

Level 1 Training

Session 1

Two Pointers and Binary Search

Ordered scans, sliding windows,
and monotone search

Understand the boundary. Choose the method.
Build a solution you can explain.

How to study this session

You can already store a sequence, write a loop, and test a condition. Now the challenge is to avoid checking the same possibilities repeatedly. This session teaches **Two Pointers and Binary Search together**: first as different ways to use order, then as tools that can cooperate inside one solution.

The route through the ideas

Begin with a pair of values whose sum must match a target. Learn how two ends can move safely, then how a midpoint can rule out half an interval. Compare both solutions to the *same* pair problem. Next, process many queries, maintain a moving window, and connect windows to binary searches on prefix sums. Finally, search a possible answer and build its checking function from a scan.

This is one chapter with one sequence of sections, examples, practice problems, hints, and solutions. Sections are numbered **1.1, 1.2, and so on**. There is one revision section at the end.

A routine to use on every worked problem

1. **Try a direct solution first.** Name the candidates that a correct slow approach would inspect.
2. **Write the meaning of the state.** An index might be a possible partner, the next unread item, or the edge of a window. Those meanings are not interchangeable.
3. **Explain one update.** State what becomes impossible or unnecessary when an index changes. Then show that the process terminates.
4. **Check an awkward case.** Use duplicates, zeros, a missing answer, or the smallest allowed input. Write the expected result before running the program.

Code and prerequisites

The complete programs use **C++23** and standard headers. Each is supplied as a separate `.cpp` file with its own `main`. Compile one file at a time. Short fragments are labelled with their intended location; deliberately wrong fragments are never templates to copy.

You need Level 0 loops, functions, references, arrays, strings, vectors, pairs, sorting, prefix sums, and basic Big-O notation. Recursion, masks, number theory, and advanced data structures are not prerequisites.

Read in stages

Cover each **Pause and predict** answer before reading it. Attempt the mixed practice before consulting hints. The guided pair-distance problem brings both techniques together; the optional extensions and Practice 18 to 20 can wait until the core material is comfortable.

The Level 1 learning path

Session	Topics
1	Two Pointers and Binary Search. Ordered scans, sliding windows, sorted lookup, monotone boundaries, and search on the answer.
2	Recursion and Backtracking. Calls, decision trees, enumeration, pruning, and state restoration.
3	Bit Manipulation and Bitmasking. Binary values, bitwise operations, XOR, compact sets, and subsets.
4	Counting Principles. Addition, multiplication, nCr , nPr , and inclusion-exclusion.
5	Number Theory, Part 1. Divisors, prime factorization, and the sieve.
6	Number Theory, Part 2. Modular arithmetic, binary power (exponentiation), modular inverse, GCD, and LCM.

Session 1 of the six-session series

This handout contains **Two Pointers and Binary Search**. The complete collected book contains all six sessions. Its Session 1 is the same chapter as this handout, with matching section, example, and practice numbers.

What you should be able to explain afterwards

Why a pointer can move without returning. Why binary search keeps the correct boundary. When a sum window needs nonnegative values. How to prove a feasibility test. When two pointers are faster than repeated binary searches, and when binary search can use a two-pointer scan as its checker.

Conventions used throughout

Array indices in code are **zero-based**. A task may ask you to print a one-based position; that conversion is stated explicitly. A subarray or substring is contiguous. A subsequence may skip elements but preserves their relative order.

Programs assume valid input within the stated bounds and print no prompts. Empty input sequences are permitted only where explicitly stated. The text exercises use ASCII characters. Ordinary numeric examples use `long long` for values, totals, products, and counts; indices fit in `int` under the chapter's size limits.

Complexity counts fixed-width arithmetic operations. A stated scan cost excludes sorting unless the text says otherwise. When input may be empty, expressions such as $O(n)$ are conventional growth descriptions for positive n ; handling an empty input takes constant work.

Contents

How to study this session	i
The Level 1 learning path	ii
1 Two Pointers and Binary Search	1
1.1 Start with the candidates, not with a template	2
1.2 Give every index a precise meaning	3
1.3 Use two ends to match or eliminate candidates	4
1.4 Halve an ordered interval to find a boundary	6
1.5 Turn boundaries into exact and range queries	8
1.6 Solve the same pair problem with either method	10
1.7 Count whole groups of sorted pairs	12
1.8 Merge two sorted sequences without restarting	13
1.9 Answer many queries with searches or a sweep	15
1.10 Use forward positions without changing the order	17
1.11 Slide a fixed-length window by changing two contributions	19
1.12 Keep the longest nonnegative window within a budget	20
1.13 Find a shortest window by measuring before removal	22
1.14 Count every valid window instead of keeping one length	24
1.15 Maintain frequencies when a sum is not the right state	25
1.16 Connect prefix searches to a moving window	26
1.17 Recognize when a tempting shortcut is invalid	28
1.18 Binary-search a possible answer instead of an index	29
1.19 Find the last feasible answer and protect arithmetic	30
1.20 Find minimum production time with a monotone count	32
1.21 Maximize a cut height by reversing the truth direction	34
1.22 Partition a sequence: prove both the check and the search	36
1.23 Maximize a minimum gap with an earliest-choice check	38
1.24 Combine the techniques: the k th smallest pair distance	40
1.25 Optional extension: exact sums from two at-most counts	42
1.26 Optional extension: bisect real-valued answers	43
1.27 Choose a method, then test its entire argument	45
1.28 Mixed practice: decide which boundary the task needs	46
1.29 Mixed drills: trace, justify, and compare	56
1.30 Hints before solutions	62
1.31 Explained practice solutions	67
1.32 Explained mixed drills	87
1.33 One quick reference for both techniques	93
1.34 Technical notes, verification, and sources	97

SESSION 1

Two Pointers and Binary Search

Use order to avoid repeated work. Keep the answer possible.

Imagine a sorted list of prices. You want two different items with a particular total. A direct program checks every pair. A better program asks what sorted order tells you: if even the largest partner is too small, can the current smallest value ever work?

That question leads to two pointers. A related question leads to binary search: if the middle value is below the value you need, which half can no longer contain it? Both techniques replace repeated inspection with a justified change to a boundary.

The similarity matters, but the techniques are not identical. Two pointers usually reuse progress across neighbouring candidates. Binary search usually narrows one ordered search space by halves. Some problems allow either; some allow only one; some use both.

The questions that organize this chapter

Question	Useful state	Typical tool
Which pair can work?	Two remaining ends, or one fixed value and a searched partner.	Two pointers or binary search.
How many values are below a threshold?	A boundary in sorted data.	Binary search; a sweep for ordered queries.
Which contiguous segment is valid?	A window and its current sum or frequencies.	Sliding window; sometimes prefix search.
What is the smallest feasible answer?	An answer interval and a yes/no checker.	Binary search, possibly with a scan inside.

A first reading and a second reading

On your first reading, understand the pointer meanings, the lower-bound search, ordinary windows, and the first-feasible answer search. Use the worked comparisons to connect them. Then attempt several mixed practice problems before reading their solutions.

On your second reading, revisit the counting arguments, greedy checking proofs, the combined pair-distance example, and optional extensions. The goal is to select and justify an algorithm, not to collect many templates with similar variable names.

Your starting sentence

“My remaining candidates are . . . , and the input guarantees Therefore this update cannot discard a required answer because” Write that sentence before the loop.

1.1 Start with the candidates, not with a template

One problem to carry through the session

A sorted price list is

$$a = [2, 4, 4, 7, 9, 12].$$

Can two *different positions* have total 13? The answer is yes: $4 + 9 = 13$. The two entries containing 4 are separate items. Their equal values do not make their positions identical.

The direct solution checks every $i < j$. It is useful because its meaning is obvious: every unordered pair of distinct positions appears exactly once.

Fragment inside a helper; the input values and target are long long

```
for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
        if (a[i] + a[j] == target) return true;
    }
}
return false;
```

There are $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ candidate pairs. At $n = 200000$, that is 19,999,900,000 pairs. This is a count of candidates, not a prediction of an exact runtime.

Two different improvements

You could fix one value and **binary-search** for its missing partner. Alternatively, you could place **two pointers** at the ends and eliminate an entire group of impossible pairs with each move. We will derive both, compare their cost, and reuse their boundary ideas in later problems.

Neither improvement is justified just because there are two variables or a midpoint. The improvement comes from a fact about the candidates. Here that fact is sorted order.

Three promises that must not be confused

Sorted values let comparisons rule out earlier or later values. **Nonnegative values** make a segment sum grow or stay unchanged when you append an element. A **monotone predicate** has one direction of change across its candidate answers. Later, we will use each promise for a different reason.

Pause and predict: 1

For $a = \{6\}$ and target 12, should the pair problem return YES? What about $a = \{6, 6\}$?

Check your prediction. The first answer is NO: a position cannot pair with itself. The second is YES because the two equal values occupy different positions.

1.2 Give every index a precise meaning

In this chapter, a pointer is normally an **integer position**, not a raw C++ memory pointer. Draw the sequence and place each index above the item or boundary it describes.

Elements and boundaries are different

For n elements, the valid element indices are $0, 1, \dots, n - 1$. The boundary n means “after the final element.” It is useful as a search result, but it is not a valid element to read.

Index	0	1	2	3	4	5	6
Value	2	4	4	7	9	12	<i>end</i>

A **closed** interval $[l, r]$ includes both endpoints and has length $r - l + 1$ when nonempty. A **half-open** interval $[l, r)$ includes l but excludes r , and has length $r - l$. The half-open interval $[0, n)$ is the whole array; $[3, 3)$ is empty.

We use closed endpoints for opposite-end scans and inclusive windows. We use a half-open unknown interval for array lower bound. For answer search we use a closed interval of candidate *values*. Every transition will restate which interval is intended.

An invariant is a promise that survives an update

A loop invariant is a statement that is true before the first iteration and remains true after every iteration. For a window it might be “sum equals the total of the current elements.” For a search it might be “the answer is still inside these boundaries.”

To justify a loop, explain its **initial state**, why each update **preserves the promise**, and what the promise means when the loop **finishes**. Also identify a quantity that makes progress: remaining width, number of unread elements, or distance to the end.

Signed indices avoid one beginner trap

```
int n = static_cast<int>(a.size()); // n <= 200000 here
int right = n - 1;                // becomes -1 when n is zero
```

This conversion is justified by the stated bound. In contrast, `a.size() - 1` is unsigned arithmetic and is not a safe way to construct a negative empty-array index. Always test a boundary before indexing.

Pause and predict: 2

For $n = 6$, what does the lower-bound result 6 mean? May the program read `a[6]`?

Check your prediction. It means the requested boundary is after all six values. It does not identify an element. Reading `a[6]` would be outside the vector’s element range.

1.3 Use two ends to match or eliminate candidates

Matching symmetric positions

A palindrome reads the same forwards and backwards. Compare the two outer characters. A mismatch proves failure; a match lets you move inward because that pair no longer needs inspection. For odd lengths, the central character matches itself automatically.

Contract for Example 01. Read one ASCII line of length 0 to 200000, including spaces. Compare characters exactly, with case and spaces significant. Print YES for a palindrome, otherwise NO. The empty line is a palindrome.

Example 01: A palindrome check | Complete C++23 program

```
#include <iostream>
#include <string>
int main() {
    std::string text;
    std::getline(std::cin, text);
    int left = 0;
    int right = static_cast<int>(text.size()) - 1;
    bool matches = true;
    while (left < right) {
        if (text[left] != text[right]) {
            matches = false;
            break;
        }
        ++left;
        --right;
    }
    std::cout << (matches ? "YES" : "NO") << '\n';
}
```

Sample input

level

Sample output

YES

Before each comparison, every pair outside $[left, right]$ has already matched. If the indices meet or cross without a mismatch, all required pairs have matched. Each successful iteration removes two positions, so the scan takes $O(n)$ time and $O(1)$ working memory beyond the input string.

Pause and predict: 3

Does a palindrome check require the characters to be sorted?

Check your prediction. No. It uses symmetry in the original order. Sorting would change the word and therefore change the question.

Eliminating pairs in a sorted array

For the target-sum problem, start with `left = 0` and `right = n - 1`. Compare the sum of the two current values with the target.

If $a[left] + a[right]$ is too small, every allowed partner $j \leq right$ for this left value is also too small:

$$a[left] + a[j] \leq a[left] + a[right] < target.$$

Therefore **discard the left value**, not the right one. If the sum is too large, every possible partner $i \geq left$ for the right value is too large. Discard the right value. An equal sum is a valid pair because the loop requires $left < right$.

left	right	sum	target	Decision
0	5	14	13	Too large: no remaining partner works with 12.
0	4	11	13	Too small: no remaining partner works with 2.
1	4	13	13	Found two different positions.

Contract for Example 02. Read n and target, then n values in nondecreasing order. Allow $0 \leq n \leq 200000$, $|a_i| \leq 10^9$, and $|target| \leq 2 \cdot 10^9$. Print YES or NO.

Example 02: Find a sorted pair | Complete C++23 program

```
#include <iostream>
#include <vector>
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int left = 0, right = n - 1;
    while (left < right) {
        long long sum = a[left] + a[right];
        if (sum == target) {
            std::cout << "YES\n";
            return 0;
        }
        if (sum < target) ++left;
        else --right;
    }
    std::cout << "NO\n";
}
```

Sample input

6 13
2 4 4 7 9 12

Sample output

YES

Every discarded endpoint has been proved unable to participate in a remaining answer. The remaining width decreases each time, so there are at most $n - 1$ comparisons of candidate sums. The scan is $O(n)$ time with $O(1)$ extra working memory. Negative numbers and duplicates are allowed: the proof uses *sorted order*, not positivity.

1.4 Halve an ordered interval to find a boundary

Suppose the question changes: “Where is the first value at least x ?” Scanning from the beginning works, but a comparison at the midpoint can classify many positions at once.

Define the answer as the first index b with $a[b] \geq x$, or $b = n$ if none exists. This is the **lower bound**. For our running array and $x = 4$, the answer is 1, not 2: equality alone is not enough; we need the *first* equality.

Maintain a half-open unknown interval

Start with $lo = 0$, $hi = n$. The invariant is:

$$a[i] < x \quad \text{for } i < lo, \quad a[i] \geq x \quad \text{for } i \geq hi \text{ within the array.}$$

The answer boundary lies in $[lo, hi]$. Only the *unclassified elements* lie in $[lo, hi]$. While $lo < hi$, the midpoint is a valid element index.

If $a[mid] < x$, sorted order makes every element at or before mid too small, so set $lo = mid + 1$. Otherwise, mid may be the answer, so keep it as a candidate boundary by setting $hi = mid$.

lo	hi	mid	value	Update for $x = 4$
0	6	3	7	hi = 3; the first acceptable value is no later than 3.
0	3	1	4	hi = 1; an earlier equal value might still exist.
0	1	0	2	lo = 1; index 0 is too small.

At $lo = hi = 1$, the unknown interval is empty and the classification proves the answer is 1. No final array access is needed to return this boundary.

Pause and predict: 4

What happens when every value is smaller than x ? What happens when the array is empty?

Check your prediction. In both cases, lower bound returns n . For the empty array, $n = 0$ and the loop never runs. A boundary result exists even though no matching element exists.

Pause and predict: 5

Why would $lo = mid$ be a broken replacement for $lo = mid + 1$ in this loop?

Check your prediction. When $hi = lo + 1$, the lower midpoint equals lo . Assigning $lo = mid$ changes nothing, so a false midpoint can cause an infinite loop.

Implement the invariant directly

Contract for Example 03. Read n , x , then n sorted values, with the same length and value bounds as Example 02. Print the zero-based lower-bound index, allowing n as the “none” boundary.

