negative values, $k = 1$, $k = n$, and equal best windows. The sample's best block is 2, -1, 4, starting at 1 and summing to 5.

Solution 5: Equal sums on both sides

Idea. Scan positions in increasing order. Compare $P[i - 1]$ with $P[n] - P[i]$, stopping at the first equality.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    std::cin >> n;
    std::vector<long long> pref(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
    }
    int answer = -1;
    for (int i = 1; i <= n; ++i) {
        const long long left = pref[i - 1];
        const long long right = pref[n] - pref[i];
        if (left == right) {
            answer = i;
            break;
        }
    }
    std::cout << answer << '\n';
}
```

Why it works. The two expressions represent exactly the two sides without the pivot. The first match in increasing position order is the smallest qualifying position.

Complexity. $O(n)$ time and $O(n)$ storage. A running-left-total variant could use less auxiliary storage after the array is available.

Check the boundaries. A one-element array always returns 1. In the sample, the three right-side values at position 1 sum to zero, matching the empty left side.

Solution 6: Count a queried value

Idea. Allocate $U + 1$ zeros, increment the entry indexed by each input value, and directly read the queried entry.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0, q = 0;
    std::cin >> n >> upper >> q;
    std::vector<int> freq(upper + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value];
    }
    while (q--) {
        int value = 0;
        std::cin >> value;
        std::cout << freq[value] << '\n';
    }
}
```

Why it works. Each occurrence contributes one to exactly its value's entry. After all input, that entry is therefore the total number of matching observations.

Complexity. $O(U + n + q + 1)$ time and $O(U + 1)$ storage, including domain initialization.

Check the boundaries. Test $U = 0$, empty observations, and a query for an absent value. In the sample, 4 occurs three times while 3 never occurs.

Solution 7: Distinct count and smallest mode

Idea. Count positive frequency entries. Keep a best value while scanning from 0 through U , replacing it only when a strictly larger frequency is found.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0;
    std::cin >> n >> upper;
    std::vector<int> freq(upper + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value];
    }
    int best = 0, distinct = 0;
    for (int value = 0; value <= upper; ++value) {
        if (freq[value] > 0) ++distinct;
        if (freq[value] > freq[best]) best = value;
    }
    std::cout << distinct << '\n';
    std::cout << best << ' ' << freq[best] << '\n';
}
```

Why it works. Positive entries correspond one-to-one with distinct observed values. A later tied value is larger, so refusing equal replacements preserves the required smallest mode.

Complexity. $O(n + U + 1)$ time and $O(U + 1)$ storage.

Check the boundaries. Try all values equal, a missing value 0, and ties. The sample ties values 1 and 3 at frequency 2; the required mode is 1.

Solution 8: Same letters with the same multiplicities

Idea. Use 26 signed counters. Map a letter to $c - 'a'$, add first-word occurrences, subtract second-word occurrences, and check that every entry is zero.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string a, b;
    std::cin >> a >> b;
    std::vector<int> balance(26, 0);
    for (char c : a) ++balance[c - 'a'];
    for (char c : b) --balance[c - 'a'];
    bool same = true;
    for (int count : balance) {
        if (count != 0) same = false;
    }
    std::cout << (same ? "YES" : "NO") << '\n';
}
```

Why it works. A zero balance for every possible letter means neither word has an extra occurrence of any letter. Their multisets of letters are identical, which is precisely the rearrangement requirement.

Complexity. For lengths n, m , $O(n + m + 26)$ time. Counters use $O(26)$ extra space; stored input strings use $O(n + m)$.

Check the boundaries. Test repeated letters, different lengths, and one-character words. The sample fails because it contains different counts of **a** and **b**.

Solution 9: Equal values at different positions

Idea. Add the already-seen frequency to the answer, then increment that frequency. Store the answer as long long.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0;
    std::cin >> n >> upper;
    std::vector<long long> freq(upper + 1, 0);
    long long pairs = 0;
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        pairs += freq[value];
        ++freq[value];
    }
    std::cout << pairs << '\n';
}
```

Why it works. Each pair is counted at its later endpoint. Looking up the count before inserting the current item prevents counting an item paired with itself.

Complexity. $O(n + U + 1)$ time and $O(U + 1)$ storage.

Check the boundaries. Test no values, all-distinct values, and 200000 equal values. Four copies of 2 in the sample create $\binom{4}{2} = 6$ pairs.

Solution 10: How many observations fall in a value range?

Idea. Build $C[v+1] = C[v] + \text{freq}[v]$, with $C[0] = 0$. A value interval is answered by $C[R+1] - C[L]$.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, upper = 0, q = 0;
    std::cin >> n >> upper >> q;
    std::vector<int> freq(upper + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value];
    }
    std::vector<int> pref(upper + 2, 0);
    for (int value = 0; value <= upper; ++value)
        pref[value + 1] = pref[value] + freq[value];
    while (q--) {
        int low = 0, high = 0;
        std::cin >> low >> high;
        std::cout << pref[high + 1] - pref[low] << '\n';
    }
}
```

Why it works. $C[R+1]$ counts every observation at most R . $C[L]$ removes exactly the observations below L , leaving those within the inclusive value interval.

Complexity. $O(n + U + q + 1)$ time and $O(U + 1)$ storage.

Check the boundaries. Test $[0, 0]$, $[U, U]$, and an empty observed domain. The sample's $[0, 2]$ counts two zeros and three twos, giving 5.