Example 03: Manual lower bound | Complete C++23 program

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

int firstAtLeast(const std::vector<long long>& a, long long x) {
    int lo = 0, hi = static_cast<int>(a.size());
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] < x) lo = mid + 1;
        else hi = mid;
    }
    return lo;
}

int main() {
    int n;
    long long x;
    std::cin >> n >> x;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::cout << firstAtLeast(a, x) << '\n';
}
```

Sample input

```
6 4
2 4 4 7 9 12
```

Sample output

```
1
```

The statement `hi = mid` does not mean that the midpoint was rejected as an answer. It means the midpoint no longer needs classification: it is already known to be acceptable, and the first acceptable position cannot lie after it.

Why the loop finishes

With $lo < hi$, the lower midpoint satisfies $lo \leq mid < hi$. Either the new lo is strictly larger or the new hi is strictly smaller. Each step leaves at most about half the elements unclassified. Repeated halving gives $O(\log(n+1))$ time and $O(1)$ working memory. Reading the entire input still costs $O(n)$.

Membership is a question about the boundary

Fragment after calling the helper

```
int index = firstAtLeast(a, x);
bool present = index < static_cast<int>(a.size()) && a[index] == x;
```

The bounds check must come first. A value greater than x is a valid lower bound but not an exact match. With $x = 6$, the lower bound is index 3, whose value is 7; membership is false.

1.5 Turn boundaries into exact and range queries

The **upper bound** of x is the first index whose value is *greater than* x , or n if none exists. The manual algorithm is the same classification process, but it skips values satisfying $a[mid] \leq x$ rather than only $a[mid] < x$.

Let $L(x)$ be lower bound and $U(x)$ be upper bound. Sorted order puts every occurrence of x in the half-open block $[L(x), U(x))$. Therefore:

Question	Answer
How many values are less than x ?	$L(x)$
How many are at most x ?	$U(x)$
How many equal x ?	$U(x) - L(x)$
First occurrence, if present?	$L(x)$
Last occurrence, if present?	$U(x) - 1$
How many values lie in $[A, B]$, with $A \leq B$?	$U(B) - L(A)$

“Last occurrence” requires a membership check; $U(x) - 1$ by itself means the last value *at most* x , which might be smaller than x .

The standard-library versions

The `<algorithm>` header supplies `std::lower_bound` and `std::upper_bound`. They return **iterators**, not numeric indices. For a vector, subtract `a.begin()` to get an index, and compare with `a.end()` before dereferencing. The searched range excludes its ending iterator.^{[1-1][1-2]}

`std::binary_search` returns a boolean membership result. `std::equal_range` returns both bounds as a pair of iterators. These functions use the chosen comparison order; sorting ascending and then searching with an inconsistent descending comparator is not valid.^[1-3]

Pause and predict: 6

For $[2, 4, 4, 7, 9, 12]$, what are $L(6)$ and $U(6)$? How many occurrences of 6 are there?

Check your prediction. Both boundaries are 3, so the count is zero. A missing value has an empty occurrence block.

One sorted array, several exact-value queries

Contract for Example 04. Read n, q with $0 \leq n, q \leq 200000$, then sorted values and q query values. Use $|a_i| \leq 10^9$ and $|x| \leq 2 \cdot 10^9$. For each query print lower-bound index, upper-bound index, and occurrence count.

Example 04: Boundaries and duplicate counts | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    while (q-- > 0) {
        long long x;
        std::cin >> x;
        auto first = std::lower_bound(a.begin(), a.end(), x);
        auto after = std::upper_bound(a.begin(), a.end(), x);
        std::cout << first - a.begin() << ' '
                  << after - a.begin() << ' '
                  << after - first << '\n';
    }
}
```

Sample input

```
6 3
2 4 4 7 9 12
4
6
20
```

Sample output

```
1 3 2
3 3 0
6 6 0
```

Predecessors and successors

For the first value at least x , use lower bound and require that it is not the end. For the largest value strictly below x , use the iterator immediately before lower bound, but only when lower bound is not the beginning. The largest value at most x is immediately before upper bound, under the same beginning check.

For a nearest-value query in a nonempty sorted array, only the predecessor of lower bound and lower bound itself can win. Everything farther left is no closer than that predecessor; everything farther right is no closer than lower bound. Compare both distances and apply the task's tie rule. Practice 13 develops the complete implementation.

Do not hide the container cost

For a vector, each boundary lookup takes $O(\log(n+1))$ work. The whole program takes $O(n+q \log(n+1))$ including reading, with $O(n)$ storage. The generic algorithms may take linear iterator movement on non-random-access iterators. For `set` or `map`, use their member `lower_bound` instead.^[1-3]

1.6 Solve the same pair problem with either method

Now return to target sum. If index i is fixed, the required partner value is $target - a[i]$. Search only indices *after* i . This prevents self-pairing and avoids giving both orders to the same pair.

The searched suffix is sorted. If a valid pair has indices $i < j$ after sorting, the loop will consider i , and the suffix search will find the partner value. Checking `it != a.end()` before reading `*it` handles a missing partner safely.

Compare the work, not the line count

Method	Work after sorting	What it reuses
Try all pairs	$O(n^2)$	Nothing between pair checks.
Fix a value and binary-search	$O(n \log(n + 1))$	Sorted order within each lookup.
Opposite-end two pointers	$O(n)$	The previous endpoints and eliminated pairs.

Sorting contributes $O(n \log n)$ comparisons when needed. Thus both faster approaches are $O(n \log n)$ overall on unsorted input, but the two-pointer pass performs less work afterwards. On already sorted input, that difference is not hidden by sorting.^[1-4]

Pause and predict: 7

May you sort an array when the task asks for a contiguous segment in the original order? How can you keep original IDs for a pair task?

Check your prediction. Sorting generally changes which values are adjacent, so it changes a subarray task. For a pair task, sort `(value, original_index)` records and return the saved IDs, as in Practice 03.

Implement the searched-partner approach

Contract for Example 05. Read the same values and target as Example 02, but the input may be unsorted. Print YES or NO. Sorting is allowed because the question asks only for existence, not original positions or contiguous segments.

Example 05: Fix one value and search its partner | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    for (int i = 0; i < n; ++i) {
        long long needed = target - a[i];
        auto it = std::lower_bound(a.begin() + i + 1,
                                   a.end(), needed);
        if (it != a.end() && *it == needed) {
            std::cout << "YES\n";
            return 0;
        }
    }
    std::cout << "NO\n";
}
```

Sample input

6 13
9 4 12 2 7 4

Sample output

YES

1.7 Count whole groups of sorted pairs

Finding one pair and counting all pairs need different answer updates. Suppose we want the number of position pairs $i < j$ with $a[i] + a[j] \leq X$.

At endpoints $left < right$, if their sum is at most X , then pairing $a[left]$ with *every* index from $left + 1$ through $right$ is valid. Add $right - left$, then move $left$. If the sum is too large, the current $right$ value has no valid remaining partner, so move $right$.

For $[2, 4, 4, 7, 9, 12]$ and $X = 11$, first discard 12. Then $2 + 9 = 11$ gives four pairs with the 2. The remaining 9 is too large for a 4, so discard it. The first 4 contributes two pairs; the second contributes one. The total is $4 + 2 + 1 = 7$.

Contract for Example 06. Read n, X and an arbitrary-order array, under the numeric bounds of Example 02. Count *position pairs*; duplicate values still represent separate choices.

Example 06: Count pairs with sum at most a limit | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long limit;
    std::cin >> n >> limit;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    int left = 0, right = n - 1;
    long long answer = 0;
    while (left < right) {
        if (a[left] + a[right] <= limit) {
            answer += right - left;
            ++left;
        } else {
            --right;
        }
    }
    std::cout << answer << '\n';
}
```

Sample input

6 11
2 4 4 7 9 12

Sample output

7

Each counted group has a different left endpoint, so no pair is counted twice. Every discarded right endpoint has no valid partner left. This establishes both completeness and uniqueness. The scan uses $O(n)$ time after sorting and $O(1)$ working memory; the stored input uses $O(n)$ memory. Sorting may use implementation working memory separately.

Pause and predict: 8

For $[3, 3, 3]$ and $X = 6$, should the answer be 1 or 3?

Check your prediction. It is 3: positions $(0, 1)$, $(0, 2)$, and $(1, 2)$ are different pairs. Counting distinct *value combinations* would be a different problem.

1.8 Merge two sorted sequences without restarting

Two pointers do not have to belong to the same array. Suppose two sorted groups of scores must become one sorted list. Keep i at the next unread value of the first list and j at the next unread value of the second.

The next output value is the smaller of the two available values. Why? Every later value in the first list is at least its current head; the same is true in the second list. Nothing hidden later can be smaller than both heads.

Unread first list	Unread second list	Next action
2, 4, 4, 9	1, 4, 7	Output 1; advance only the second index.
2, 4, 4, 9	4, 7	Output 2; advance only the first index.
4, 4, 9	4, 7	Output the first list's 4; keep the other 4 for later.
4, 9	4, 7	Output the first list's next 4.
9	4, 7	Output 4, then 7, then the remaining 9.

The output is always a sorted prefix containing exactly the values already consumed. Every iteration consumes one value, including the two tail loops. Total work is $O(n + m)$; the result needs $O(n + m)$ output storage. The pointer variables use $O(1)$ additional working memory.

Pause and predict: 9

When one input ends, should the algorithm stop immediately?

Check your prediction. No. Every unread value in the other sorted input must still be copied. The main comparison loop stops, but the appropriate tail loop continues.

Implement the merge and finish both tails

Contract for Example 07. Read n, m , then two nondecreasing arrays with $0 \leq n, m$ and $n + m \leq 200000$. Values satisfy $|a_i|, |b_i| \leq 10^9$. Print every value of the merged sorted sequence, retaining duplicates. Print EMPTY only when both arrays are empty.

Example 07: Merge two sorted inputs | Complete C++23 program

```
#include <iostream>
#include <vector>
int main() {
    int n, m;
    std::cin >> n >> m;
    std::vector<long long> a(n), b(m), merged;
    for (auto& value : a) std::cin >> value;
    for (auto& value : b) std::cin >> value;
    int i = 0, j = 0;
    while (i < n && j < m) {
        if (a[i] <= b[j]) merged.push_back(a[i++]);
        else merged.push_back(b[j++]);
    }
    while (i < n) merged.push_back(a[i++]);
    while (j < m) merged.push_back(b[j++]);
    if (merged.empty()) std::cout << "EMPTY";
    for (int k = 0; k < n + m; ++k) {
        if (k > 0) std::cout << ' ';
        std::cout << merged[k];
    }
    std::cout << '\n';
}
```

Sample input

```
4 3
2 4 4 9
1 4 7
```

Sample output

```
1 2 4 4 4 7 9
```

1.9 Answer many queries with searches or a sweep

You have one collection of values and many questions: “How many values are at most x ?” There are two reasonable approaches.

Independent queries. Sort the values once, then use upper bound for every x . Query order does not matter. After sorting, this costs $O(q \log(n+1))$ for q queries. This is natural when queries must be answered in their arrival order.

Ordered queries. If all queries are available in advance and their answers do not affect later queries, sort them by x . A boundary that counts values at most a smaller threshold never has to move backwards for a larger threshold. This is called an *offline sweep*: reorder the independent work, but restore the requested output order.

A trace using the same running array

For sorted $a = [2, 4, 4, 7, 9, 12]$, the original query order is $[6, 1, 4, 12]$. Store $(\text{threshold}, \text{original_id})$, then process:

threshold	original ID	newly consumed	total consumed
1	1	0	0
4	2	3	3
6	0	0	3
12	3	3	6

Store each count at its original ID. The final output is 3, 0, 3, 6, not the sorted-query order 0, 3, 3, 6.

What the shared boundary means

Before answering threshold x , advance **used** while the next value is at most x . Now the first **used** elements qualify and every remaining element is too large. The boundary is exactly $U(x)$, the same answer that upper bound would find independently.

The difference is not the answer definition. It is whether neighbouring queries can **reuse the previous boundary**.

Pause and predict: 10

Can you sort queries when the next query is generated using the answer to the previous one?

Check your prediction. Not by this simple offline method. The queries are no longer an independent batch that can be reordered without changing the task.

Store answers by original query ID

Contract for Example 08. Read n, q , an arbitrary-order array, and q thresholds. Allow $0 \leq n, q \leq 200000$, $|a_i| \leq 10^9$, and $|x| \leq 2 \cdot 10^9$. Print each count on its own line in the *original* query order.

Example 08: A shared boundary across sorted queries | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::vector<std::pair<long long, int>> queries(q);
    for (int id = 0; id < q; ++id) {
        std::cin >> queries[id].first;
        queries[id].second = id;
    }
    std::sort(a.begin(), a.end());
    std::sort(queries.begin(), queries.end());
    std::vector<int> answer(q);
    int used = 0;
    for (const auto& query : queries) {
        while (used < n && a[used] <= query.first) ++used;
        answer[query.second] = used;
    }
    for (int count : answer) std::cout << count << '\n';
}
```

Sample input

```
6 4
9 4 12 2 7 4
6
1
4
12
```

Sample output

```
3
0
3
6
```

The scan performs at most n boundary advances and q answers. Its cost is $O(n + q)$ *after both sorts*. The full cost is $O(n \log n + q \log q + n + q)$, with $O(n + q)$ stored values, query records, and answers.

The sweep is not always the better choice

If q is small, or the values are already sorted but queries arrive online, separate binary searches are simpler and can be cheaper. Sorting a huge batch of queries is extra work. Choose based on which data is already ordered and which operations the problem actually permits.

When both values and thresholds are already sorted, the sweep needs no preprocessing. This is the clearest case where reusing a boundary changes q independent searches into one coordinated scan.

1.10 Use forward positions without changing the order

Sorted numeric values are not required for every two-pointer pattern. Sometimes the useful order is simply the order in which input must be processed.

Match a subsequence

To check whether `ace` is a subsequence of `abcde`, keep one position in the smaller string and scan the larger string. Match the next required character as early as possible. Skipped characters in the larger string do not need to be revisited.

Choosing an earlier matching position cannot leave *fewer* later positions available than choosing a later matching position. Thus, if any complete match exists, the earliest-match policy can still complete it. The matched prefix length never decreases.

Contract for Example 09. Read two ASCII lines: the required subsequence, then the containing text. Each may be empty and has length at most 200000. Spaces and case are significant. Print YES or NO.

Example 09: Two progress positions in text | Complete C++23 program

```
#include <iostream>
#include <string>
int main() {
    std::string small, large;
    std::getline(std::cin, small);
    std::getline(std::cin, large);
    int matched = 0;
    int required = static_cast<int>(small.size());
    for (char current : large) {
        if (matched < required && small[matched] == current) {
            ++matched;
        }
    }
    std::cout << (matched == required ? "YES" : "NO") << '\n';
}
```

Sample input

ace
abcde

Sample output

YES

`matched < required` prevents reading past the smaller string after it has been fully matched. An empty required string succeeds immediately. The processing cost is $O(m)$ for a text of length m ; reading and storing both strings costs $O(n + m)$.

Pause and predict: 11

Is `ac` a substring of `abc`? Is it a subsequence?

Check your prediction. It is not a substring because its letters are not adjacent. It is a subsequence because the letters appear in order with a skipped character between them.

Read and write different positions in one array

Now consider a sorted array with repeated values. We want to keep one copy of each distinct value. `read` identifies the next input element; `written` is the size of the accepted output prefix.

Accept a value when the output is empty or the value differs from the last accepted value. Write it at `a[written]`, then increment `written`. The write position never passes the read position, so the algorithm never overwrites an unread future element.

Contract for Example 10. Read n and a sorted array under the ordinary bounds. Print its number of distinct values, then those values on one line. For $n = 0$, print 0 followed by EMPTY on the next line.

Example 10: Compact a sorted sequence | Complete C++23 program

```
#include <iostream>
#include <vector>
int main() {
    int n;
    std::cin >> n;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int written = 0;
    for (int read = 0; read < n; ++read) {
        if (written == 0 || a[read] != a[written - 1]) {
            a[written] = a[read];
            ++written;
        }
    }
    a.resize(written);
    std::cout << written << '\n';
    if (written == 0) std::cout << "EMPTY";
    for (int i = 0; i < written; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << a[i];
    }
    std::cout << '\n';
}
```

Sample input

7
2 2 4 4 4 9 12

Sample output

4
2 4 9 12

The invariant is that $a[0 \dots \text{written})$ contains exactly one copy of each value processed so far, in order. Sorted input makes equal values consecutive. The scan takes $O(n)$ time and $O(1)$ working memory beyond the vector. Resizing removes the unused suffix from the logical sequence; it is not what removes the duplicates.

A related pattern with a different condition

For stable filtering, keep an element whenever it satisfies a condition, not whenever it differs from the previous element. The condition “value is not zero” can move all nonzero values to the front without changing their relative order. Fill the remaining suffix with zeros afterwards. Practice 07 asks you to implement that variation.

Do not apply sorted deduplication directly to unsorted data: $[4, 2, 4]$ would still contain the two 4s because they are not adjacent.

1.11 Slide a fixed-length window by changing two contributions

A window is a contiguous segment whose boundaries move. For a fixed length k , every step removes one old value and adds one new value. The array does *not* need to be sorted, and its values may be negative.

Suppose the current sum describes $a[l] + \dots + a[r]$, where $r - l + 1 = k$. The next window is $[l + 1, r + 1]$, so

$$newSum = oldSum - a[l] + a[r + 1].$$

Everything between those two boundaries is unchanged. Recalculating the middle values would repeat work.

Contract for Example 11. Read n, k with $1 \leq k \leq n \leq 200000$, then values with $|a_i| \leq 10^9$. Print the largest sum of a length- k subarray.

Example 11: Maximum sum at a fixed length | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    long long sum = 0;
    for (int i = 0; i < k; ++i) sum += a[i];
    long long best = sum;
    for (int right = k; right < n; ++right) {
        sum += a[right];
        sum -= a[right - k];
        best = std::max(best, sum);
    }
    std::cout << best << '\n';
}
```

Sample input

```
6 3
2 1 3 1 1 4
```

Sample output

```
6
```

For $[2, 1, 3, 1, 1, 4]$ and $k = 3$, the four window sums are 6, 5, 5, 6. They correspond to starts 0, 1, 2, 3. Initializing **best** from the first window, rather than zero, also makes all-negative inputs correct.

The cost follows from the changes

Building the first window takes k additions. The remaining $n - k$ windows each need one addition, one subtraction, and an answer comparison. Total time is $O(n)$ and the working state beyond the input is $O(1)$.

Pause and predict: 12

For $[-5, -2, -4]$ and $k = 2$, why would initializing the answer to zero be wrong?

Check your prediction. The actual window sums are -7 and -6 , so the maximum is -6 . Zero is not the sum of any permitted window.