Solution 11: Frequencies across zero

Idea. Use the index $x - (-1000) = x + 1000$ in a 2001-entry frequency vector for both input increments and queries.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    constexpr int LOW = -1000;
    constexpr int HIGH = 1000;
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<int> freq(HIGH - LOW + 1, 0);
    for (int i = 0; i < n; ++i) {
        int value = 0;
        std::cin >> value;
        ++freq[value - LOW];
    }
    while (q--) {
        int value = 0;
        std::cin >> value;
        std::cout << freq[value - LOW] << '\n';
    }
}
```

Why it works. Adding a fixed offset maps each allowed integer to a unique valid index, without merging signs or values. Each mapped frequency therefore counts exactly one original key.

Complexity. $O(n + q + 2001)$ time and $O(2001)$ storage. The domain is fixed in this task.

Check the boundaries. Test both endpoints -1000 and 1000 , zero, and a missing query. Never use a negative value directly as a vector index.

Solution 12: Add to many intervals

Idea. Accumulate markers at `diff[l]` and `diff[r + 1]`, then scan their running sum to print each final value.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, updates = 0;
    std::cin >> n >> updates;
    std::vector<long long> diff(n + 2, 0);
    for (int j = 0; j < updates; ++j) {
        int l = 0, r = 0;
        long long delta = 0;
        std::cin >> l >> r >> delta;
        diff[l] += delta;
        diff[r + 1] -= delta;
    }
    long long current = 0;
    for (int i = 1; i <= n; ++i) {
        current += diff[i];
        if (i > 1) std::cout << ' ';
        std::cout << current;
    }
    std::cout << '\n';
}
```

Why it works. A single update contributes its amount exactly between its two markers. Summing all markers combines the independent effects of all covering updates.

Complexity. $O(n + u)$ time and $O(n)$ storage, including the sentinel beyond position n .

Check the boundaries. Test zero updates, a full-range update, negative amounts, overlaps, and cancellation. The sample adds 5 everywhere, then removes 2 from positions 2 and 3.

Solution 13: Add to an existing array

Idea. Reconstruct the total added amount at each position, then print the original value plus that amount.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, updates = 0;
    std::cin >> n >> updates;
    std::vector<long long> a(n + 1, 0);
    for (int i = 1; i <= n; ++i) std::cin >> a[i];
    std::vector<long long> diff(n + 2, 0);
    for (int j = 0; j < updates; ++j) {
        int l = 0, r = 0;
        long long delta = 0;
        std::cin >> l >> r >> delta;
        diff[l] += delta;
        diff[r + 1] -= delta;
    }
    long long added = 0;
    for (int i = 1; i <= n; ++i) {
        added += diff[i];
        if (i > 1) std::cout << ' ';
        std::cout << a[i] + added;
    }
    std::cout << '\n';
}
```

Why it works. The separate markers represent only updates. Their running sum includes precisely the covering additions, so adding the unchanged original value produces the final value without double-counting.

Complexity. $O(n + u)$ time and $O(n)$ storage.

Check the boundaries. With no updates, output must equal the original array. In the sample, position 2 becomes $2 + 10 - 1 = 11$.

Solution 14: Find the first busiest slot

Idea. Use $+1$ and -1 boundary markers. During reconstruction, keep the greatest coverage and update its position only on a strict improvement.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, intervals = 0;
    std::cin >> n >> intervals;
    std::vector<int> diff(n + 2, 0);
    for (int j = 0; j < intervals; ++j) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        ++diff[l];
        --diff[r + 1];
    }
    int active = 0, best = -1, position = 1;
    for (int i = 1; i <= n; ++i) {
        active += diff[i];
        if (active > best) {
            best = active;
            position = i;
        }
    }
    std::cout << best << ' ' << position << '\n';
}
```

Why it works. The running sum equals each slot's coverage. Every slot is considered, and the first maximum survives ties because later equal values do not replace it.

Complexity. $O(n + m)$ time and $O(n)$ storage.

Check the boundaries. Test no intervals, all intervals identical, and peaks at both ends. The sample has maximum 2 at slots 2, 3, and 5, so it reports slot 2.

Solution 15: Queries after every addition is complete

Idea. Record all additions. Scan once to recover each final value and immediately extend its prefix total. Then answer ordinary range sums.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, updates = 0, q = 0;
    std::cin >> n >> updates >> q;
    std::vector<long long> a(n + 1, 0);
    for (int i = 1; i <= n; ++i) std::cin >> a[i];
    std::vector<long long> diff(n + 2, 0);
    for (int j = 0; j < updates; ++j) {
        int l = 0, r = 0;
        long long delta = 0;
        std::cin >> l >> r >> delta;
        diff[l] += delta;
        diff[r + 1] -= delta;
    }
    std::vector<long long> pref(n + 1, 0);
    long long added = 0;
    for (int i = 1; i <= n; ++i) {
        added += diff[i];
        pref[i] = pref[i - 1] + a[i] + added;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << pref[r] - pref[l - 1] << '\n';
    }
}
```

Why it works. The reconstruction yields the correct value after all updates at each position. Prefixing those final values, rather than the old array or raw markers, makes each subsequent subtraction a final-array sum.

Complexity. $O(n + u + q)$ time and $O(n)$ storage. The tighter update bound keeps final-array totals within `long long`.

Check the boundaries. Test zero queries and zero additions independently. The sample's final array is 6, 2, 3, so the requested sums are 11 and 5.

Solution 16: Sum a rectangle

Idea. Build a padded two-dimensional prefix table with the upper and left prefixes minus their overlap. Each query uses four table entries.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0, q = 0;
    std::cin >> rows >> cols >> q;
    std::vector<std::vector<long long>> pref(
        rows + 1, std::vector<long long>(cols + 1, 0));
    for (int r = 1; r <= rows; ++r) {
        for (int c = 1; c <= cols; ++c) {
            long long value = 0;
            std::cin >> value;
            pref[r][c] = value + pref[r - 1][c]
                + pref[r][c - 1] - pref[r - 1][c - 1];
        }
    }
    while (q--) {
        int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
        std::cin >> r1 >> c1 >> r2 >> c2;
        const long long answer = pref[r2][c2]
            - pref[r1 - 1][c2] - pref[r2][c1 - 1]
            + pref[r1 - 1][c1 - 1];
        std::cout << answer << '\n';
    }
}
```

Why it works. The first subtraction excludes cells above the rectangle; the second excludes cells to its left. Adding the common excluded corner once corrects the double subtraction, leaving every wanted cell exactly once.