1.12 Keep the longest nonnegative window within a budget

Now the length may change. For nonnegative values, find the longest subarray with sum at most B , where $B \geq 0$.

When a new rightmost value arrives, add it. If the sum is too large, remove values from the left until the window is valid again. Then measure its length. This order is **expand, repair, measure**.

Why moving the left endpoint is safe

With nonnegative values, extending a window cannot reduce its sum. Once a left endpoint becomes invalid at some right endpoint, keeping that left endpoint will also be invalid for every later right endpoint. It will never be needed again for this budget.

After repair, every earlier start has already been proved invalid, and the retained window is the longest valid window ending at the current right endpoint. Taking the maximum over all right endpoints therefore considers a best candidate for every possible ending position.

right	added	left after repair	sum	Valid window, $B = 6$
0	2	0	2	[2], length 1.
1	1	0	3	[2, 1], length 2.
2	3	0	6	[2, 1, 3], length 3.
3	1	1	5	Remove 2; [1, 3, 1], length 3.
4	1	1	6	[1, 3, 1, 1], length 4.
5	4	3	6	Remove 1, then 3; [1, 1, 4], length 3.

The repair needs a **while**, not a single **if**. The final row needed two removals.

Pause and predict: 13

Must the values be sorted for this sum-window argument? Can the array contain zeros?

Check your prediction. Sorted order is not required and should not be introduced by sorting. Nonnegativity is required. Zeros are allowed because extending or shrinking may leave the sum unchanged without reversing the needed inequalities.

Implementation and empty-window safety

Contract for Example 12. Read n, B , then n values, with $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $0 \leq B \leq 2 \cdot 10^{14}$. Print the maximum valid length; print 0 when no nonempty subarray qualifies.

Example 12: Longest affordable segment | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long budget;
    std::cin >> n >> budget;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int left = 0, best = 0;
    long long sum = 0;
    for (int right = 0; right < n; ++right) {
        sum += a[right];
        while (sum > budget) sum -= a[left++];
        best = std::max(best, right - left + 1);
    }
    std::cout << best << '\n';
}
```

Sample input

```
6 6
2 1 3 1 1 4
```

Sample output

```
4
```

If a single value exceeds the budget, repair may move `left` to `right + 1`. The window is then empty, its sum is zero, and its length formula gives zero. Because $B \geq 0$, repair stops before reading another element from that empty window.

A nested loop is not automatically quadratic

The right index advances exactly n times. Across the *entire program*, the left index also advances at most n times. It never restarts for each right endpoint. Consequently, the total number of window updates is at most $2n$, and the scan is $O(n)$ time with $O(1)$ working memory.

Counting “a loop inside a loop” would miss the shared progress. Count how often the indices can move instead.

1.13 Find a shortest window by measuring before removal

The task is now the shortest nonempty subarray with sum at least T , with nonnegative values and $T > 0$. A larger sum is no longer a failure: it means the current window is a candidate answer.

Append the next value. While the sum reaches T , first record the window length, then remove its leftmost value to try a shorter window. The order is **expand, measure while valid, shrink**.

For $[2, 1, 3, 1, 1, 4]$ and $T = 6$, the first valid window is $[2, 1, 3]$ of length 3. Removing 2 makes its sum 4, so expansion must resume. Later windows can be tested and shortened, but no length-2 window in this input reaches 6.

Why a removed left endpoint is no longer needed

When a start *left* is removed, the program has already measured a valid window from that start. Any future window with the same start ends later and cannot be shorter. If the current sum is below the target, removing more nonnegative values cannot repair it; expansion is the next possible improvement.

Each endpoint advances at most n times. The time is $O(n)$ and the extra working memory is $O(1)$. Requiring $T > 0$ is important: an empty window has sum zero, so it cannot keep this shrinking loop running.

Pause and predict: 14

Why is it wrong to remove the left value first and record the answer only after the sum falls below T ?

Check your prediction. The remaining window may be invalid. The valid candidate must be measured before the removal that destroys its validity.

Implement the shortest-target scan

Contract for Example 13. Use $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $1 \leq T \leq 2 \cdot 10^{14}$. Read n, T and the values. Print the minimum length, or 0 if no such nonempty subarray exists.

Example 13: Shortest segment reaching a target | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int left = 0, best = n + 1;
    long long sum = 0;
    for (int right = 0; right < n; ++right) {
        sum += a[right];
        while (sum >= target) {
            best = std::min(best, right - left + 1);
            sum -= a[left++];
        }
    }
    std::cout << (best == n + 1 ? 0 : best) << '\n';
}
```

Sample input

Sample output

6 6

3

2 1 3 1 1 4

1.14 Count every valid window instead of keeping one length

For each right endpoint, suppose repair has left the smallest valid start *left* for a sum limit $K \geq 0$. Every start s with $left \leq s \leq right$ also qualifies: removing additional nonnegative values cannot increase the sum.

There are exactly $right - left + 1$ such starts. Add that number, not merely 1. The groups for different right endpoints are disjoint, because a subarray has exactly one ending position.

Contract for Example 14. Read n, K and nonnegative values. Use $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $-1 \leq K \leq 2 \cdot 10^{14}$. Count nonempty subarrays with sum at most K . For negative K , the answer is zero.

Example 14: Count subarrays with sum at most a limit | Complete C++23 program

```
#include <iostream>
#include <vector>
long long countAtMost(const std::vector<long long>& a,
                     long long limit) {
    if (limit < 0) return 0;
    int left = 0;
    long long sum = 0, count = 0;
    for (int right = 0; right < static_cast<int>(a.size()); ++right) {
        sum += a[right];
        while (sum > limit) sum -= a[left++];
        count += right - left + 1;
    }
    return count;
}
int main() {
    int n;
    long long limit;
    std::cin >> n >> limit;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::cout << countAtMost(a, limit) << '\n';
}
```

Sample input

```
5 1
0 1 0 2 0
```

Sample output

```
7
```

Zeros create several valid choices, not just one

At right endpoint 2 of $[0, 1, 0, 2, 0]$ with limit 1, the starts 0, 1, and 2 give sums 1, 1, and 0. All three must be counted. A method that increments the answer once whenever a window is valid misses valid subarrays.

There can be $n(n+1)/2$ nonempty subarrays. At $n = 200000$, that is 20,000,100,000, so the count must not be stored in a typical 32-bit `int`. The algorithm still needs only $O(n)$ time and $O(1)$ working memory.

Pause and predict: 15

How many nonempty subarrays of $[0, 0]$ have sum at most 0?

Check your prediction. Three: the first zero, the second zero, and both together. The contributions by ending position are 1 and 2.

1.15 Maintain frequencies when a sum is not the right state

Not every window is described by a total. To find the longest substring with at most k distinct lowercase letters, maintain a frequency for each letter and a count of how many frequencies are positive.

Adding the first copy of a letter increases `distinct`. Adding another copy does not. Removing the last copy decreases `distinct`. Removing one copy while others remain does not.

The condition has the same repair structure as the budget window: extending may introduce another distinct letter; removing characters cannot introduce a new one. Repair until `distinct` $\leq k$, then measure.

Contract for Example 15. Read k , then a nonempty lowercase ASCII word of length at most 200000, with $0 \leq k \leq 26$. Print the longest valid substring length.

Example 15: Longest substring with at most k distinct letters | Complete C++23 program

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string>
int main() {
    int k;
    std::string text;
    std::cin >> k >> text;
    std::array<int, 26> frequency{};
    int left = 0, distinct = 0, best = 0;
    for (int right = 0; right < static_cast<int>(text.size()); ++right) {
        if (++frequency[text[right] - 'a'] == 1) ++distinct;
        while (distinct > k) {
            if (--frequency[text[left] - 'a'] == 0) --distinct;
            ++left;
        }
        best = std::max(best, right - left + 1);
    }
    std::cout << best << '\n';
}
```

Sample input

2 abaccc

Sample output

4

For abaccc and $k = 2$, adding the first c to aba creates three distinct letters. Removing the first a is insufficient because another a remains. Removing b repairs the window. The remaining accc eventually has length 4.

Choose the storage from the domain

A 26-entry array is sufficient because the contract permits only lowercase ASCII letters. For large arbitrary integer keys, a map or hash map may be needed; include the cost of its operations in your analysis. This particular program has $O(n)$ time and $O(26)$ working memory beyond the string.

Pause and predict: 16

For an active window containing aab, what happens to the distinct count after removing the first a? What happens after removing the second?

Check your prediction. The first removal leaves one a, so distinct stays at 2. The second makes the a frequency zero, so distinct becomes 1.

1.16 Connect prefix searches to a moving window

Return to the shortest nonnegative subarray with sum at least T . We already have a linear window solution. Now derive a binary-search solution to see exactly what work the window avoids repeating. Define $P[0] = 0$ and let $P[i]$ be the sum of the first i elements. Then the half-open segment $[l, e)$ has sum

$$P[e] - P[l].$$

For a fixed start l , reaching the target means $P[e] \geq P[l] + T$. Because the original values are nonnegative, P is nondecreasing. Find the *first* qualifying endpoint with lower bound.

Search only boundaries $e \geq l + 1$ so the chosen subarray is nonempty. A found endpoint gives length $e - l$. If the search reaches the end of the prefix vector, this start has no feasible endpoint.

start l	$P[l] + 6$	first endpoint e	Length for $[2, 1, 3, 1, 1, 4]$
0	6	3	$3 - 0 = 3$
1	8	5	$5 - 1 = 4$
2	9	6	$6 - 2 = 4$
3	12	6	$6 - 3 = 3$
4	13	none	No remaining segment reaches 6.
5	14	none	No remaining segment reaches 6.

Where the repeated work remains

Each fixed start performs an independent lower-bound search. But as l increases, both $P[l] + T$ and the first feasible endpoint never move backwards. That is the same shared-boundary opportunity we used for ordered queries.

A forward endpoint scan can therefore reuse its previous endpoint instead of searching again. The shrinking-window solution organizes the same monotone boundary from the other direction: for each ending position, remove starts whose shortest useful result has already been measured.

Pause and predict: 17

Must the prefix values be strictly increasing for lower bound to work?

Check your prediction. No. Nondecreasing order is sufficient. Zero-valued elements create repeated prefix values, and lower bound deliberately handles repeated values.

Implement the alternative and compare the answers

Contract for Example 16. Use exactly the input and output contract of Example 13. This lets you test the two implementations on the same cases.

Example 16: Shortest segment using prefix sums and lower bound | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> prefix(n + 1, 0);
    for (int i = 0; i < n; ++i) {
        long long value;
        std::cin >> value;
        prefix[i + 1] = prefix[i] + value;
    }
    int best = n + 1;
    for (int left = 0; left < n; ++left) {
        auto it = std::lower_bound(prefix.begin() + left + 1,
                                   prefix.end(), prefix[left] + target);
        if (it != prefix.end()) {
            int end = static_cast<int>(it - prefix.begin());
            best = std::min(best, end - left);
        }
    }
    std::cout << (best == n + 1 ? 0 : best) << '\n';
}
```

Sample input

```
6 6
2 1 3 1 1 4
```

Sample output

```
3
```

Which implementation should you choose?

The prefix method spends $O(n)$ time building the summary and $O(n \log(n+1))$ time on the searches. It stores $n+1$ prefix totals. The ordinary window takes $O(n)$ time and $O(1)$ working memory beyond the stored input.

For this one target on a nonnegative array, the window is normally the simpler efficient choice once its invariant is understood. The prefix method is still valuable: it directly converts a sum condition into a searched boundary and connects to other problems where a prefix table is already useful.

Do not conclude that every prefix-based question becomes a two-pointer problem. The boundary must move monotonically between the particular queries you are answering. Independently ordered thresholds often still favour separate binary searches.

A useful testing habit

For small random nonnegative arrays, compare three answers: enumerate every subarray, run the window, and run the prefix search. An agreement is useful evidence; the separate correctness arguments explain why the methods work for all allowed inputs. The supplied tests make these comparisons against an independent direct enumeration.

1.17 Recognize when a tempting shortcut is invalid

The most dangerous mistake is to apply a familiar loop after silently changing its assumptions. Test the *reason for the move*, not just its syntax.

Negative values break these sum-window arguments

For $[5, -4]$ and budget 1, the whole length-2 subarray is valid. A longest-window algorithm that discards 5 immediately never recovers that start. With negative values, a later extension can make an invalid sum valid again.

For $[1, -1, 5]$ and target 5, the shortest valid subarray is $[5]$. A shortest-window algorithm may stop shrinking after removing the first 1, because the remaining sum drops below 5. Removing the next value, -1 , would *increase* the sum again. The nonnegative proof no longer applies.

The prefix array for $[5, -4]$ is $[0, 5, 1]$, which is not sorted. Ordinary lower bound cannot be applied to it as though it were nondecreasing.

Sorting can repair one promise while destroying another

For $[4, 100, 4]$ with budget 8, the longest valid original subarray has length 1. Sorting creates $[4, 4, 100]$, where a length-2 window qualifies. That is a different answer to a different sequence. You may sort for an unordered pair task, not automatically for a contiguous-segment task.

Equality alone is not a monotone predicate

As you extend $[1, 2, 3]$, the prefix sums are 1, 3, 6. The test “sum equals 3” produces false, true, false. A first-true binary search requires the true region to continue after it begins, so this equality test is not suitable.

You can instead search “sum is at least 3” and then check equality at the resulting boundary. Similarly, exact-sum *counting* with zeros needs more care than returning one matching window. The optional exact-count section derives a safe approach.

Two index variables do not prove a linear-time algorithm

If the second index restarts for every first index, there may still be $O(n^2)$ advances. If it only moves forward globally, count those total advances. If each advance performs an expensive operation, include that operation’s cost as well.

Pause and predict: 18

Is the failure of a nonnegative sliding window evidence that no efficient algorithm exists for signed values?

Check your prediction. No. It only means that this particular invariant and update rule are not justified. Signed variants may need a different summary or data structure; they are not solved by removing the input restriction from this code.

1.18 Binary-search a possible answer instead of an index

So far, binary search has located a boundary in stored sorted data. It can also locate a boundary in a sequence of answers that we never explicitly store.

Suppose a task asks for the smallest machine time that produces enough items. For each candidate time t , define a **predicate**, a function returning true or false:

$can(t) = \text{“at least the required number of items are finished by time } t\text{.”}$

If a time is sufficient, every larger time is also sufficient. Across increasing times, the predicate has the pattern

false, false, ..., false, true, true, ..., true.

The answer is the **first true** value. This is the same boundary shape as “first array value at least x ,” but evaluating one position now runs a checker rather than reading one element.

Specify the checker before the search loop

Write what one candidate means, which answers are allowed, what true means, why the truth value is monotone, and how to compute it. A monotone-looking formula is not enough if the implementation computes it incorrectly or overflows.

Choose a closed candidate interval $[lo, hi]$ containing the answer. For a first-true search, prove that at least one feasible candidate exists; commonly, prove $can(hi)$. You do not need $can(lo)$ to be false.

Fragment inside a helper: `can` is a supplied monotone predicate

```
while (lo < hi) {
    long long mid = lo + (hi - lo) / 2;
    if (can(mid)) hi = mid;           // mid may be the first success
    else lo = mid + 1;               // mid and all smaller values fail
}
return lo;
```

The invariant and the stopping condition

The first feasible value remains inside $[lo, hi]$. A true midpoint rules out larger answers as the first feasible answer; a false midpoint rules out itself and every smaller value. The width shrinks until one candidate remains.

If the interval contains R integer candidates and a checker costs C , the search costs $O(C \log R)$ for a nontrivial interval. Do not write $O(\log R)$ when each check scans an array.

Pause and predict: 19

What happens if every candidate is false but you run this template without checking feasibility?

Check your prediction. The loop still returns a boundary, often the original upper endpoint, but that value is not a valid solution. An existing feasible upper bound or a separate impossible-case check is part of the algorithm.

1.19 Find the last feasible answer and protect arithmetic

A maximization task often has the opposite truth pattern:

true, true, ..., true, false, false, ..., false.

We want the last true value. Start with a candidate interval containing the answer and with a known feasible lower bound. A true midpoint moves the lower bound *to* the midpoint. To guarantee progress, use the **upper midpoint**.

Fragment inside a helper; $hi - lo + 1$ must fit the chosen type

```
while (lo < hi) {
    long long mid = lo + (hi - lo + 1) / 2;
    if (can(mid)) lo = mid;
    else hi = mid - 1;
}
return lo;
```

If $hi = lo + 1$, the upper midpoint is hi . A lower midpoint would equal lo , and $lo = mid$ could fail to advance.

A small numerical application: floor square root

Find the largest integer x with $x^2 \leq N$. The predicate is true and then false as x increases. Searching $[0, N]$ contains the answer for every $N \geq 0$.

But $mid * mid$ can overflow even when the final square root fits. For positive mid , the equivalent integer test $mid \leq N / mid$ avoids the product. Integer division truncates towards zero, so for nonnegative operands it gives the floor.^{[1-5][1-6]}

When $N = 0$, the loop does not run. Otherwise, any tested upper midpoint is positive, so division by zero cannot occur. The search takes $O(\log(N + 1))$ time and $O(1)$ space.

Pause and predict: 20

Does storing $mid * mid$ in a long long variable make an overflowing multiplication safe?

Check your prediction. No. The expression must already be safe when it is evaluated. A wider destination does not repair an earlier overflowing operation, and even long long operands can produce a product too large for long long.

Implement the safe integer square-root search

Contract for Example 17. Read an integer N with $0 \leq N \leq 10^{18}$. Print $\lfloor \sqrt{N} \rfloor$.

Example 17: Last feasible integer square root | Complete C++23 program

```
#include <iostream>
int main() {
    long long n;
    std::cin >> n;
    long long lo = 0, hi = n;
    while (lo < hi) {
        long long mid = lo + (hi - lo + 1) / 2;
        if (mid <= n / mid) lo = mid;
        else hi = mid - 1;
    }
    std::cout << lo << '\n';
}
```

Sample input

Sample output

50

7

Trace the candidate values for $N = 53$

lo	hi	upper mid	Decision before the next iteration
0	53	27	$27 > 53/27$: too large; set hi to 26.
0	26	13	$13 > 53/13$: too large; set hi to 12.
0	12	6	$6 \leq 53/6$: feasible; set lo to 6.
6	12	9	$9 > 53/9$: too large; set hi to 8.
6	8	7	$7 \leq 53/7$: feasible; set lo to 7.
7	8	8	$8 > 53/8$: too large; set hi to 7.

The final interval is $[7, 7]$. The mathematical check is $7^2 = 49 \leq 53 < 64 = 8^2$. The program did not need to compute those squares during the search; its division test avoided an unsafe intermediate product for large inputs.

1.20 Find minimum production time with a monotone count

There are n machines. Machine i takes t_i time units per item, operates independently, and completes its first item at time t_i . Find the first integer time when at least k items have been produced.

At time T , machine i has completed $\lfloor T/t_i \rfloor$ items. Add these contributions and test whether the total reaches k . Every individual contribution is nondecreasing in T , so the sum is also nondecreasing.

Prove a usable bound

If the fastest machine needs $t_{\min}$ per item, it can produce all k items by itself in $t_{\min}k$ time. Thus $[0, t_{\min}k]$ contains the first feasible time. With $t_i, k \leq 10^9$, the product is at most 10^{18} and fits in the chosen type. For $k = 0$, the answer is 0.

For times $[3, 5, 7]$ and target 8, time 12 gives $4 + 2 + 1 = 7$ items. Time 14 gives $4 + 2 + 2 = 8$. Time 13 still gives 7, so the first feasible time is 14.

Stop counting once the decision is known

Even with a safe time bound, the theoretical sum of contributions from many fast machines might be enormous. The checker only needs to know whether the target is reached. Compare the next contribution with `target - made` before adding it. If it is large enough, return true immediately.

The search makes $O(\log(t_{\min}k + 1))$ checks; each scans at most n machines. Total time is $O(n \log(t_{\min}k + 1))$ plus input, with $O(n)$ stored times and $O(1)$ checker state.

Pause and predict: 21

Should the checker return true only when production equals k exactly?

Check your prediction. No. The task allows at least k items. Several machines may finish together and skip over the exact count. Equality also would not produce a false-then-true predicate.

Implement a capped production checker

Contract for Example 18. Read n, k and the positive machine times. Use $1 \leq n \leq 200000$, $1 \leq t_i \leq 10^9$, and $0 \leq k \leq 10^9$. Print the minimum time.

Example 18: Minimum time to finish enough items | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
bool enough(const std::vector<long long>& times,
            long long target, long long time) {
    long long made = 0;
    for (long long cycle : times) {
        long long contribution = time / cycle;
        if (contribution >= target - made) return true;
        made += contribution;
    }
    return made >= target;
}
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> times(n);
    for (auto& cycle : times) std::cin >> cycle;
    long long lo = 0;
    long long hi = *std::min_element(times.begin(), times.end()) * target;
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (enough(times, target, mid)) hi = mid;
        else lo = mid + 1;
    }
    std::cout << lo << '\n';
}
```

Sample input

Sample output

3 8

14

3 5 7

1.21 Maximize a cut height by reversing the truth direction

A horizontal cut at integer height H collects $\max(0, a_i - H)$ units from each item of height a_i . Find the greatest permitted height that still collects at least W units.

Raising H never increases collected material, so the predicate “enough material remains” is true and then false. This is a last-true search, unlike minimum production time.

State the permitted domain and the impossible case

The allowed cut heights are integers from 0 through the tallest input height. At height 0, all material is collected. If that total is smaller than W , no allowed answer exists. If $W = 0$, the answer is the tallest input height, because that is the largest height in the permitted domain.

For heights $[9, 13, 6]$ and $W = 10$, cutting at 6 gives $3 + 7 + 0 = 10$. Cutting at 7 gives only $2 + 6 + 0 = 8$. Thus 6 is feasible and every greater candidate fails.

The total material is at most $2 \cdot 10^{14}$ under the contract, so the sum fits in `long long`. The checker is $O(n)$, the search is $O(n \log(H_{\max} + 1))$, and the input needs $O(n)$ storage.

Pause and predict: 22

When a trial cut is feasible, should the search keep smaller cut heights or investigate larger ones?

Check your prediction. It should investigate larger heights while retaining the feasible midpoint as a candidate. The goal is the *greatest* feasible height, not merely any feasible height.

Implement the last-feasible cut search

Contract for Example 19. Read n, W and n heights, with $1 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $0 \leq W \leq 2 \cdot 10^{14}$. Print the greatest valid cut height in $[0, \max a_i]$, or -1 if impossible.

Example 19: Largest cut height that meets a requirement | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long required;
    std::cin >> n >> required;
    std::vector<long long> height(n);
    long long total = 0;
    for (auto& value : height) {
        std::cin >> value;
        total += value;
    }
    if (total < required) {
        std::cout << "-1\n";
        return 0;
    }
    long long lo = 0;
    long long hi = *std::max_element(height.begin(), height.end());
    while (lo < hi) {
        long long mid = lo + (hi - lo + 1) / 2;
        long long wood = 0;
        for (long long value : height) wood += std::max(0LL, value - mid);
        if (wood >= required) lo = mid;
        else hi = mid - 1;
    }
    std::cout << lo << '\n';
}
```

Sample input

Sample output

3 10

6

9 13 6

1.22 Partition a sequence: prove both the check and the search

Given nonnegative workloads in their original order, split them into at most k nonempty consecutive groups. Minimize the largest group sum. You cannot reorder the workloads or split one workload between groups.

For a candidate capacity C , ask whether the sequence can be partitioned using at most k groups, each of sum at most C . If capacity C works, every larger capacity also works using the same groups. The outer search is therefore first true.

Build a greedy checker

Fill the current group from left to right until the next value would exceed C . Then start a new group. If an individual value exceeds C , the candidate is impossible.

This greedy rule requires a proof: a checker that rejects a feasible capacity would break the search even if its true/false results happened to look monotone.

Why filling the current group is safe

After one group, greedy has consumed at least as many items as any feasible partition's first group: it stopped only when another item would exceed capacity.

Assume greedy has consumed at least as far after j groups. A competing partition's next group begins no later than greedy's next unread item. Because all values are nonnegative, the suffix of that competing group remaining after greedy's start also fits in capacity C . Greedy can therefore reach at least the competing partition's next endpoint.

By induction, greedy never needs more groups than a feasible competing partition. If greedy exceeds k , no permitted partition exists. If greedy uses at most k , its own groups are a witness of feasibility.

Bounds come from unavoidable and sufficient choices

Every capacity must be at least the largest workload. The sum of all workloads is sufficient because one group can contain everything. Search between those two bounds.

For $[2, 1, 3, 1, 1, 4]$ and $k = 3$, capacity 4 produces $[2, 1] \mid [3, 1] \mid [1] \mid [4]$, four groups. Capacity 5 produces $[2, 1] \mid [3, 1, 1] \mid [4]$, three groups. The optimum is 5.

Pause and predict: 23

Why is “the checker is greedy” not by itself a correctness argument?

Check your prediction. A locally attractive choice may block a better later arrangement. Here the endpoint comparison proves that the greedy choice never falls behind a competing feasible partition. That proof, not the word greedy, makes the checker reliable.

The connection back to windows

The checker is another left-to-right segmentation scan: the current sum describes the unfinished group, and every element is processed once per candidate capacity. Binary search decides *which capacity* to test; the scan decides *whether it works*. Counting only the outer iterations would omit most of the work.

Implementation with original order preserved

Contract for Example 20. Read n, k and nonnegative values. Use $1 \leq k \leq n \leq 200000$ and $0 \leq a_i \leq 10^9$. Print the minimum possible largest group sum for *at most* k groups.

Example 20: Minimum capacity for consecutive groups | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
bool fits(const std::vector<long long>& a, int k, long long capacity) {
    int groups = 1;
    long long used = 0;
    for (long long value : a) {
        if (value > capacity) return false;
        if (value > capacity - used) {
            ++groups;
            used = 0;
        }
        used += value;
        if (groups > k) return false;
    }
    return true;
}
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    long long lo = *std::max_element(a.begin(), a.end());
    long long hi = std::accumulate(a.begin(), a.end(), 0LL);
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (fits(a, k, mid)) hi = mid;
        else lo = mid + 1;
    }
    std::cout << lo << '\n';
}
```

Sample input

```
6 3
2 1 3 1 1 4
```

Sample output

```
5
```

OLL tells `std::accumulate` to use a `long long` accumulator. The checker uses `value > capacity - used` rather than forming an unnecessary sum before its comparison. The initial lower bound already ensures that every single value fits.^[1-7]

With $S = \sum a_i$ and $M = \max a_i$, the search is $O(n \log(S - M + 1))$ for a nontrivial range, plus input. It stores $O(n)$ values and uses constant checker state. The all-zero case starts with $lo = hi = 0$ and returns immediately.

1.23 Maximize a minimum gap with an earliest-choice check

Choose k distinct positions from a collection of coordinates so that the minimum distance between consecutive chosen coordinates is as large as possible. Sort the coordinates first; unlike a fixed original-order subarray problem, this does not change which locations may be selected.

For a trial gap d , select the first coordinate, then repeatedly select the earliest remaining coordinate at least d beyond the last selected one. The predicate asks whether this can select at least k locations.

Why the earliest feasible choice works

Compare greedy's chosen coordinates $g_1, g_2, \dots$ with any feasible choices $p_1, p_2, \dots$. Initially $g_1 \leq p_1$. If $g_j \leq p_j$, then $p_{j+1} \geq p_j + d \geq g_j + d$, so p_{j+1} is available as a next greedy choice. The earliest available choice therefore satisfies $g_{j+1} \leq p_{j+1}$.

Thus greedy never uses up more coordinate space before its next choice than a feasible solution does. If any solution can choose k positions, greedy can also choose k .

Monotonicity and bounds

A feasible distance remains feasible when reduced. The predicate is true and then false, so use the last-true template. Gap 0 is feasible when $k \leq n$, and no minimum gap can exceed the full coordinate range.

For $[1, 2, 8, 12, 17]$ and $k = 3$, gap 7 allows 1, 8, 17. Gap 8 cannot place three positions. Sorting costs $O(n \log n)$, and checking the answer takes $O(n \log(D + 1))$ for coordinate range D . Input storage is $O(n)$.

Pause and predict: 24

Why does this task exclude $k = 1$ from its contract?

Check your prediction. With one chosen location there is no adjacent chosen pair whose minimum gap is being optimized. A different task could define a convention, but it must say what that convention is.

Implement the earliest-choice checker

Contract for Example 21. Read n, k and coordinates, with $2 \leq k \leq n \leq 200000$ and $|x_i| \leq 10^9$. Positions are distinct input entries; equal coordinates are allowed and may force answer 0. Print the largest feasible integer gap.

Example 21: Place items with a maximum minimum gap | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
bool canPlace(const std::vector<long long>& x, int k, long long gap) {
    int placed = 1;
    long long last = x[0];
    for (int i = 1; i < static_cast<int>(x.size()); ++i) {
        if (x[i] - last >= gap) {
            last = x[i];
            if (++placed >= k) return true;
        }
    }
    return placed >= k;
}
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<long long> x(n);
    for (auto& value : x) std::cin >> value;
    std::sort(x.begin(), x.end());
    long long lo = 0, hi = x.back() - x.front();
    while (lo < hi) {
        long long mid = lo + (hi - lo + 1) / 2;
        if (canPlace(x, k, mid)) lo = mid;
        else hi = mid - 1;
    }
    std::cout << lo << '\n';
}
```

Sample input

Sample output

5 3

7

1 2 8 12 17

1.24 Combine the techniques: the k th smallest pair distance

Guided synthesis. There are $n(n-1)/2$ position pairs. Each pair contributes one absolute difference. Find the k th smallest difference, counting repeated differences separately, without storing every pair.

A direct method builds all differences and sorts them. That costs quadratic storage before the sort even begins. Instead, choose a candidate distance d and ask:

$$C(d) = \#\{(i, j) : i < j, |a_i - a_j| \leq d\}, \quad C(d) \geq k?$$

Increasing d cannot remove any counted pair. The answer is the first distance for which the count reaches k .

Use a two-pointer window inside the checker

After sorting the values, fix a right endpoint. Move the left endpoint until $a[\text{right}] - a[\text{left}] \leq d$. Every index from left through $\text{right} - 1$ is a valid partner for right . Add $\text{right} - \text{left}$, not $\text{right} - \text{left} + 1$, because a position must not pair with itself.

Across one check, both endpoints move only forwards, so $C(d)$ is computed in $O(n)$ time. The checker must start with fresh `left` and `count` values for every trial distance: the binary-search candidates do not arrive in monotone order.

Trial d	$C(d)$ for $[1, 4, 4, 9]$	First-true meaning for $k = 4$
-----------	---------------------------	--------------------------------

0	1	Only the equal-valued pair qualifies.
3	3	Not enough pairs yet.
4	3	Still false.
5	5	True; the count jumps over 4.
8	6	True, but later than the required boundary.

The sorted difference list is $[0, 3, 3, 5, 5, 8]$. Its fourth entry is 5, exactly the first threshold with count at least 4.

Pause and predict: 25

Would searching for `countClose(a, d) == k` be correct?

Check your prediction. No. A count can jump over k , as the table shows. The correct monotone predicate is `countClose(a, d) >= k`.

One complete algorithm, two levels of invariants

Contract for Example 22. Read n, k and values, with $2 \leq n \leq 200000$, $|a_i| \leq 10^9$, and $1 \leq k \leq n(n-1)/2$. Print the k th smallest absolute difference across distinct position pairs.

Example 22: Binary search with a two-pointer counting check | Complete C++23 program

```
#include <algorithm>
#include <iostream>
#include <vector>
long long countClose(const std::vector<long long>& a, long long gap) {
    int left = 0;
    long long count = 0;
    for (int right = 0; right < static_cast<int>(a.size()); ++right) {
        while (a[right] - a[left] > gap) ++left;
        count += right - left;
    }
    return count;
}
int main() {
    int n;
    long long k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    long long lo = 0, hi = a.back() - a.front();
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (countClose(a, mid) >= k) hi = mid;
        else lo = mid + 1;
    }
    std::cout << lo << '\n';
}
```

Sample input

4 4
1 4 4 9

Sample output

5

The inner invariant identifies all valid partners for the current right endpoint and counts each pair once. The outer invariant keeps the first threshold whose count reaches k in the answer interval. Both proofs are necessary; neither one substitutes for the other.

With $D = \max a_i - \min a_i$, total time is $O(n \log n + n \log(D+1))$ including sorting, and the stored input occupies $O(n)$ memory. No list of pair distances is created. A `long long` count is needed because the number of pairs may exceed a typical 32-bit integer.

What this example should teach you to recognize

You do not always choose *between* binary search and two pointers. A hard optimization question can become a sequence of easy threshold questions. If those questions have a linear scan, binary search can call that scan repeatedly. Always count the full cost: number of thresholds multiplied by the cost of one check.

1.25 Optional extension: exact sums from two at-most counts

For nonnegative integer values, let $F(K)$ be the number of nonempty subarrays with sum at most K , and define $F(K) = 0$ for $K < 0$.

Every subarray counted by $F(K - 1)$ is also counted by $F(K)$. The difference keeps exactly the subarrays whose integer sum lies above $K - 1$ and at most K , which means their sum is exactly K :

$$\#\{\text{sum} = K\} = F(K) - F(K - 1).$$

This argument is about **nested sets of counted objects**, not about subtracting the lengths of two best windows. It also does not remove the nonnegative assumption required by our implementation of F .

Contract for Example 23. Read n, K and nonnegative values, with $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $0 \leq K \leq 2 \cdot 10^{14}$. Count nonempty subarrays with sum exactly K .

Example 23: Exact-sum counting with zeros | Complete C++23 program

```
#include <iostream>
#include <vector>
long long atMost(const std::vector<long long>& a, long long limit) {
    if (limit < 0) return 0;
    int left = 0;
    long long sum = 0, count = 0;
    for (int right = 0; right < static_cast<int>(a.size()); ++right) {
        sum += a[right];
        while (sum > limit) sum -= a[left++];
        count += right - left + 1;
    }
    return count;
}
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::cout << atMost(a, target) - atMost(a, target - 1) << '\n';
}
```

Sample input

```
5 1
0 1 0 2 0
```

Sample output

```
4
```

For the sample, the four sum-1 subarrays are $[0, 1]$, $[0, 1, 0]$, $[1]$, and $[1, 0]$. Running the at-most helper twice is still $O(n)$ time with $O(1)$ working memory beyond the array.