Complexity. $O(RC + q)$ time and $O(RC)$ storage. Each rectangle query takes $O(1)$.

Check the boundaries. Test one cell, one row, one column, and the full grid. The sample's first rectangle sums $-2 + 3 + 5 - 6 = 0$.

Solution 17: Count marked cells in a rectangle

Idea. Build the two-dimensional prefix table from the condition `row[c - 1] == '#'`. Apply the four-corner query formula.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0, q = 0;
    std::cin >> rows >> cols >> q;
    std::vector<std::vector<int>> pref(
        rows + 1, std::vector<int>(cols + 1, 0));
    for (int r = 1; r <= rows; ++r) {
        std::string row;
        std::cin >> row;
        for (int c = 1; c <= cols; ++c) {
            const int marked = (row[c - 1] == '#');
            pref[r][c] = marked + pref[r - 1][c]
                + pref[r][c - 1] - pref[r - 1][c - 1];
        }
    }
    while (q--) {
        int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
        std::cin >> r1 >> c1 >> r2 >> c2;
        std::cout << pref[r2][c2] - pref[r1 - 1][c2]
            - pref[r2][c1 - 1] + pref[r1 - 1][c1 - 1]
            << '\n';
    }
}
```

Why it works. A sum of the 0/1 cell indicators counts each marked cell once. The rectangle formula selects exactly the requested indicators.

Complexity. $O(RC + q)$ time and $O(RC)$ storage. Counts are at most 250000 under the stated bounds.

Check the boundaries. Test no marks, all marks, and a rectangle on the border. The sample has four marked cells in the full grid, but only one in column 2.

Solution 18: Count one letter inside a substring

Idea. Copy the preceding 26-count row and increment the current letter. A query subtracts two entries in the column for its letter.

Complete solution | C++23

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::string text;
    int q = 0;
    std::cin >> text >> q;
    const int n = static_cast<int>(text.size());
    std::vector<std::vector<int>>> pref(
        n + 1, std::vector<int>(26, 0));
    for (int i = 1; i <= n; ++i) {
        pref[i] = pref[i - 1];
        ++pref[i][text[i - 1] - 'a'];
    }
    while (q--) {
        int l = 0, r = 0;
        char c = 'a';
        std::cin >> l >> r >> c;
        const int id = c - 'a';
        std::cout << pref[r][id] - pref[l - 1][id] << '\n';
    }
}
```

Why it works. Row i counts exactly the first i characters for every letter. Removing row $l - 1$'s contribution from row r leaves the requested substring's count.

Complexity. $O(26n + q)$ time and $O(26n)$ storage. Query time is $O(1)$ for one letter.

Check the boundaries. Check that string indices are shifted by one. Test absent letters and single-character ranges. The word **banana** contains three as.

Solution 19: How many slots meet the fixed threshold?

Idea. Record interval endpoints in a difference array. During reconstruction, build a prefix that adds one exactly when coverage reaches the threshold. Query that final prefix.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, intervals = 0, threshold = 0, q = 0;
    std::cin >> n >> intervals >> threshold >> q;
    std::vector<int> diff(n + 2, 0);
    for (int j = 0; j < intervals; ++j) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        ++diff[l];
        --diff[r + 1];
    }
    std::vector<int> good(n + 1, 0);
    int active = 0;
    for (int i = 1; i <= n; ++i) {
        active += diff[i];
        good[i] = good[i - 1] + (active >= threshold);
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        std::cout << good[r] - good[l - 1] << '\n';
    }
}
```

Why it works. The running coverage counts active intervals. Its threshold indicator is one exactly at qualifying slots. Subtracting indicator prefixes therefore counts slots, not the total number of interval incidences.

Complexity. $O(n + m + q)$ time and $O(n)$ storage.

Check the boundaries. Test a threshold greater than every coverage count and queries containing uncovered gaps. In the sample, slot 3 is the only uncovered slot.

Solution 20: Optional: count all target-sum subarrays

Idea. Start a map with one empty prefix of total zero. For each new prefix, add the frequency of its required earlier total, then record the current total.

Complete solution | C++23

```
#include <iostream>
#include <map>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0;
    long long target = 0;
    std::cin >> n >> target;
    std::map<long long, long long> seen;
    seen[0] = 1;
    long long prefix = 0, answer = 0;
    for (int i = 0; i < n; ++i) {
        long long value = 0;
        std::cin >> value;
        prefix += value;
        const auto it = seen.find(prefix - target);
        if (it != seen.end()) answer += it->second;
        ++seen[prefix];
    }
    std::cout << answer << '\n';
}
```

Why it works. Earlier prefixes identify exactly the left boundaries whose subarray ending here has sum K . Each subarray has one right endpoint, so counting at that endpoint counts it once; lookup before insertion excludes empty ranges.

Complexity. $O(n \log(n + 1))$ time apart from constant setup, and $O(n)$ worst-case storage. Use `long long` for totals and the answer.

Check the boundaries. For all zeros, the answer is $n(n + 1)/2$. The sample has four zero prefix boundaries, yielding six zero-sum subarrays.

Solution 21: Optional: weights start at one inside each query

Idea. Build ordinary prefixes P and global weighted prefixes W . Answer with $W[r] - W[l - 1] - (l - 1)(P[r] - P[l - 1])$.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n = 0, q = 0;
    std::cin >> n >> q;
    std::vector<long long> pref(n + 1, 0), weighted(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value = 0;
        std::cin >> value;
        pref[i] = pref[i - 1] + value;
        weighted[i] = weighted[i - 1] + i * value;
    }
    while (q--) {
        int l = 0, r = 0;
        std::cin >> l >> r;
        const long long sum = pref[r] - pref[l - 1];
        const long long weightedSum = weighted[r] - weighted[l - 1];
        std::cout << weightedSum - (l - 1LL) * sum << '\n';
    }
}
```

Why it works. Subtracting $l - 1$ times the ordinary range sum reduces every global weight i to the required local weight $i - l + 1$.

Complexity. $O(n + q)$ time and $O(n)$ storage. Keep all weighted arithmetic in `long long`.

Check the boundaries. A one-element query has weight 1. The sample's $[2, 4]$ gives $3 + 2 \cdot 1 + 3 \cdot 4 = 17$.

Solution 22: Optional: batch additions on a zero grid

Idea. For each addition, record signs $+$, $-$, $-$, $+$ at its four start/stop corners. Reconstruct each real cell from the upper, left, and upper-left reconstructed entries.

Complete solution | C++23

```
#include <iostream>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int rows = 0, cols = 0, updates = 0;
    std::cin >> rows >> cols >> updates;
    std::vector<std::vector<long long>> diff(
        rows + 2, std::vector<long long>(cols + 2, 0));
    for (int j = 0; j < updates; ++j) {
        int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
        long long delta = 0;
        std::cin >> r1 >> c1 >> r2 >> c2 >> delta;
        diff[r1][c1] += delta;
        diff[r2 + 1][c1] -= delta;
        diff[r1][c2 + 1] -= delta;
        diff[r2 + 1][c2 + 1] += delta;
    }
    for (int r = 1; r <= rows; ++r) {
        for (int c = 1; c <= cols; ++c) {
            diff[r][c] += diff[r - 1][c] + diff[r][c - 1]
                - diff[r - 1][c - 1];
            if (c > 1) std::cout << ' ';
            std::cout << diff[r][c];
        }
        std::cout << '\n';
    }
}
```

Why it works. After reconstruction, one update contributes its amount exactly where both coordinate ranges include the cell. Summing these independent contributions gives the final grid for all updates.

Complexity. $O(RC + u)$ time and $O(RC)$ storage, with sentinel rows and columns allocated.

Check the boundaries. Test a full-grid addition, negative rectangles, and a one-cell grid. The sample's second update adds 3 to both cells of row 2.

6.37 Explained trace and complexity drills

Check both the result and the reasoning. When your answer differs, find the first entry or boundary at which your interpretation changed.

Drill 1: A prefix is allowed to decrease

Start with $P[0] = 0$. Successive additions give

$$P = [0, -3, 2, 2, 0].$$

The requested interval contains 5, 0, -2. Its answer is

$$P[4] - P[1] = 0 - (-3) = 3.$$

Subtracting a negative prefix is valid: it cancels the unwanted contribution just as subtracting a positive prefix does.

The prefix array is not sorted here. Adding -3 decreases the first total; adding -2 decreases the last one. Prefix-sum subtraction works for signed values without a monotonicity requirement. A later algorithm that uses sortedness or binary search would need that property separately. Nonnegative inputs make successive prefix totals nondecreasing, but arbitrary signed inputs do not.

Drill 2: Translate endpoints before calculating

The included indices are 1 and 2, holding 5 and -1. They are positions 2 and 3 in one-based numbering, so the equivalent inclusive interval is $[2, 3]$.

The prefix array is $[0, 2, 7, 6, 10]$. A half-open zero-based interval uses

$$P[R] - P[L] = P[3] - P[1] = 6 - 2 = 4.$$

The second proposed formula also includes the initial 2. The third also includes the final 4. Both errors come from changing a boundary's meaning, not from arithmetic.

The interval $[2, 2)$ contains no elements. Its sum is $P[2] - P[2] = 0$. In contrast, a one-based inclusive interval $[2, 2]$ contains one element. Always read whether the right endpoint is included.

Drill 3: Position ranges and value ranges

The frequency array is

$$\text{freq} = [1, 1, 0, 0, 2].$$

Value 0 appears once, value 1 once, and value 4 twice. Frequencies sum to 4, the number of observations.

The first two original positions contain 4 and 1, giving a position-range sum of 5. The value interval $[1, 2]$, however, contains only the single observed 1, so its count is 1. These queries use different axes even though their written endpoints are identical.

The frequency array cannot recover the position of the first 4. For example, 0, 4, 1, 4 has exactly the same frequencies but a different first occurrence. Counting by value deliberately discards original order. Use the original sequence or separate positional information when order matters.

Drill 4: An offset is a reversible mapping

There are $2 - (-3) + 1 = 6$ possible values, so valid indices are 0 through 5. The value -1 maps to

$$-1 - (-3) = 2.$$

Index 5 maps back to $5 + (-3) = 2$, the highest allowed value.

Every distinct value in this domain receives a distinct valid index. In contrast, absolute value merges -1 and 1, and also merges -2 and 2. Their counts would be combined even though they are different values.

An offset removes a small negative starting point, but it does not shrink the domain's width. A domain spanning billions of integers still needs a different representation, such as a map or compression, rather than a huge direct-access vector.