Pause and predict: 26

For target 0, why must the helper return immediately when called with limit -1 ?

Check your prediction. No nonnegative subarray can qualify. Trying to repair until a sum becomes at most -1 would not stop at the empty window, whose sum is 0, and could move outside the array.

1.26 Optional extension: bisect real-valued answers

An integer search stops when one candidate remains. A real-valued search cannot do that: between two different real values there are more candidates. Instead, shrink an interval until it is accurate enough, or use a fixed iteration count justified by the initial width and required precision.

Suppose ropes may be cut into equal positive real lengths x , discarding leftovers. A rope of length a_i yields $\lfloor a_i/x \rfloor$ pieces. Ask for the largest length that yields at least k pieces. This predicate is true for short lengths and false for long lengths, but the endpoint updates use `mid`, not `mid + 1` or `mid - 1`.

Zero is only a boundary representing the limiting feasible side; the checker never divides by zero. The conceptual interval width after s halvings is the initial width divided by 2^s . In actual floating-point arithmetic, the midpoint may eventually equal an endpoint; the explicit test stops further non-progressing iterations.

The checker caps the count before converting a potentially enormous quotient to an integer. The numerical answer remains an approximation, not an exact certificate at a discontinuity. The tests compare against exact rational optima on small inputs. Time is $O(90n)$ after reading, with $O(n)$ storage.

Pause and predict: 27

Why are the integer updates `lo = mid + 1` and `hi = mid - 1` wrong here?

Check your prediction. Adding or subtracting 1 skips a real interval that may contain the optimum. The discrete idea of an adjacent integer candidate does not apply to a real-valued domain.

Implement a bounded-precision search

Contract for Example 24. Read n, k and positive integer rope lengths, with $1 \leq n \leq 100000$, $1 \leq a_i \leq 10^6$, and $1 \leq k \leq 10^9$. Output the optimum real length with absolute error at most 10^{-6} .

Example 24: Maximum real piece length | Complete C++23 program

```
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <vector>
bool enough(const std::vector<long double>& ropes,
            long long needed, long double length) {
    long long made = 0;
    for (long double rope : ropes) {
        long double pieces = rope / length;
        if (pieces >= static_cast<long double>(needed - made)) return true;
        made += static_cast<long long>(pieces);
    }
    return made >= needed;
}
int main() {
    int n;
    long long k;
    std::cin >> n >> k;
    std::vector<long double> ropes(n);
    for (auto& value : ropes) std::cin >> value;
    long double lo = 0, hi = *std::max_element(ropes.begin(), ropes.end());
    for (int step = 0; step < 90; ++step) {
        long double mid = lo + (hi - lo) / 2;
        if (mid == lo || mid == hi) break;
        if (enough(ropes, k, mid)) lo = mid;
        else hi = mid;
    }
    std::cout << std::fixed << std::setprecision(10) << lo << '\n';
}
```

Sample input

4 5
8 7 5 4

Sample output

4.0

1.27 Choose a method, then test its entire argument

Before typing a familiar loop, answer four questions: Is original order essential? Is there sorted order or another monotone property? Are questions independent, or can neighbouring questions reuse progress? How much work does one update or one predicate evaluation perform?

For stored sorted values, start with a binary-search boundary for an independent query. For a target pair, opposite-end pointers can reuse eliminations. For an independent batch of ordered thresholds, a sweep can reuse the previous answer boundary.

For original-order segments, a fixed-length rolling sum works even with signed values. A variable sum window needs the stated monotonicity, here supplied by nonnegative values. Do not sort away the original adjacency requirement.

For an optimized numerical answer, write a yes/no checker first. Prove whether it is false-then-true or true-then-false, derive valid bounds, then select the matching search template. A checker may itself be a two-pointer scan, as in the pair-distance example.

Protect the computation as well as the idea

With $n \leq 200000$ and $|a_i| \leq 10^9$, whole-array totals can reach $2 \cdot 10^{14}$. Pair and subarray counts can exceed $2 \cdot 10^{10}$. Store those values in `long long`; make every intermediate expression safe too. In this chapter, midpoint spans fit `long long`. Do not extend the same formulas to arbitrary signed endpoints without rechecking subtraction and addition.

Never read `*end()`, decrement a beginning iterator, or access a self-pair to make a condition convenient. A missing answer, an empty array, and a zero count are legitimate outcomes when the contract permits them.

Test a small instance three ways

List the correct answers by hand. Run a direct implementation on small inputs. Compare the optimized program against that independent result. Include an absent target, all equal values, all zeros, an extreme threshold, and an answer at each end of the search domain.

For multiple test cases, create fresh input and state inside `solve()`. A binary-search checker also needs fresh state on each call. Bound total work across all cases rather than analysing only one small case.

Pause and predict: 28

A checker scans n values, and the search tries $O(\log R)$ thresholds. Can the full search be described as $O(\log R)$?

Check your prediction. No. Its checking work is $O(n \log R)$. Add any sorting, preprocessing, input, and output costs that the complete program performs.

1.28 Mixed practice: decide which boundary the task needs

These problems belong to one practice set. The first group checks precise indices and scanning; the middle group develops windows and lookup; the final group asks you to build a checker or combine techniques. Do not assume that a problem needs binary search just because it appears late in the session.

For every attempt, write the input assumption, the state meaning, one safe update, and the total time and storage. Use the separate hints only after a genuine attempt. Practice 18 to 20 are a second-reading route. Examples and solutions assume a training environment with a sufficiently wide int for the stated sizes and at least a 64-bit long long.

Practice 01: A mirror sequence

Decide whether a sequence reads identically from left to right and from right to left. Compare the original order; do not sort it.

Input and bounds. Read n , then n integers. Use $0 \leq n \leq 200000$ and $|a_i| \leq 10^9$.

Output. Print YES or NO. An empty sequence is a palindrome.

Sample input	Sample output
5 4 8 2 8 4	YES

Hint: Section 1.30. Explained answer: Solution 01.

Practice 02: Two boundary answers

For each query x on a sorted sequence, find the first index whose value is at least x and the last index whose value is at most x . These are inequality boundaries, not necessarily occurrences of x .

Input and bounds. Read n, q , then n nondecreasing values and q query values. Use $0 \leq n, q \leq 200000$, $|a_i| \leq 10^9$, and $|x| \leq 2 \cdot 10^9$.

Output. For each query print two zero-based indices. Use n when no value is at least x , and -1 when no value is at most x .

Sample input	Sample output
5 3 2 4 4 7 9 4 1 10	1 2 0 -1 5 4

Hint: Section 1.30. Explained answer: Solution 02.

Practice 03: Return the original pair

Find two different original positions whose values sum to the target. The input is not sorted, and the output must identify positions in that original input.

Input and bounds. Read n, T and the values. Use $0 \leq n \leq 200000$, $|a_i| \leq 10^9$, and $|T| \leq 2 \cdot 10^9$.

Output. Print any valid one-based pair $i < j$, or print -1 when no pair exists.

Sample input

5 11
8 1 6 3 9

Sample output

1 4

Hint: Section 1.30. Explained answer: Solution 03.

Practice 04: Values inside a closed interval

Answer how many stored values lie between two query limits, inclusive. This is a range of values, not a range of original positions.

Input and bounds. Read n, q , the arbitrary-order values, then q pairs L, R . Use $0 \leq n, q \leq 200000$, $|a_i| \leq 10^9$, and $-2 \cdot 10^9 \leq L \leq R \leq 2 \cdot 10^9$.

Output. Print one occurrence count per query. Duplicate values count separately.

Sample input

5 3
8 2 4 4 10
4 8
0 1
2 10

Sample output

3
0
5

Hint: Section 1.30. Explained answer: Solution 04.

Practice 05: Pair totals between two limits

Count position pairs $i < j$ whose total lies in the closed interval $[L, R]$. Values may be negative, and equal-valued positions are still different choices.

Input and bounds. Read n, L, R and an arbitrary-order sequence, with $0 \leq n \leq 200000$, $|a_i| \leq 10^9$, and $-2 \cdot 10^9 \leq L \leq R \leq 2 \cdot 10^9$.

Output. Print the number of qualifying pairs.

Sample input	Sample output
5 5 8 1 3 3 5 7	5

Hint: Section 1.30. Explained answer: Solution 05.

Practice 06: Keep the common copies

Intersect two sorted sequences while retaining duplicate copies. A value occurring u times in one input and v times in the other should occur $\min(u, v)$ times in the output.

Input and bounds. Read n, m , then both nondecreasing arrays. Use $0 \leq n, m, n + m \leq 200000$, and absolute values at most 10^9 .

Output. Print the output size, then its sorted values on one line. Print EMPTY on that second line when the intersection is empty.

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

Hint: Section 1.30. Explained answer: Solution 06.

Practice 07: Move zeros without reordering other values

Move every zero to the end of a sequence. Keep the relative order of all nonzero values, including negative values. The final sequence must have the original length.

Input and bounds. Read n and the values, with $0 \leq n \leq 200000$ and $|a_i| \leq 10^9$.

Output. Print the resulting sequence, or EMPTY for $n = 0$.

Sample input**Sample output**

6
0 4 -2 0 7 0

4 -2 7 0 0 0

Hint: Section 1.30. Explained answer: Solution 07.

Practice 08: The closest values from two lists

Choose one value from each of two nonempty lists and minimize their absolute difference. Return the minimum difference, not their positions.

Input and bounds. Read n, m , then both arrays, with $1 \leq n, m \leq 200000$ and absolute values at most 10^9 . The arrays may be unsorted.

Output. Print the smallest absolute difference. A shared value makes the answer 0.

Sample input**Sample output**

3 3
12 3 20
9 18 5

2

Hint: Section 1.30. Explained answer: Solution 08.

Practice 09: The cheapest fixed-length segment

Among all subarrays of length k , find the smallest sum. If several have the same sum, choose the earliest starting position.

Input and bounds. Read n, k and the values, with $1 \leq k \leq n \leq 200000$ and $|a_i| \leq 10^9$.

Output. Print the minimum sum and its one-based starting position.

Sample input**Sample output**

5 2

-1 2

4 -2 1 -1 3

Hint: Section 1.30. Explained answer: Solution 09.

Practice 10: Repair at most k zeros

In a binary sequence, you may change at most k zeros to ones. Find the longest original contiguous segment that can become all ones.

Input and bounds. Read n, k and n binary values, with $0 \leq n \leq 200000$ and $0 \leq k \leq n$.

Output. Print the maximum possible segment length; 0 is allowed.

Sample input**Sample output**

7 1

5

1 0 1 1 0 1 1

Hint: Section 1.30. Explained answer: Solution 10.

Practice 11: Return a shortest target segment

Find a shortest nonempty subarray with sum at least T . If several shortest subarrays exist, choose the one with the earliest start. Return its endpoints as well as its length.

Input and bounds. Read n, T and values, with $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $1 \leq T \leq 2 \cdot 10^{14}$.

Output. Print length, one-based left endpoint, and one-based right endpoint. Print only -1 if impossible.

Sample input

6 7
2 1 4 2 5 1

Sample output

2 4 5

Hint: Section 1.30. Explained answer: Solution 11.

Practice 12: Count substrings with few distinct letters

Count nonempty substrings containing at most k distinct letters. Different positions count as different substrings even when their text is equal.

Input and bounds. Read k and a nonempty lowercase ASCII word of length at most 200000, with $0 \leq k \leq 26$.

Output. Print the number of qualifying substrings.

Sample input

2 abac

Sample output

8

Hint: Section 1.30. Explained answer: Solution 12.

Practice 13: The nearest available value

For every query value, return a stored value with minimum absolute distance. If two different values are equally close, return the smaller one.

Input and bounds. Read n, q , the arbitrary-order values, then q queries. Use $1 \leq n \leq 200000$, $0 \leq q \leq 200000$, $|a_i| \leq 10^9$, and $|x| \leq 2 \cdot 10^9$.

Output. Print the selected value for each query, not its index.

Sample input	Sample output
4 3	2
10 2 6 6	10
4	10
9	
15	

Hint: Section 1.30. Explained answer: Solution 13.

Practice 14: The first day that reaches a total

Each day adds a nonnegative number of packages to a cumulative total. For each positive target, find the first day at which the total reaches that target.

Input and bounds. Read n, q , then n daily counts and q targets. Use $0 \leq n, q \leq 200000$, daily counts in $[0, 10^9]$, and targets in $[1, 2 \cdot 10^{14}]$.

Output. Print a one-based day number for each target, or -1 if the target is never reached.

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

Hint: Section 1.30. Explained answer: Solution 14.

Practice 15: Minimum integer processing speed

A worker processes one positive job at a time. At speed s , a job of size a_i takes $\lceil a_i/s \rceil$ whole hours. Unused capacity in its final hour cannot be shared with another job. Find the smallest positive integer speed that finishes within H hours.

Input and bounds. Read n, H and positive job sizes, with $1 \leq n \leq 200000$, $1 \leq a_i \leq 10^9$, and $0 \leq H \leq 2 \cdot 10^{14}$.

Output. Print the minimum speed, or -1 when impossible.

Sample input	Sample output
3 7	4
7 11 6	

Hint: Section 1.30. Explained answer: Solution 15.

Practice 16: Exactly k consecutive groups

Split a nonnegative sequence into exactly k nonempty consecutive groups. Minimize the largest group sum. Keep the original order.

Input and bounds. Read n, k and values, with $1 \leq k \leq n \leq 200000$ and $0 \leq a_i \leq 10^9$.

Output. Print the minimum achievable largest group sum.

Sample input	Sample output
5 3	5
3 1 4 1 5	

Hint: Section 1.30. Explained answer: Solution 16.

Practice 17: Whole-number rope pieces

Cut ropes into pieces of one common positive integer length, discarding leftovers. Find the largest length that yields at least k pieces. Pieces from different ropes cannot be joined.

Input and bounds. Read n, k and rope lengths, with $1 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $1 \leq k \leq 10^9$.

Output. Print the largest feasible integer length, or 0 if no positive length works.

Sample input	Sample output
3 7 8 5 6	2

Hint: Section 1.30. Explained answer: Solution 17.

Practice 18: Exactly a target sum (extension)

Count nonempty subarrays whose sum equals the nonnegative integer target K . Zeros are allowed, so several different subarrays may end at the same position with the same sum.

Input and bounds. Read n, K and values, with $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $0 \leq K \leq 2 \cdot 10^{14}$.

Output. Print the number of qualifying position intervals.

Sample input	Sample output
4 0 0 0 1 0	4

Hint: Section 1.30. Explained answer: Solution 18.

Practice 19: The kth smallest pair total (synthesis)

Form the sum of every pair of different positions. Sort these sums conceptually, retaining repeated sums. Find the one-based kth entry without explicitly storing all pairs.

Input and bounds. Read n, k and values, with $2 \leq n \leq 200000$, $|a_i| \leq 10^9$, and $1 \leq k \leq n(n-1)/2$.

Output. Print the kth smallest pair sum. It may be negative.

Sample input
Sample output

4 4

10

1 4 4 9

 Hint: Section 1.30. Explained answer: Solution 19.
Practice 20: Search the window length (comparison challenge)

Find the longest subarray of nonnegative values with sum at most B , but implement it by binary-searching the length. Afterwards explain why the ordinary variable window is faster for this exact task.

Input and bounds. Read n, B and values, with $0 \leq n \leq 200000$, $0 \leq a_i \leq 10^9$, and $0 \leq B \leq 2 \cdot 10^{14}$.

Output. Print the greatest feasible length, allowing 0.

Sample input
Sample output

6 6

4

2 1 3 1 1 4

 Hint: Section 1.30. Explained answer: Solution 20.

1.29 Mixed drills: trace, justify, and compare

Write the intermediate states and the reason for each change, not only the final answer. These drills deliberately mix both techniques. Hints are in Section 1.30; worked answers are in Section 1.32.

Drill 1: Discard a value or discard a half?

Use the sorted array $[2, 4, 4, 7, 9, 12]$ and target total 16.

Trace the opposite-end pair scan. Record $(left, right)$, the tested total, and the reason for each move. Then fix the value 4 at index 1 and describe the half-open range a binary search should inspect for its partner. Explain why searching the whole array is an unsafe general habit, even though this particular partner is different from 4. Your trace or argument:

Drill 2: Boundary indices are not necessarily matches

For $a = [2, 4, 4, 7, 9, 12]$, give the lower-bound index, upper-bound index, and occurrence count for each of $x = 4$, $x = 6$, and $x = 20$.

Then answer how many stored occurrences lie in the value interval $[4, 9]$. Explain which of your indices may legally equal n , and why that does not give an element that can be read. Your trace or argument:

Drill 3: Find the non-progressing loop

This deliberately wrong fragment is meant to find the first value at least x in a sorted vector:

```
int lo = 0, hi = n;
while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] < x) lo = mid; // wrong update
    else hi = mid;
}
```

Give a one-element input on which it does not terminate. Repair only the incorrect update and justify both safety and progress. Your trace or argument:

Drill 4: Which input promise is doing the work?

A sorted pair-sum scan is used on $[-5, -2, 3, 8]$ with target 6. A longest-budget window is used on $[5, -4]$ with budget 1.