Drill 5: Trace both boundaries of every update

The first update adds 2 at $D[1]$ and subtracts 2 at $D[4]$. The second subtracts 1 at $D[3]$ and adds 1 at $D[6]$. Combining markers gives

$$D[1 \dots 6] = [2, 0, -1, -2, 0, 1].$$

Running totals through the real positions give

$$a = [2, 2, 1, -1, -1].$$

At position 3 both updates are active, so the result is $2 - 1 = 1$. At positions 4 and 5 only the negative update remains.

Position 6 is a boundary just after the last real element, not a sixth element. Its marker cancels the remaining -1 , returning the running change to zero. Storing it lets the update use the same formula even when $r = n$.

Drill 6: Do not add the base array twice

The direct answer is $[2, 6, 9]$. Full differences of the original array have real entries $[2, 0, 3]$. Adding the update marker at position 2 changes them to $[2, 4, 3]$; a cancellation is stored after position 3. Their running totals are already $[2, 6, 9]$.

With an update-only difference array, the reconstructed changes are $[0, 4, 4]$. Adding those changes to the original values gives the same correct result.

The incorrect hybrid adds the base to values that already contain it:

$$[2, 6, 9] + [2, 2, 5] = [4, 8, 14].$$

Both valid approaches work, but they give the difference array different meanings. Write that meaning beside its declaration. Reusing the formula without preserving its meaning is the source of the double-counting error.

Drill 7: When an old prefix stops describing the data

The old formula returns $P[3] - P[1] = 6 - 1 = 5$. The new array is $[1, 7, 3]$, so the correct sum is 10. Entries $P[2]$ and $P[3]$ are stale; their corrected values would be 8 and 11. Entries $P[0]$ and $P[1]$ do not include the changed element and remain correct.

A final reconstruction after all updates describes the final state, not each intermediate state. It cannot simply replace a query that must see only earlier updates. The batch method in this chapter requires all its updates to precede the queries about their combined result. Interleaved update/query problems need additional reasoning or other data structures. Static prefix sums do not silently become a dynamic-query solution.

Drill 8: Count initialization and hidden copies

A initializes $U + 1 = 1000001$ counters on each test case. Across 100 cases, that alone means 100000100 initialized entries. Its total work is $O(T(U + n + q + 1))$, not merely $O(T(n + q))$. A large mostly unused value domain can dominate the workload.

B copies $n + 1$ stored values before each helper body runs. Its q queries therefore cost $O(nq)$ overall for nonempty arrays, rather than $O(q)$. The one subtraction does not remove the parameter-construction cost.

A read-only helper should normally receive `const std::vector<long long>& pref`. The reference avoids a vector copy and `const` prevents ordinary modification through that parameter. The caller must keep the vector alive throughout the call.

Drill 9: A count of qualifying positions is not a coverage sum

The qualification array is $[1, 0, 1, 0]$. Its prefix array is $[0, 1, 1, 2, 2]$, so the requested answer is

$$P[4] - P[1] = 2 - 1 = 1.$$

Only position 3 qualifies within that interval.

A prefix of raw coverage would instead return $0 + 3 + 1 = 4$, which counts coverage contributions, not qualifying positions. Neither result is an arithmetic mistake; they answer different questions.

The qualification prefix was built for threshold 2. Changing the threshold changes the indicator values, so the same stored summary is not generally valid for threshold 3. For example, querying the whole array gives two qualifying positions at threshold 2 but only one at threshold 3. The core method assumes one fixed threshold during preprocessing.

Drill 10: A rectangle needs four boundaries

Including column 0, the nonzero prefix rows are

$$P[1] = [0, 1, 3, 6], \quad P[2] = [0, 5, 12, 21].$$

For $(r_1, c_1, r_2, c_2) = (1, 2, 2, 3)$, the formula gives

$$P[2][3] - P[0][3] - P[2][1] + P[0][1] = 21 - 0 - 5 + 0 = 16.$$

The included values are 2, 3, 5, and 6, also summing to 16.

The padded row and column represent empty regions, so their totals are zero. This allows exactly the same formula when a query begins in row 1 or column 1. Without padding, those boundaries would need explicit special cases to avoid a negative index. Padding simplifies the code; it does not change the region being summed.

Drill 11: A wide destination does not widen an earlier calculation

`std::partial_sum` accumulates using the input value type, here `int`. The third total is 3000000000, outside a 32-bit signed `int`. Writing the result into a `long long` destination occurs too late to prevent that intermediate signed overflow.

Use `std::vector<long long> a` as the input, or build the prefix explicitly with a `long long` accumulator. A wider destination alone is not the repair. This is also why the manual examples read values into an appropriately wide type before arithmetic.

For equal pairs, the answer is

$$\frac{100000 \cdot 99999}{2} = 4999950000.$$

It needs more than a 32-bit signed integer. If `n` is an `int`, write `1LL * n * (n - 1) / 2` so the multiplication is already wide. The stated bound makes that expression safe in the tested 64-bit `long long` environment.

Drill 12: Optional: a prefix must look only into the past

Initially the zero-prefix count is 1. At the first input, the lookup contributes 1 and the count then becomes 2. At the second, the lookup contributes 2 and the count becomes 3. The answer is $1 + 2 = 3$: the first singleton, the second singleton, and the whole array.

Incrementing first makes the contributions 2 and 3, producing 5. The extra contribution on each step pairs the current boundary with itself and therefore counts an empty subarray that the task forbids.