Does the presence of negative values invalidate both methods, only one, or neither? Give the valid pair, trace the failing window, and state whether sorting the second input would be an acceptable general repair for an original-order subarray task. Your trace or argument:

Drill 5: Offline queries and saved output order

The sorted values are $[2, 4, 4, 7, 9, 12]$. In the original query order, the thresholds are $[6, 1, 4, 12]$. Each answer is the number of values at most the threshold.

Show the queries after sorting with original IDs, the scan boundary after each, and the answers restored to original order. Compare the complete cost with doing an upper-bound search for each query, assuming the values are already sorted. Your trace or argument:

Drill 6: Expand, repair, and measure

Run the longest-budget algorithm on $[2, 1, 3, 1, 1, 4]$ with budget 6. For every right endpoint record the sum just after insertion, every removed value, the repaired left endpoint, and the repaired window length.

Give the final maximum length. Explain why a single if statement replacing the shrinking while loop can leave this particular algorithm with an invalid state. Your trace or argument:

Drill 7: A shortest window has a different measurement point

For nonnegative values $[2, 1, 3]$ and target 6, trace the shortest-target algorithm. Should it record the window length before removing the leftmost value, or after?

Explain why the condition in this algorithm removes values while the window is valid, unlike the longest-budget algorithm. What changes if the problem permits target 0 but still requires a nonempty answer? Your trace or argument:

Drill 8: Count starts, not just one best interval

For $[0, 0, 1]$, list the nonempty subarrays of sum at most 0 and those of sum exactly 1. Give their counts.

A repaired at-most-sum window adds $\text{right-left}+1$, whereas the pair-distance checker adds right-left . Explain the extra one. Why is subtracting two longest-window lengths not a substitute for subtracting two at-most counts? Your trace or argument:

Drill 9: A window and a prefix search solve the same task

Let $a = [2, 1, 3, 1, 1, 4]$ and let $P[e]$ be the sum of the first e elements, so $P = [0, 2, 3, 6, 7, 8, 12]$.

For a segment starting at index 1 and target 6, derive the required prefix threshold. Find its earliest exclusive end, its length, and its one-based inclusive endpoints. Explain why the prefix search costs more than reusing a moving endpoint across all starts. Your trace or argument:

Drill 10: First true, last true, and the midpoint

For candidate integers 0 through 7, predicate A has truth values F,F,F,F,T,T,T,T. Predicate B has values T,T,T,T,F,F,F,F.

Which boundary is needed to minimize A and maximize B? Trace the first iteration of each inclusive-interval template. Then use $lo=2$, $hi=3$ to explain why retaining a successful lower midpoint can prevent a last-true search from progressing. Your trace or argument:

Drill 11: Prove the checker, not only its monotonicity

For $[2, 1, 3, 1, 1, 4]$, compute the minimum number of consecutive groups needed at capacities 4 and 5. A group sum cannot exceed the capacity.

Give the minimum largest sum for at most three groups. Explain the exchange argument behind greedily extending the current group, and why the same feasibility test also works when exactly three nonempty groups are required. Your trace or argument:

Drill 12: An outer search calls an inner scan

For sorted values $[1, 4, 4, 9]$, list all six pair distances with multiplicities. Find the fourth smallest.

Give the counts of distances at most 3, 4, and 5. Explain why searching for `count==4` is wrong, why the counting scan must reset its state on every predicate call, and the full cost for n values with coordinate range D . Your trace or argument:

1.30 Hints before solutions

These hints point to the missing modelling decision without supplying a full implementation. After using one, return to your own code and try a small trace before opening the solution.

Coding problems

Practice 01: A mirror sequence

Compare symmetric positions and stop at the first mismatch. A middle element needs no partner.

Practice 02: Two boundary answers

The first answer is lower bound. The second is the upper-bound index minus one. Neither boundary needs to be dereferenced.

Practice 03: Return the original pair

Sort pairs of value and saved one-based ID. Make pointer decisions using values, but return the stored IDs.

Practice 04: Values inside a closed interval

Sort once. Subtract the number of values below L from the number at most R .

Practice 05: Pair totals between two limits

Let $G(X)$ count pairs with sum at most X . Derive $G(R) - G(L - 1)$, then implement G using the sorted pair-count scan.

Practice 06: Keep the common copies

Advance the side containing the smaller value. On equality, output one copy and advance both sides.

Coding hints continued

Practice 07: Move zeros without reordering other values

Use a read index and a write index. Copy nonzero values forward in their encounter order, then fill the unused suffix with zeros.

Practice 08: The closest values from two lists

Sort both arrays. Compare their current values and advance the side containing the smaller value.

Practice 09: The cheapest fixed-length segment

Build the first window. Slide it by one position at a time, updating the best result only for a strictly smaller sum.

Practice 10: Repair at most k zeros

A window is feasible exactly when its zero count is at most k . Maintain that count instead of changing the array.

Practice 11: Return a shortest target segment

Use the shortest-window loop, but store a complete answer whenever you measure a valid window. Compare length first, then start.

Practice 12: Count substrings with few distinct letters

Repair a frequency window as in Example 15. After repair, every suffix ending at right and starting no earlier than left is valid.

Practice 13: The nearest available value

Lower bound divides values below the query from values at least the query. Only the value at that boundary and its predecessor can win.

Coding hints continued

Practice 14: The first day that reaches a total

Build the prefix array with a leading zero. Search boundaries representing real days, not the initial day-zero total.

Practice 15: Minimum integer processing speed

Every positive job takes at least one hour, so check $H < n$. For feasible inputs, search speeds from 1 to the largest job size.

Practice 16: Exactly k consecutive groups

The greedy checker finds whether at most k groups suffice. Explain why a solution using fewer groups can be split to obtain exactly k without increasing the maximum sum.

Practice 17: Whole-number rope pieces

For a positive trial length, count $\lfloor a_i / \text{length} \rfloor$ pieces. Search for the last success, keeping zero only as a fallback boundary and never dividing by it.

Practice 18: Exactly a target sum (extension)

Use the nested counts $F(K)$ and $F(K-1)$, not the difference between two longest lengths. Handle a negative limit before entering the window loop.

Practice 19: The kth smallest pair total (synthesis)

For a trial total X, count pairs whose sum is at most X using two pointers. Search for the first X whose count is at least k.

Practice 20: Search the window length (comparison challenge)

For a fixed trial length, scan all windows using a rolling sum. A feasible nonempty window contains feasible shorter windows because the values are nonnegative.

Hints for the mixed drills

Drill 1: Discard a value or discard a half?

The endpoint scan begins with total 14. The fixed-index search must use only positions strictly after the chosen index.

Drill 2: Boundary indices are not necessarily matches

Lower bound counts values below x ; upper bound counts values at most x . A boundary may lie after the last element.

Drill 3: Find the non-progressing loop

With $lo=0$ and $hi=1$, the lower midpoint is 0. Decide whether the tested middle position is still a candidate after $a[mid] < x$.

Drill 4: Which input promise is doing the work?

Sorted sums change predictably when an endpoint moves even with negative values. Removing a negative value from a window can increase its sum.

Drill 5: Offline queries and saved output order

Use zero-based query IDs. Sort the pairs $(threshold, ID)$, but store each answer in its saved ID slot.

Drill 6: Expand, repair, and measure

At the last right endpoint, the sum becomes 10 and more than one value must be removed.

Drill hints continued**Drill 7: A shortest window has a different measurement point**

The first valid window has length 3 and sum 6. Removing its first element changes the sum to 4.

Drill 8: Count starts, not just one best interval

Two zeros at different positions give three nonempty zero-sum intervals. A pair must not use the same position twice.

Drill 9: A window and a prefix search solve the same task

The sum of $[start, end)$ is $P[end] - P[start]$. Search only exclusive ends after start for a nonempty segment.

Drill 10: First true, last true, and the midpoint

First true keeps a true mid as the upper endpoint; last true keeps a true mid as the lower endpoint. The latter needs an upper midpoint.

Drill 11: Prove the checker, not only its monotonicity

At capacity 4 the greedy blocks end after positions 1, 3, 4, and 5. At capacity 5 the middle group can contain 3,1,1.

Drill 12: An outer search calls an inner scan

At distance 5, the count jumps from 3 to 5. Binary-search trial distances need not arrive in increasing order.

1.31 Explained practice solutions

Read a solution by separating its model, its implementation, and its correctness argument. A template is useful only when you can explain why the input satisfies its assumptions. The complete programs below are also supplied as individual source files.

Solution 01: A mirror sequence

Idea. Place one index at each end. While they identify different positions and their values match, move both inward. At exit, crossed or equal indices mean that every required pair matched.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int n;
    std::cin >> n;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int left = 0, right = n - 1;
    while (left < right && a[left] == a[right]) {
        ++left;
        --right;
    }
    std::cout << (left >= right ? "YES" : "NO") << '\n';
}
```

Why it works. Before every iteration, all symmetric pairs outside the current interval have matched. A mismatch inside the interval is enough to reject the whole sequence. If no mismatch is found before the indices meet, every symmetric pair has been checked, so the sequence is a palindrome.

Complexity. The scan is $O(n)$ time with $O(1)$ working memory beyond the $O(n)$ input vector.

Boundary checks. Include $n = 0$, $n = 1$, an even-length palindrome, and a mismatch at the outermost and innermost pair.

Solution 02: Two boundary answers

Idea. Use lower bound to count the values strictly below x . Use upper bound to count the values at most x , then subtract one to obtain the last such index.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    while (q-- > 0) {
        long long x;
        std::cin >> x;
        int first = static_cast<int>(
            std::lower_bound(a.begin(), a.end(), x) - a.begin());
        int last = static_cast<int>(
            std::upper_bound(a.begin(), a.end(), x) - a.begin()) - 1;
        std::cout << first << ' ' << last << '\n';
    }
}
```

Why it works. Sorted order places every value below x before its lower bound, so that boundary is the first acceptable index. Upper bound lies immediately after every value at most x , so its preceding numeric index is the requested last position. If there is no such value, the count is zero and the resulting index is -1 .

Complexity. $O(n + q \log(n + 1))$ time including input; $O(n)$ stored input and constant working state per query.

Boundary checks. Use a missing value between two elements, all-equal values, thresholds below and above the whole array, and an empty array.

Solution 03: Return the original pair

Idea. Attach an original ID to each value before sorting. Search with opposite-end pointers. On a match, order the two saved IDs increasingly for output.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<std::pair<long long, int>> a(n);
    for (int i = 0; i < n; ++i) {
        std::cin >> a[i].first;
        a[i].second = i + 1;
    }
    std::sort(a.begin(), a.end());
    int left = 0, right = n - 1;
    while (left < right) {
        long long sum = a[left].first + a[right].first;
        if (sum == target) {
            int i = a[left].second, j = a[right].second;
            std::cout << std::min(i, j) << ' ' << std::max(i, j) << '\n';
            return 0;
        }
        if (sum < target) ++left;
        else --right;
    }
    std::cout << "-1\n";
}
```

Why it works. Sorting the records preserves every input item and its ID. The endpoint elimination proof applies to the sorted values, so a solution is found exactly when a value pair exists. Distinct sorted positions contain distinct records and therefore distinct original IDs, even if their values are equal.

Complexity. $O(n \log n)$ time including sorting and $O(n)$ record storage. The pointer scan itself is $O(n)$ time with constant state.

Boundary checks. Test a single value that would match itself, two equal values at different positions, negative values, and a pair whose original IDs are far apart.

Solution 04: Values inside a closed interval

Idea. In the sorted array, the qualifying block starts at lower bound of L and ends just before upper bound of R . Its length is the difference of those iterators.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    while (q-- > 0) {
        long long left, right;
        std::cin >> left >> right;
        auto first = std::lower_bound(a.begin(), a.end(), left);
        auto after = std::upper_bound(a.begin(), a.end(), right);
        std::cout << after - first << '\n';
    }
}
```

Why it works. Every earlier value is below L and every later value is above R . All values between the two boundaries satisfy both inequalities. Therefore the boundary difference counts exactly the requested occurrences, including duplicates and the endpoints.

Complexity. $O(n \log n + q \log(n + 1))$ time and $O(n)$ input storage, with constant per-query state.

Boundary checks. Use $L = R$, an empty qualifying block, duplicate endpoint values, an empty array, and a query covering all stored values.

Solution 05: Pair totals between two limits

Idea. Sort once and run the at-most pair counter twice. The first count includes every requested pair and all pairs below the lower limit; the second removes exactly those below the lower limit.

Complete solution | C++23

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

long long countPairs(const std::vector<long long>& a, long long limit) {
    int left = 0, right = static_cast<int>(a.size()) - 1;
    long long count = 0;
    while (left < right) {
        if (a[left] + a[right] <= limit) {
            count += right - left;
            ++left;
        } else --right;
    }
    return count;
}

int main() {
    int n;
    long long low, high;
    std::cin >> n >> low >> high;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    std::cout << countPairs(a, high) - countPairs(a, low - 1) << '\n';
}
```

Why it works. Pair sums are integers. A pair counted by $G(R)$ but not $G(L - 1)$ has sum at least L and at most R . The sorted counter partitions pairs by their left endpoint and counts each exactly once. Its proof uses sorted order, so negative values are allowed.

Complexity. $O(n \log n)$ time including sorting, two $O(n)$ scans, and $O(n)$ input storage. The count is long long.

Boundary checks. Include $L = R$, all negative values, all equal values, $n < 2$, and limits outside the possible sum range. The printed bounds make $L - 1$ safe.

Solution 06: Keep the common copies

Idea. Scan the two unread heads. A smaller value cannot match the other head or any later value on that side. Equal heads form one available matching copy.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int n, m;
    std::cin >> n >> m;
    std::vector<long long> a(n), b(m), common;
    for (auto& value : a) std::cin >> value;
    for (auto& value : b) std::cin >> value;
    int i = 0, j = 0;
    while (i < n && j < m) {
        if (a[i] < b[j]) ++i;
        else if (a[i] > b[j]) ++j;
        else {
            common.push_back(a[i]);
            ++i;
            ++j;
        }
    }
    std::cout << common.size() << '\n';
    if (common.empty()) std::cout << "EMPTY";
    for (int k = 0; k < static_cast<int>(common.size()); ++k) {
        if (k > 0) std::cout << ' ';
        std::cout << common[k];
    }
    std::cout << '\n';
}
```

Why it works. Discarding a smaller head cannot lose a future match because the other unread sequence is sorted and starts larger. Matching equal heads consumes one copy from each input, so repeated equal values can be matched exactly until the smaller multiplicity is exhausted. The output is therefore the required multiset intersection.

Complexity. $O(n + m)$ time; $O(n + m)$ input storage and $O(\min(n, m))$ possible output storage. Pointer state is constant.

Boundary checks. Use one empty sequence, no shared values, unequal numbers of repeated copies, and inputs containing only the same value.

Solution 07: Move zeros without reordering other values

Idea. The accepted prefix holds all nonzero values seen so far. A nonzero value is copied to the next free position. When the scan finishes, fill every remaining position with zero.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int n;
    std::cin >> n;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int written = 0;
    for (int read = 0; read < n; ++read) {
        if (a[read] != 0) a[written++] = a[read];
    }
    while (written < n) a[written++] = 0;
    if (n == 0) std::cout << "EMPTY";
    for (int i = 0; i < n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << a[i];
    }
    std::cout << '\n';
}
```

Why it works. At each step the write index is no farther right than the read index, so writing cannot destroy a future unread value. Values are accepted in input order, which preserves stability. The number of unfilled positions equals the number of skipped zeros, so filling the suffix restores the correct multiplicity and length.

Complexity. $O(n)$ time and $O(1)$ working memory beyond the original vector.

Boundary checks. Use all zeros, no zeros, zeros at both ends, and alternating zero and negative values. Sorting would fail the relative-order requirement.

Solution 08: The closest values from two lists

Idea. Keep the smallest difference seen at the current pair. If $a[i] < b[j]$, advancing j cannot make this $a[i]$ closer, so move i . Otherwise move j .

Complete solution | C++23

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>
int main() {
    int n, m;
    std::cin >> n >> m;
    std::vector<long long> a(n), b(m);
    for (auto& value : a) std::cin >> value;
    for (auto& value : b) std::cin >> value;
    std::sort(a.begin(), a.end());
    std::sort(b.begin(), b.end());
    int i = 0, j = 0;
    long long best = std::abs(a[0] - b[0]);
    while (i < n && j < m) {
        best = std::min(best, std::abs(a[i] - b[j]));
        if (a[i] < b[j]) ++i;
        else ++j;
    }
    std::cout << best << '\n';
}
```

Why it works. When $a[i] < b[j]$, every later value in the second sorted list is at least $b[j]$, and therefore cannot improve the difference for this fixed $a[i]$. The symmetric statement holds when $b[j] < a[i]$. Equality already attains the smallest possible difference. Discarding the smaller head after recording the current difference cannot lose a better answer.

Complexity. $O(n \log n + m \log m + n + m)$ time including sorting and scanning. The two input arrays occupy $O(n + m)$ storage.

Boundary checks. Include negative values, a shared value, duplicates, one-element lists, and lists with completely separated ranges.

Solution 09: The cheapest fixed-length segment

Idea. Initialize both the best total and its position from the first actual window. For later windows, replace the outgoing contribution with the incoming one. Keep an earlier position on a tie.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    long long sum = 0;
    for (int i = 0; i < k; ++i) sum += a[i];
    long long best = sum;
    int start = 0;
    for (int right = k; right < n; ++right) {
        sum += a[right] - a[right - k];
        if (sum < best) {
            best = sum;
            start = right - k + 1;
        }
    }
    std::cout << best << ' ' << start + 1 << '\n';
}
```

Why it works. The maintained sum equals the current length- k window after every update. Every possible starting position is visited exactly once, in increasing order. Replacing the answer only on strict improvement therefore returns the minimum sum and the earliest start among tied minima.

Complexity. $O(n)$ time and $O(1)$ working memory beyond the input. The sum is long long.

Boundary checks. Include $k = 1$, $k = n$, all-negative values, and several identical minimum sums.

Solution 10: Repair at most k zeros

Idea. Append each bit, update the zero count, and remove leftmost bits while too many zeros remain. Measure the repaired window.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<int> a(n);
    for (int& value : a) std::cin >> value;
    int left = 0, zeros = 0, best = 0;
    for (int right = 0; right < n; ++right) {
        if (a[right] == 0) ++zeros;
        while (zeros > k) {
            if (a[left] == 0) --zeros;
            ++left;
        }
        best = std::max(best, right - left + 1);
    }
    std::cout << best << '\n';
}
```

Why it works. A segment needs one change for each zero, so the zero-count condition is equivalent to the task. Extending cannot reduce this count, and removing elements cannot increase it. The repaired window is the longest feasible window for its ending position, so the largest recorded length is optimal.

Complexity. $O(n)$ time and $O(1)$ working memory beyond the input.

Boundary checks. Use $k = 0$, an all-zero sequence, $k = n$, an empty sequence, and a case where one expansion requires multiple removals.

Solution 11: Return a shortest target segment

Idea. Whenever the current sum reaches the target, evaluate the window before removing its leftmost value. Replace the stored triple for a shorter window or for an equal length with an earlier start.

Complete solution | C++23

```
#include <iostream>
#include <vector>
int main() {
    int n;
    long long target;
    std::cin >> n >> target;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int left = 0, best = n + 1, start = -1, finish = -1;
    long long sum = 0;
    for (int right = 0; right < n; ++right) {
        sum += a[right];
        while (sum >= target) {
            int length = right - left + 1;
            if (length < best || (length == best && left < start)) {
                best = length;
                start = left;
                finish = right;
            }
            sum -= a[left++];
        }
    }
    if (start == -1) std::cout << "-1\n";
    else std::cout << best << ' ' << start + 1 << ' ' << finish + 1 << '\n';
}
```

Why it works. A removed start has already produced a measured feasible window, and any later ending position with that start would be longer. If the sum is too small, deleting nonnegative values cannot improve feasibility. Thus a shortest useful candidate is considered before each start is discarded, and the explicit tie comparison selects the required earliest one.

Complexity. $O(n)$ time and $O(1)$ working state beyond the array.

Boundary checks. Use zeros at the window boundaries, a single sufficient value, no sufficient subarray, and equally short answers at different positions.

Solution 12: Count substrings with few distinct letters

Idea. Keep 26 frequencies and the number of positive entries. Repair until at most k letters remain, then add the number of valid starting positions, not just one.

Complete solution | C++23

```
#include <array>
#include <iostream>
#include <string>
int main() {
    int k;
    std::string text;
    std::cin >> k >> text;
    std::array<int, 26> frequency{};
    int left = 0, distinct = 0;
    long long count = 0;
    for (int right = 0; right < static_cast<int>(text.size()); ++right) {
        if (++frequency[text[right] - 'a'] == 1) ++distinct;
        while (distinct > k) {
            if (--frequency[text[left] - 'a'] == 0) --distinct;
            ++left;
        }
        count += right - left + 1;
    }
    std::cout << count << '\n';
}
```

Why it works. Removing characters cannot increase the distinct-letter count. After repair, every start from left to right yields a valid suffix, while every earlier start has been shown invalid. Their count is $\text{right} - \text{left} + 1$. Grouping by the right endpoint counts each substring exactly once.

Complexity. $O(n)$ time and $O(26)$ working memory beyond the string. Use long long for the answer.

Boundary checks. Include $k = 0$, one repeated letter, a word with all different letters, and a long repeated word whose substring count exceeds a 32-bit integer.

Solution 13: The nearest available value

Idea. Sort the values once. Handle a boundary at the beginning or end separately. Otherwise compare the distances to the immediate predecessor and successor; use a non-strict comparison to prefer the smaller value on a tie.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    while (q-- > 0) {
        long long x;
        std::cin >> x;
        auto it = std::lower_bound(a.begin(), a.end(), x);
        long long answer;
        if (it == a.begin()) answer = *it;
        else if (it == a.end()) answer = a.back();
        else {
            long long below = *(it - 1), above = *it;
            answer = (x - below <= above - x ? below : above);
        }
        std::cout << answer << '\n';
    }
}
```

Why it works. Every value farther left is no closer than the greatest value below the query, and every value farther right is no closer than the smallest value at least the query. Hence one of the two neighbouring candidates is optimal. The boundary guards handle queries outside the stored range without invalid iterator access.

Complexity. $O(n \log n + q \log n)$ time and $O(n)$ storage. Distances are evaluated in long long.

Boundary checks. Use a query smaller than all values, larger than all values, an exact match, duplicated values, and an equal-distance tie.

Solution 14: The first day that reaches a total

Idea. Prefix entry i is the total after exactly i days. Search the nondecreasing entries from index 1 through n for the first one at least the target.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n, q;
    std::cin >> n >> q;
    std::vector<long long> prefix(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long value;
        std::cin >> value;
        prefix[i] = prefix[i - 1] + value;
    }
    while (q-- > 0) {
        long long target;
        std::cin >> target;
        auto it = std::lower_bound(prefix.begin() + 1, prefix.end(), target);
        if (it == prefix.end()) std::cout << "-1\n";
        else std::cout << it - prefix.begin() << '\n';
    }
}
```

Why it works. Nonnegative daily additions make cumulative totals nondecreasing. Lower bound therefore returns the earliest qualifying day, including when several zero days repeat a total. An end iterator means the final cumulative total is still too small, so no later qualifying day exists.

Complexity. $O(n + q \log(n + 1))$ time and $O(n)$ prefix storage.

Boundary checks. Include initial zero days, repeated prefix values, a target equal to the final total, an unreachable target, and no days.

Solution 15: Minimum integer processing speed

Idea. For a trial speed, sum each job's separate rounded-up time. Increasing speed cannot increase any job time, so the feasibility predicate is false then true.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
bool finishes(const std::vector<long long>& work,
              long long hours, long long speed) {
    long long used = 0;
    for (long long value : work) {
        long long need = value / speed + (value % speed != 0);
        if (need > hours - used) return false;
        used += need;
    }
    return true;
}
int main() {
    int n;
    long long hours;
    std::cin >> n >> hours;
    std::vector<long long> work(n);
    for (auto& value : work) std::cin >> value;
    if (hours < n) {
        std::cout << "-1\n";
        return 0;
    }
    long long lo = 1, hi = *std::max_element(work.begin(), work.end());
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (finishes(work, hours, mid)) hi = mid;
        else lo = mid + 1;
    }
    std::cout << lo << '\n';
}
```

Why it works. The checker counts exactly the hours required under the no-sharing rule. At speed $\max(a_i)$, every job uses one hour, giving a feasible upper bound when $H \geq n$. The first-true search returns the smallest feasible speed. The expression $\text{value}/\text{speed}$ plus a nonzero-remainder test computes each ceiling without a potentially overflowing numerator adjustment.

Complexity. $O(n \log(M + 1))$ time for $M = \max a_i$ and $O(n)$ input storage.

Boundary checks. Use $H < n$, $H = n$, one job, uneven divisibility, and a case where rounding the total size once would underestimate the needed hours.

Solution 16: Exactly k consecutive groups

Idea. Use the minimum-capacity search from the worked partition problem. Its checker needs only to test whether the greedy group count is at most k . The output requirement is exact, but the feasibility test can still use at most.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
bool enoughGroups(const std::vector<long long>& a,
                  int k, long long limit) {
    int groups = 1;
    long long sum = 0;
    for (long long value : a) {
        if (value > limit - sum) {
            ++groups;
            sum = 0;
        }
        sum += value;
    }
    return groups <= k;
}
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    long long lo = *std::max_element(a.begin(), a.end());
    long long hi = std::accumulate(a.begin(), a.end(), 0LL);
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (enoughGroups(a, k, mid)) hi = mid;
        else lo = mid + 1;
    }
    std::cout << lo << '\n';
}
```

Why it works. Any exact- k partition is an at-most- k partition. Conversely, when fewer than k nonempty groups cover n items and k is at most n , some group has at least two items. Split it. Nonnegative values ensure neither resulting sum exceeds the original group sum. Repeat until there are exactly k groups. Thus the two feasibility questions have the same answer for each capacity.

Complexity. $O(n \log(S - M + 1))$ checking work for a nontrivial range, where S is the total and M is the maximum value, plus input. Storage is $O(n)$.

Boundary checks. Use $k = 1$, $k = n$, all zeros, one large item, and a feasible greedy partition with fewer than k groups.

Solution 17: Whole-number rope pieces

Idea. Use a last-true search from 0 to the longest rope. Every tested upper midpoint is positive while the interval has positive width. If all positive lengths fail, the result remains zero.

Complete solution | C++23

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int n;
    long long k;
    std::cin >> n >> k;
    std::vector<long long> ropes(n);
    for (auto& value : ropes) std::cin >> value;
    long long lo = 0, hi = *std::max_element(ropes.begin(), ropes.end());
    while (lo < hi) {
        long long mid = lo + (hi - lo + 1) / 2;
        long long count = 0;
        for (long long rope : ropes) {
            count += rope / mid;
            if (count >= k) break;
        }
        if (count >= k) lo = mid;
        else hi = mid - 1;
    }
    std::cout << lo << '\n';
}
```

Why it works. A shorter positive piece length never decreases the number of obtainable pieces. The true region among positive lengths is therefore a prefix. Zero acts as a defined output sentinel before that region, not as a value passed to the checker. The last-true updates preserve the largest feasible positive length whenever one exists.

Complexity. $O(n \log(M + 1))$ time for $M = \max a_i$ and $O(n)$ stored input. The count fits long long under these bounds.

Boundary checks. Use all-zero ropes, more required pieces than can be made at length 1, one rope, and a maximum answer equal to the longest rope.

Solution 18: Exactly a target sum (extension)

Idea. Count all subarrays with sum at most K , then subtract those whose sum is at most $K-1$. Both passes use the same repaired-window counting helper.

Complete solution | C++23

```
#include <iostream>
#include <vector>
long long atMost(const std::vector<long long>& a, long long k) {
    if (k < 0) return 0;
    int left = 0;
    long long sum = 0, answer = 0;
    for (int right = 0; right < static_cast<int>(a.size()); ++right) {
        sum += a[right];
        while (sum > k) sum -= a[left++];
        answer += right - left + 1;
    }
    return answer;
}
int main() {
    int n;
    long long k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::cout << atMost(a, k) - atMost(a, k - 1) << '\n';
}
```

Why it works. For integer sums, the set difference leaves exactly the subarrays of sum K . Each helper counts all valid starts for every ending position, so zero-valued elements do not cause undercounting. Returning zero for a negative limit correctly handles $K=0$ without an invalid shrinking loop.

Complexity. Two linear scans give $O(n)$ time and $O(1)$ working memory beyond the array.

Boundary checks. Use target 0, a run of zeros, a positive target with zeros at both ends of a valid interval, an absent target, and an empty array.

Solution 19: The kth smallest pair total (synthesis)

Idea. Combine the sorted pair-count scan with first-true answer search. The smallest possible pair total is the sum of the first two sorted values, and the largest is the sum of the last two.

Complete solution | C++23

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

long long countPairs(const std::vector<long long>& a, long long limit) {
    int left = 0, right = static_cast<int>(a.size()) - 1;
    long long count = 0;
    while (left < right) {
        if (a[left] + a[right] <= limit) {
            count += right - left;
            ++left;
        } else --right;
    }
    return count;
}

int main() {
    int n;
    long long k;
    std::cin >> n >> k;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    std::sort(a.begin(), a.end());
    long long lo = a[0] + a[1], hi = a[n - 2] + a[n - 1];
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (countPairs(a, mid) >= k) hi = mid;
        else lo = mid + 1;
    }
    std::cout << lo << '\n';
}
```

Why it works. The inner scan counts each unordered position pair once. As X increases, counted pairs cannot disappear, so $\text{count}(X)$ at least k is monotone. The first satisfying threshold is exactly the k th order statistic, even when many pairs share one sum and the count jumps over k . The supplied endpoints include every possible pair sum.

Complexity. $O(n \log n + n \log(R + 1))$ time, where R is the difference between the largest and smallest pair totals. Input storage is $O(n)$; no quadratic pair list is created.

Boundary checks. Use repeated values, a negative optimum, $k=1$, the last possible rank, and a count that jumps over k at the answer.

Solution 20: Search the window length (comparison challenge)

Idea. The checker asks whether at least one window of a given length has sum at most B . Use a fixed-window scan for each trial length and a last-true search over lengths from 0 to n .

Complete solution | C++23

```
#include <iostream>
#include <vector>
bool exists(const std::vector<long long>& a, int length, long long budget) {
    if (length == 0) return true;
    long long sum = 0;
    for (int i = 0; i < length; ++i) sum += a[i];
    if (sum <= budget) return true;
    for (int i = length; i < static_cast<int>(a.size()); ++i) {
        sum += a[i] - a[i - length];
        if (sum <= budget) return true;
    }
    return false;
}
int main() {
    int n;
    long long budget;
    std::cin >> n >> budget;
    std::vector<long long> a(n);
    for (auto& value : a) std::cin >> value;
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = lo + (hi - lo + 1) / 2;
        if (exists(a, mid, budget)) lo = mid;
        else hi = mid - 1;
    }
    std::cout << lo << '\n';
}
```

Why it works. A feasible length- L window contains a length- $(L-1)$ window of no larger sum. Therefore feasible lengths form an initial interval, with zero defined as feasible. The fixed-window scan checks every candidate segment of the chosen length. The last-true search consequently returns the maximum feasible length.

Complexity. $O(n \log(n+1))$ time and $O(1)$ working memory beyond the array. Example 12 solves the same task in $O(n)$ time by reusing one window rather than restarting a scan for each trial length.

Boundary checks. Use an empty array, budget 0 with zeros, every value above the budget, a whole-array answer, and a signed counterexample to explain the nonnegative restriction.

1.32 Explained mixed drills

Compare these arguments with your own trace. The point is to connect each update to a precise candidate set, not to memorize the particular values used here.

Drill 1: Discard a value or discard a half?

The scan starts at $(0, 5)$, where $2 + 12 = 14 < 16$. Even the largest available partner is too small for 2, so no remaining pair using index 0 can work. Move left to 1. At $(1, 5)$, $4 + 12 = 16$, so a pair has been found.

For the fixed index 1, search for 12 in indices $[2, 6)$. A lower-bound search on that range visits midpoint 4 (value 9), then midpoint 5 (value 12), and returns index 5. A whole-array search could reuse the fixed position when the target is twice its value. Restricting to later positions both prevents self-pairs and gives every unordered pair one orientation.

The endpoint method takes $O(n)$ scanning time. Repeating a logarithmic partner search for every left position takes $O(n \log n)$. Both still need sorting first if the input is not already sorted.

Drill 2: Boundary indices are not necessarily matches

For $x = 4$, the boundaries are $(1, 3)$, so the occurrence count is $3 - 1 = 2$. For $x = 6$, both boundaries are 3 and the count is 0. For $x = 20$, both boundaries are 6 and the count is again 0.

The interval $[4, 9]$ begins at lower bound of 4, index 1, and ends at upper bound of 9, index 5. Its occurrence count is $5 - 1 = 4$: the values are 4, 4, 7, 9.

Both boundaries may be n . They identify a position between elements, not a stored element at every possible result. Subtracting boundary indices or comparing an iterator to end is valid; reading $a[n]$ or dereferencing end is not. An exact-membership test must first check that the lower-bound result is smaller than n .

Drill 3: Find the non-progressing loop

Take $a = [2]$ and $x = 3$. Initially $(lo, hi) = (0, 1)$ and mid is 0. The condition is true, but setting lo to mid keeps the interval unchanged. Every later iteration repeats the same state.

Replace the update with $lo = mid + 1$. When $a[mid] < x$, sorted order proves that every position through mid is also too small. The answer must be strictly after mid , so discarding mid is safe. Since mid is at least lo , the new lo is strictly larger. On the other branch, mid is strictly below hi while lo is below hi , so assigning $hi = mid$ also shrinks the interval.

Do not apply this repair mechanically to every binary-search loop: an inclusive last-true search uses a different invariant and an upper midpoint when it retains a successful mid .

Drill 4: Which input promise is doing the work?

Only the ordinary sum-bounded variable window loses its correctness guarantee. The sorted pair scan can find $-2 + 8 = 6$. Its elimination argument compares ordered values, not their signs.

For the budget input, the window first sees 5, which exceeds 1, and removes it. After reading -4 , it can report length 1, but the original two-element window has total 1 and length 2. The discarded 5 later became useful when a negative value arrived. The greedy removal was not safe under these inputs.