Without the initial empty prefix, the correct lookup-before-insert order produces contributions 0 and 1, giving only 1. Both subarrays that begin at position 1 are lost. The initial `seen[0] = 1` accounts for the boundary immediately before the first element; it does not count an empty output subarray.

6.38 Quick reference, glossary, and readiness checklist

This is a revision aid, not a replacement for the worked explanations. Before choosing a formula, write what one stored entry means and which coordinate system the query uses.

One-dimensional formulas

Let $P[k]$ be the sum of the first k values, with $P[0] = 0$. The sequence has n values; the prefix has $n + 1$ entries.

Task	Rule
Build from one-based values	$P[i] = P[i - 1] + a[i]$.
Sum, one-based inclusive $[l, r]$	$P[r] - P[l - 1]$.
Sum, zero-based inclusive $[L, R]$	$P[R + 1] - P[L]$.
Sum, zero-based half-open $[L, R)$	$P[R] - P[L]$.
Recover one value	$a[i] = P[i] - P[i - 1]$.
Count a fixed property on $[l, r]$	Prefix the 0/1 property values, then subtract.
Sum strictly left of position i	$P[i - 1]$.
Sum strictly right of position i	$P[n] - P[i]$.

Count by value, not by original position

For an integer domain $[0, U]$, allocate $U + 1$ counters and add one to `freq[x]` for every observed value x . To count observed values in a value interval, define

$$C[0] = 0, \quad C[v + 1] = C[v] + \text{freq}[v].$$

Then a value interval $[L, R]$ contains $C[R + 1] - C[L]$ observations. The shift by one leaves room for value 0.

For a small signed domain $[\text{LOW}, \text{HIGH}]$, use index $x - \text{LOW}$, with $\text{HIGH} - \text{LOW} + 1$ entries. An offset does not make a wide domain small.

Add on a range, then reconstruct

For one-based inclusive updates on an initially zero sequence, allocate `diff(n + 2, 0)`. For an addition of v on $[l, r]$, record

$$D[l] += v, \quad D[r + 1] -= v.$$

A running sum through positions 1 to n gives the final values. If D stores *only updates*, add those reconstructed changes to the base array. If D was initialized as the base array's *full differences*, reconstruction already includes the base.

Grid formulas and the four-corner pattern

For one-based matrix positions, allocate an extra zero row and column. Define $P[r][c]$ to be the total of rows 1 through r and columns 1 through c .

$$P[r][c] = a[r][c] + P[r-1][c] + P[r][c-1] - P[r-1][c-1].$$

For an inclusive rectangle (r_1, c_1) to (r_2, c_2) , use

$$\begin{aligned} \text{answer} &= P[r_2][c_2] - P[r_1-1][c_2] \\ &\quad - P[r_2][c_1-1] + P[r_1-1][c_1-1]. \end{aligned}$$

The shared corner was removed twice, so add it back once. For optional rectangle additions, the four difference markers are $+v, -v, -v, +v$ at (r_1, c_1) , (r_2+1, c_1) , (r_1, c_2+1) , and (r_2+1, c_2+1) . Allocate both dimensions with two extra entries for those outside boundaries.

Total work: include building and initialization

Here q is the number of queries, u the number of updates, n the sequence length, $U+1$ the value-domain size, and R, C the grid dimensions. Numeric operations are treated as constant cost. Storage includes the main summaries, not just loop counters.

Method	Total time	Summary storage
Scan every interval	$O(n + nq)$ worst case	None beyond stored input.
Static prefix sums	$O(n + q)$	$O(n)$.
Direct frequency lookup	$O(U + n + q + 1)$	$O(U + 1)$.
Value-range count prefix	$O(U + n + q + 1)$	$O(U + 1)$.
26 letter-prefix columns	$O(26n + q)$	$O(26n)$.
Batch range additions	$O(n + u)$	$O(n)$.
Batch updates, then sums	$O(n + u + q)$	$O(n)$.
Static rectangle sums	$O(RC + q)$	$O(RC)$.
Batch rectangle additions	$O(RC + u)$	$O(RC)$.

For a zero-length input or an empty domain of observations, a constant amount of setup still happens. Terms such as $O(n+1)$ make that convention explicit. A frequency domain containing just value 0 still has one counter.

A method is useful only for the promised workload

Prefix sums do not directly support interleaved changes and queries. Difference arrays do not instantly reveal every updated value before reconstruction. Frequency arrays require a manageable value domain. Choose the tool whose stored information actually answers the question.

Glossary: the words behind the formulas

Range query

A question about a consecutive interval or rectangular region.

Static data

Data that does not change while the relevant queries are answered.

Preprocessing

Work done once to prepare information for later queries.

Prefix

An initial part of a sequence, including the possible empty prefix.

Prefix sum

A stored cumulative total for each possible prefix length.

Suffix

A final part of a sequence, ending at its last element.

Inclusive interval

An interval that includes both stated endpoints, such as $[l, r]$.

Half-open interval

An interval that includes its left endpoint but excludes its right, such as $[L, R)$.

Padding or sentinel entry

A deliberately allocated boundary entry used to simplify formulas. It is not automatically a real input value.

Frequency

The number of occurrences of one value or category.

Value domain

The set or interval of values that an input is allowed to contain.

Offset mapping

Translating a bounded value x to index $x - \text{LOW}$.

Predicate and indicator

A predicate is a true/false condition; its indicator is 1 when true and 0 when false.

Difference array

A representation of changes between successive values, or of the changes that pending updates introduce at boundaries.

Reconstruction

Recovering the represented values by accumulating differences.

Glossary continued

Batch updates

A collection of changes processed together before answering questions about their combined result.

Coverage

How many supplied intervals or rectangles include a particular position or cell.