Sorting is not a general repair: it changes which elements are adjacent, so it changes the set of original-order subarrays. Fixed-length rolling sums remain valid with signed values because they maintain the exact next interval; they do not discard candidate starts by a sum monotonicity argument.

Drill 5: Offline queries and saved output order

The sorted query records are $(1, 1), (4, 2), (6, 0), (12, 3)$. The first-unconsumed value index becomes 0, then 3, then stays 3, then becomes 6. These indices are also the requested counts. Store them in answer slots 1, 2, 0, and 3 respectively.

Reading the result slots in original order gives $[3, 0, 3, 6]$. Printing immediately during the sorted sweep would instead produce answers in threshold order, which is not what the caller requested.

With n values and q initially unsorted queries, the sweep costs $O(q \log q + n + q)$ including query sorting and output. Independent upper bounds cost $O(q \log(n + 1))$ searching work plus input and output, and avoid the query-sort step. Neither method is automatically faster for every size combination; reordering is only available when queries do not change the state needed by later queries.

Drill 6: Expand, repair, and measure

The states after each insertion and complete repair are:

Right	Inserted sum	Left after repair	Length	Removed values
0	2	0	1	none
1	3	0	2	none
2	6	0	3	none
3	7	1	3	2
4	6	1	4	none
5	10	3	3	1, then 3

The maximum length is 4, for original indices 1 through 4. At the final step, removing only the first value 1 leaves sum 9, still above budget 6. The invariant that the measured window is valid would be false. This implementation needs repeated repair until the condition holds.

The while loop does not make the scan quadratic: the left index never decreases and can pass each input position at most once over the entire run.

Drill 7: A shortest window has a different measurement point

The running sums are 2, 3, and 6. At $\text{right}=2$, the current window is valid, so record length $2 - 0 + 1 = 3$ before removing anything. Removing 2 then leaves sum 4, which is not valid, and the shrinking loop stops. If the code waited until after removal to measure validity, it could miss the only qualifying window entirely.

For a minimum length, a currently valid interval is a candidate to record and then shorten. For a maximum length under an upper bound, an invalid interval must first be repaired before it can be recorded. Their measurement points come from different objectives.

The supplied shortest-window program requires a positive target. With target 0, the empty window is also valid, so an unchanged shrinking condition could advance beyond it. A nonempty nonnegative-array variant can handle that case separately: answer 1 when n is positive and no nonempty answer when n is zero.

Drill 8: Count starts, not just one best interval

The zero-sum intervals are the first singleton zero, the second singleton zero, and their length-2 interval: count 3. The exact-sum-1 intervals end at the final position and may start at 0, 1, or 2: count 3. Thus $F(1) = 6$, $F(0) = 3$, and $F(1) - F(0) = 3$.

For nonempty subarrays ending at right , the start may equal right . There are $\text{right-left}+1$ valid start indices in an inclusive range. For pairs ending at right , the partner must be strictly smaller than right ; only left through $\text{right}-1$ are permitted, giving right-left partners.

Longest lengths summarize only one extremal size, not the number of intervals. Here the longest lengths for limits 1 and 0 are 3 and 2; their difference is 1, not the exact count 3. Set subtraction applies to occurrence counts, not to maxima.

Drill 9: A window and a prefix search solve the same task

At $\text{start}=1$, the removed prefix has sum $P[1] = 2$. Requiring $P[\text{end}] - 2 \geq 6$ is equivalent to $P[\text{end}] \geq 8$. The first such end after the start is 5, because $P[5] = 8$. The interval is $[1, 5)$, its length is 4, and its one-based inclusive endpoints are 2 and 5.

The prefix sequence is nondecreasing because the original values are nonnegative. A lower-bound search therefore finds the earliest sufficient end for this start, not an arbitrary occurrence of the threshold. Searching separately for each of n starts costs $O(n \log(n + 1))$ after linear prefix construction.

A moving-window solution can reuse endpoint progress and runs in $O(n)$ time. The binary-search solution is still useful as a second derivation and comparison. Negative original values would invalidate both this sorted-prefix argument and the ordinary monotone sum-window argument.

Drill 10: First true, last true, and the midpoint

For A, the desired first true is 4. With $\text{lo}=0$, $\text{hi}=7$, the lower midpoint is 3. $A(3)$ is false, so set $\text{lo}=4$. The interval still contains the first true, and the false tested position is discarded.

For B, the desired last true is 3. The upper midpoint of $[0, 7]$ is 4. $B(4)$ is false, so set $\text{hi}=3$. This retains every possible last success.

With $\text{lo}=2$ and $\text{hi}=3$, a lower midpoint is 2. If it is true and the last-true template sets $\text{lo}=\text{mid}$, nothing changes. The upper midpoint is 3, so a success moves lo to 3 and a failure moves hi to 2. Both branches make progress. Midpoint selection is therefore part of the invariant and update design, not a cosmetic choice.

Drill 11: Prove the checker, not only its monotonicity

At capacity 4, greedy produces $[2, 1]$, $[3, 1]$, $[1]$, and $[4]$: four groups. At capacity 5, it produces $[2, 1]$, $[3, 1, 1]$, and $[4]$: three groups. Feasibility is monotone in capacity, so the smallest capacity for at most three groups is 5.

A first group cannot end later than the longest prefix whose sum fits the cap. Greedy reaches that farthest possible endpoint. Replacing another feasible first block by that prefix only removes nonnegative values from the remaining work; the remainder can still be covered by no more blocks. Repeating this argument shows that greedy uses the fewest groups for the trial cap.

If a feasible partition uses fewer than the required k groups and k is at most n , some block contains at least two items and can be split. Nonnegativity ensures neither new sum is larger. Thus at-most- k and exactly- k feasibility coincide here, but that conclusion needs these explicit assumptions.

Drill 12: An outer search calls an inner scan

The sorted pair-distance list is $[0, 3, 3, 5, 5, 8]$, so the fourth distance is 5. Counts at thresholds 3, 4, and 5 are 3, 3, and 5. There is no threshold with count exactly 4 in this example. The correct predicate is count at least 4; its first true value is the required order statistic.

Within one threshold check, a two-pointer scan reuses a forward left boundary across increasing right endpoints. Across different checks, the threshold may decrease or increase as the outer interval changes. Reusing the old left boundary could skip newly valid pairs; the inner sum and indices must describe a fresh scan for each trial.

Sorting costs $O(n \log n)$. Each trial costs $O(n)$, and there are $O(\log(D + 1))$ trials. Total time is $O(n \log n + n \log(D + 1))$, with $O(n)$ input storage and constant scan state beyond the stored values. A 64-bit count is required at the stated large n bounds.

1.33 One quick reference for both techniques

Use this section after studying the explanations. A formula or update is not a replacement for its input assumptions.

Meanings that must stay consistent

State	Meaning
Closed interval $[l, r]$	Both endpoints are included; length $r - l + 1$ when nonempty.
Half-open interval $[l, r)$	The right endpoint is excluded; length $r - l$.
Lower-bound index of x	Number of sorted values strictly below x ; first index with value at least x , or n .
Upper-bound index of x	Number of sorted values at most x ; first index with value greater than x , or n .
Window sum	Total of the elements currently between the documented boundaries. Update it whenever a boundary changes.
Frequency of value v	Occurrences of v in the current window, not across the whole original input.
Predicate $P(x)$	A yes/no answer to a precisely defined decision problem at candidate x .

Reusable conclusions

Requested quantity	Expression or update
Occurrences of x	$\text{upper}(x) - \text{lower}(x)$.
Occurrences in $[L, R]$	$\text{upper}(R) - \text{lower}(L)$.
Last index with value at most x	$\text{upper}(x) - 1$; -1 means none.
Exact membership	First find lower bound; require index $< n$ before checking equality.
Valid subarrays ending at r	After repair, add $r - l + 1$ when every start in that inclusive range is valid.
Valid distinct-position partners for r	Add $r - l$ when partners range from l through $r - 1$.
Nonempty segment $[s, e)$ sum	$P[e] - P[s]$, with $P[e]$ the total of the first e values.
Exact integer sum count	$F(K) - F(K - 1)$, using an appropriate correct at-most counter.

The occurrence-count formulas require a sorted range. The ordinary sum-window counters in this chapter require nonnegative values. The prefix-sum subtraction identity itself works for signed values, but *binary searching* the prefixes requires their order to be suitable.

Three search contracts, not interchangeable line collections

	Lower bound in data	First true answer	Last true answer
Initial domain	Indices $[0, n)$; boundary may be n .	Inclusive integers $[lo, hi]$ containing the first true; hi is feasible.	Inclusive integers $[lo, hi]$ containing the last true; lo is feasible.
Midpoint	Lower midpoint.	Lower midpoint.	Upper midpoint.
If tested position is too early	$lo = mid + 1$ when $a[mid] < x$.	$lo = mid + 1$ when false.	$lo = mid$ when true.
Other branch	$hi = mid$.	$hi = mid$ when true.	$hi = mid - 1$ when false.
At termination	$lo == hi$; return a boundary, not an automatic match.	Return the first feasible value.	Return the last feasible value.

In bounded examples, the lower midpoint is $lo + (hi - lo) / 2$. An upper midpoint is the lower midpoint plus $(hi - lo) \% 2$. The subtraction and resulting addition still need representable values. The chapter's explicit ranges ensure that condition; an arbitrary full signed-integer range does not.

A five-step implementation checklist

1. State the candidate domain, output convention, and missing-answer behaviour.
2. State the input property that makes elimination safe: sorted values, nonnegative sums, or another proved monotone condition.
3. Define every boundary and every stored summary. Mark inclusive and exclusive endpoints on paper.
4. Prove one iteration preserves correctness and makes progress. For answer search, prove the checker separately.
5. Count all work, including sorting, query sorting, repeated checkers, input/output, and stored copies. Test empty or extreme states allowed by the contract.

Recognize the faster option without rejecting a useful alternative

Independent binary searches are often the simplest way to answer unrelated thresholds. When successive questions have monotone boundaries, two pointers may reuse their progress. A binary search on an answer may also call a two-pointer scan as its checker. The best choice depends on the actual operations, not the chapter heading.

Glossary

Term	Meaning in this session
Candidate	A position, pair, interval, or answer value that has not yet been ruled out.
Invariant	A statement that is true before and after each completed iteration.
Monotone	A property that changes in one direction. A false-then-true predicate has no later false after its first true.
Two pointers	Coordinated indices that exploit order or reuse progress. They are indices here, not raw memory addresses.
Sliding window	A contiguous interval whose boundaries move while a maintained summary describes its contents.
Offline queries	Queries collected before answering so they may be reordered, with original output order restored when required.
Stable filtering	Keeping accepted elements in their original relative order.
Lower / upper bound	The first position not less than / strictly greater than a query under the stated ordering.
Feasibility	Whether a proposed answer value can satisfy the problem's constraints.
Greedy checker	A decision procedure that makes locally chosen steps; its correctness still requires a proof.
Order statistic	A ranked occurrence in a conceptually sorted collection, including repeated values when the task retains them.
Overflow	A result outside the representable range of its type. Applying a safe formula after an unsafe intermediate result is too late.

Readiness check

You are ready to move on when you can explain why a sorted pair move is safe, write a lower-bound loop without copying, distinguish a boundary from a matching element, and choose the correct point at which to record a window's answer.

You should also be able to give a counterexample to an invalid signed-sum window, prove a predicate's direction, establish numerical search bounds, justify a checker, and explain the complete cost when a linear scan runs inside a logarithmic search.

A practice plan for this one session

First pass: boundaries and scans. Study Sections 1.1 to 1.10. Reproduce one pair trace and one lower-bound trace from memory. Try Practice 01 to 08, selecting a small subset first rather than attempting all of them in one sitting.

Second pass: intervals and shared progress. Study Sections 1.11 to 1.17. Solve Practice 09 to 14. For the shortest-target task, compare the window and prefix-search approaches on the same input and explain the complexity difference.

Third pass: decision problems. Study Sections 1.18 to 1.24. Solve Practice 15 to 17, then attempt the synthesis in Practice 19. Do not begin the outer loop until you can test the checker independently on paper.

Second-reading challenges. Revisit exact counting, real-valued search, and Practice 18 to 20. The final comparison asks for a deliberately less efficient valid approach so that you can explain when binary search is unnecessary.

Training sheet and timeline

Sheet link: _____

Start date: _____ **End date:** _____

Your mentors will confirm the practice timeline. During practice, record which assumption or boundary caused a mistake. After the contest, revisit unsolved problems without time pressure and explain the corrected idea before rewriting its code.

Connect this chapter to what comes next

Recursion and backtracking in Session 2 examine problems where alternatives cannot all be eliminated by one monotone boundary. The discipline remains the same: give the state a precise meaning, preserve valid answers, and count the real work.

Do not use exhaustive search merely because a problem is unfamiliar, and do not use two pointers merely because it contains an array. Start with a correct model; then identify what additional structure allows a shortcut.

1.34 Technical notes, verification, and sources

This section records implementation conventions and source references. It is not additional material to memorize before attempting the practice.

Language, input, and arithmetic conventions

The complete listings use standard **C++23** headers. Bounds assume an int wide enough for indices up to 200000, as in the tested GNU environment, and long long wide enough for the stated totals and pair counts. C++ does not require int to have exactly 32 bits. Each example gives its input contract; programs do not print prompts or validate arbitrary malformed input.

The string examples use their stated ASCII alphabet. Complexity treats fixed-width arithmetic and ordinary element comparisons as constant-cost operations. String or container copies are included when the code makes them. Stored input is identified separately from a scan's auxiliary state; sorting an input vector is not described as allocating a second copy of it.

For nonnegative operands, integer division computes the floor quotient used in the production and cutting examples. The ceiling identity used for positive denominators is $a / b + (a \% b \neq 0)$. Signed overflow is not a reliable wraparound mechanism; numeric bounds must protect intermediate results.^{[1-5][1-6]}

The library binary-search algorithms require their documented partitioning conditions. Nondecreasing vectors satisfy the ordinary comparison conditions used here. They perform logarithmic comparisons, but a general iterator range without random access can require linear iterator movement. This chapter's search examples use vectors.^{[1-1][1-2][1-3]}

What was checked for this rewritten session

The source package contains **24 complete examples and 20 complete practice solutions**. All **44 programs** compiled with GCC 14.2.0 in C++23 mode and passed **12,685 input/output checks**. These include every printed sample, independent small-input comparisons, exhaustive small binary-array cases, and large-boundary tests.

The test oracles use direct pair and interval enumeration, small partition or placement enumeration, independent linear searches, or exact rational optima as appropriate. The original-index pair solution is checked by validating the returned witness rather than requiring one arbitrary pair. The real-valued answer is checked with the stated numerical tolerance.

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

These checks help expose mistakes on executed paths; they are not a proof for every allowed input. The correctness arguments supply the mathematical justification. Compiler warnings, sanitizer instrumentation, and standard-library assertions are separate safeguards.^{[1-8][1-9][1-10]}

Rebuild and reproduce

Run `python build.py --target session1` from the project root to compile this handout, or `python build.py --target book` for the complete collected book. Both use this same newly written chapter and its program files. The README includes the full build and verification commands.

Run `python tests/verify_session_01.py` to rebuild and test the 44 programs. The test seed, compiler flags, case counts, and file hashes are recorded in `reports/session_01_verification.json`. A separate drill-check script verifies selected numerical claims in the written exercises.

The rewrite replaces the former two-topic Session 1 source with one ordinary section sequence and one set each of practice, hints, solutions, drills, and references. Sessions 2 to 6 retain their existing teaching content; release notes and cross-session descriptions identify the current organization.

Language and implementation references

The first seven entries point to a fixed public rendering of the N4950 C++23 working draft, not a moving draft for a later language version. These are technical references for exact rules; the explanations, traces, exercises, and proofs in this chapter are written for the training.

[1-1] C++23 N4950: lower bound

<https://timsong-cpp.github.io/cppwp/n4950/lower.bound>

[1-2] C++23 N4950: upper bound

<https://timsong-cpp.github.io/cppwp/n4950/upper.bound>

[1-3] C++23 N4950: binary-search algorithms and iterator costs

<https://timsong-cpp.github.io/cppwp/n4950/alg.binary.search>

[1-4] C++23 N4950: sort

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

[1-5] C++23 N4950: expressions and unrepresentable results

<https://timsong-cpp.github.io/cppwp/n4950/expr.pre>

[1-6] C++23 N4950: multiplication, division, and remainder

<https://timsong-cpp.github.io/cppwp/n4950/expr.mul>

[1-7] C++23 N4950: accumulation

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

Compiler checks and further reading

[1-8] GCC: instrumentation and sanitizer options

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

[1-9] GCC: warning options

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

[1-10] GNU libstdc++: configurable macros and assertions

https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_macros.html

Further reading after the core practice

Antti Laaksonen's *Competitive Programmer's Handbook* provides another treatment of binary search and two-pointer methods. Its language conventions and organization differ from this C++23 handout; use it to compare explanations, not to replace the input assumptions in a particular problem.

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

The last question to ask yourself

Can I explain which candidates an update removes and why none of them are still needed? If yes, the loop is becoming an algorithm you understand. If not, return to a smaller trace before memorizing another template.