Overlap correction

Adding back or subtracting a region that earlier terms counted the wrong number of times.

Coordinate compression

Replacing distinct values by their sorted ranks. It preserves order, not the original distances between values.

Invariant

A statement kept true after each step, used to explain why an algorithm works.

Boundary case

An allowed input that stresses an endpoint or special state, such as a single element or an update ending at n .

Numeric bound

A justified upper limit on the magnitude of an intermediate result or answer, used when selecting a type.

Differential test

Comparing two implementations, usually a fast method and a simple direct method, on the same inputs.

Readiness checklist

You are ready to leave the core of this chapter when you can do the following without copying a formula blindly:

1. Explain why the empty prefix is zero and why a prefix vector has $n + 1$ entries.
2. Derive $P[r] - P[l - 1]$ by cancelling unwanted positions, including $l = 1$.
3. Tell a position interval from a value interval and choose the matching summary.
4. Count a fixed property by turning it into 0/1 values.
5. Choose a safe frequency domain or explain why a map is more appropriate.
6. Derive both markers of an interval addition, including negative updates and $r = n$.
7. Separate base values from update-only changes without adding the base twice.
8. Explain the four-corner rectangle formula using overlap, rather than only remembering its signs.
9. Justify time, memory, and numeric bounds for the whole input, including all test cases.
10. State when the array may change and why the chosen summary stays valid.

A wrong prediction is a useful study signal. Return to the smallest example for that idea, trace it by hand, and test one deliberate variation. Confidence should come from being able to explain the result, not from having seen the code before.

6.39 Technical notes and references

These notes document the assumptions behind the beginner-facing material. They are for a second reading and for anyone rebuilding or adapting the training files.

Language version and terminology

All 52 complete Session 6 programs use standard headers and target **C++23**. None needs the GNU PBDS extension from Session 5. The language references below point to the fixed N4950 C++23 draft rather than a moving draft that may contain later changes.

A mathematical *partial sum* is a running total. The C++ library algorithm `std::partial_sum` calculates successive totals; it is not a range-update data structure. The training’s range-addition technique is named **difference array** throughout this chapter, with “partial sums” retained in the syllabus/cover wording so the connection is explicit. Reconstructing a difference array uses running sums, which connects the two ideas without making them identical.^{[6-4][6-5]}

Index, input, and character contracts

The programs assume inputs satisfy their stated format and constraints. They are contest solutions, not interactive validation forms. A valid logical endpoint must still correspond to an allocated C++ entry. In particular, **reserve** does not create indexable vector elements; these programs construct vectors with their needed sizes.^[6-2]

Most range statements use one-based inclusive endpoints. Strings still use zero-based character indices, and prefixes use lengths. Value-domain prefixes deliberately use a separate shift so value 0 is representable. A formula copied between those conventions needs a derivation, not just renamed variables.

Text problems explicitly use ASCII letters and, for marked grids, the stated single-byte symbols. Alphabet offsets are not presented as a technique for arbitrary Unicode text. Lengths and dimensions are bounded so they fit the tested **int** environment.

Numeric widths and overflow

The verification environment has a 32-bit **int** and a 64-bit **long long**. C++ does not require every implementation to use those exact widths. The large problem bounds assume at least the tested widths; use appropriate types and inspect the limits when porting to a materially different environment.

For $n \leq 200000$ and $|a_i| \leq 10^9$, an ordinary prefix has magnitude at most $2 \cdot 10^{14}$. Equal-position-pair counts can reach $n(n-1)/2 = 19999900000$. These need a wider type than a 32-bit **int**. A wide destination does not repair an expression that already overflowed; promote before multiplication or accumulate in a wide type. Signed overflow is not a valid wraparound strategy.^[6-3]

Bounds for the combined and optional methods

The batch-update-then-query examples use $|\Delta| \leq 10^6$, $u \leq 200000$, and $|a_i| \leq 10^9$. One final value has magnitude at most $10^9 + 200000 \cdot 10^6 = 2.01 \cdot 10^{11}$. A prefix of up to 200000 such values is bounded by $4.02 \cdot 10^{16}$, safely within the tested signed 64-bit range.

Weighted sums use $|a_i| \leq 10^6$ and $n \leq 200000$. Even the conservative bound $n^2 \cdot 10^6 = 4 \cdot 10^{16}$ fits the tested wide type. Grid examples use at most 500×500 cells. Rectangle-addition reconstruction is also checked against the update count and update magnitude, not just the original cell type.

Modular examples use $1 \leq M \leq 10^9$ and wide arithmetic. C++ remainder for a negative operand is not automatically a normalized value in $[0, M)$. The examples normalize a negative remainder explicitly. A zero modulus is excluded by the input contract.^[6-8]

The library helpers have specific type rules

`std::partial_sum` uses the input iterator's value type for its running accumulator.

`std::adjacent_difference` also derives its temporary value type from the input. The examples therefore supply `long long` inputs when their sums or differences require that width. For `std::accumulate`, the initial value selects the accumulator type; OLL is intentional.^{[6-4][6-5][6-6]}

Sorted ranks and bounds in the compression extension use standard binary-search algorithms. Compression preserves comparisons and equality, but a rank difference is not a physical distance or a count of every original integer between two coordinates.^[6-7]

What the stated complexity includes

Time bounds use constant-cost fixed-width arithmetic. They include initialization, rebuilding, and input/output-sized traversal, rather than just the final subtraction. Storage counts the containers created by the program, with optional comments distinguishing the input from auxiliary storage. Actual vector overhead and compiler/runtime allocations are not claimed to equal the element payload alone.

A 26-column character prefix uses $26(n+1)$ counters. For $n = 100000$ and 4-byte counters, the counter payload alone is 10400104 bytes; nested-container overhead and the stored string are additional. A constraint-dependent representation choice is often more important than replacing a loop with a shorter library call.

What was checked for this edition

Session 6 contains **30 complete teaching programs and 22 complete coding solutions**. All 52 compiled with **GCC 14.2.0** in C++23 mode and passed **8118 input/output checks**: 52 printed-sample checks, 8045 generated or exhaustive-small-case checks, and 21 additional large-boundary checks.

Expected outputs are produced independently in Python by direct summation, direct interval/cell updates, character counting, and small exhaustive subarray enumeration. The tests compare complete outputs, including tie-breaking and formatting. They exercise empty inputs where allowed, one-element inputs, negative values, all-equal values, missing frequencies, both interval endpoints, overlapping and cancelling updates, single-cell and full-grid rectangles, repeated cases, large counts, and wide intermediate totals.

The verification build uses:

```
-std=c++23 -pedantic-errors -Wall -Wextra  
-Wconversion -Wshadow -Werror -O1 -g  
-D_GLIBCXX_ASSERTIONS -D_GLIBCXX_DEBUG  
-fsanitize=undefined -fno-sanitize-recover=all
```

Warnings, library checking, and undefined-behavior instrumentation can expose faults on executed paths; they do not prove correctness for every possible input. The explanation of the invariant or cancellation is still necessary. Deliberately wrong fragments in the text are labelled and are not enrolled as working programs.^{[6-9][6-10][6-11]}

Reproduce the checks or build the PDFs

From the project root, run `python3 tests/verify_session_06.py`. The script uses a C++23-capable GNU compiler, compiles one source file at a time, and writes a JSON report with per-program checks and source hashes. Build `session_06.tex` for the handout, or `book.tex` for the complete six-session edition. The README contains exact commands and environment notes.^[6-12]

Each complete listing is imported from its distributed `.cpp` file. The standalone handout and collected book import the same chapter. Earlier verification suites were also rerun successfully: across all six sessions, **321 complete programs passed 35259 checks**. Session 5's nine GNU PBDS programs additionally need the documented extension headers.

How to read the reference markers

Markers such as ^[6-4] link to the entries below. They support language/library rules and identify background reading; the examples, tasks, traces, proofs, and test cases were prepared for this training. The first number keeps this session's references distinct in the collected book.

Algorithm and language references

[6-1] Antti Laaksonen: Competitive Programmer's Handbook, Chapter 9, range queries

<https://cses.fi/book/book.pdf>

Background reading: prefix sums and difference arrays. The book uses its own indexing conventions; translate them before comparing formulas.

[6-2] C++23 N4950: vector construction

<https://timsong-cpp.github.io/cppwp/n4950/vector.cons>

[6-3] C++23 N4950: expression results and representable ranges

<https://timsong-cpp.github.io/cppwp/n4950/expr.pre>

[6-4] C++23 N4950: partial sums

<https://timsong-cpp.github.io/cppwp/n4950/partial.sum>

[6-5] C++23 N4950: adjacent differences

<https://timsong-cpp.github.io/cppwp/n4950/adjacent.difference>

[6-6] C++23 N4950: accumulate

<https://timsong-cpp.github.io/cppwp/n4950/accumulate>

[6-7] C++23 N4950: binary search algorithms

<https://timsong-cpp.github.io/cppwp/n4950/alg.binary.search>

[6-8] C++23 N4950: multiplication, division, and remainder

<https://timsong-cpp.github.io/cppwp/n4950/expr.mul>

Compiler and verification references

[6-9] GCC: warning options

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

[6-10] GCC: instrumentation and sanitizer options

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

[6-11] GNU libstdc++: using debug mode

https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html

[6-12] GCC: selecting the C++ language standard

<https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

6.40 Completing Level 0: turn the tools into a method

You have now reached the end of the six-session syllabus. That does not mean every technique should feel automatic yet. The next step is to solve unfamiliar small problems without being told which chapter they belong to.

One workflow that uses the whole book

1. **Understand the contract.** Identify the input, the exact output, valid bounds, and whether order or duplicates matter. This is the problem-reading habit from Session 1.
2. **Choose what must be remembered.** Decide whether the task needs individual positions, text, a waiting order, unfinished actions, keys, counts, or boundaries. Use Sessions 2, 4, and 5 to choose a representation.
3. **State one clear meaning for each helper and array.** Separate jobs into functions when that improves clarity. Distinguish copies from references, using Session 3.
4. **Write a simple correct method first.** Trace three or four values on paper. When work repeats, ask whether a summary can remove the repetition, as in this session.
5. **Check the limits before implementing.** Count total work and simultaneous storage. Bound totals and products, not only one input value.
6. **Test intentionally.** Include a smallest case, an endpoint, a tie, and a case that would break your first guess. Compare a fast method against a direct one on small generated inputs when practical.
7. **Explain the result.** Say why the state or formula remains correct after each step. Then explain why the whole program fits the constraints.

A useful independent revision exercise

Choose one solved problem from each session. Hide both the solution and the chapter title. Write your own plan, source code, boundary tests, and a short explanation of the time and memory cost. Compare your reasoning with the book only after a genuine attempt.

When you cannot finish, identify the missing step precisely: reading the statement, choosing a representation, deriving an update, handling a boundary, or analysing cost. Revisit that idea rather than restarting the whole book. Understanding a small mistake deeply is more useful than copying many complete solutions.

The goal is independent reasoning

You do not need to remember every member function or every optional extension. You should be able to choose a suitable tool, look up its exact operation when needed, and explain why the resulting algorithm solves the stated problem.