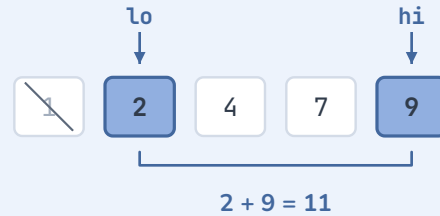


Two pointers

Chapter 2.2 · Overall level: Core



A shop lists its prices in increasing order: 1, 2, 4, 7, 9. You hold a gift card worth exactly 11. *Can you buy two different items that use up the card exactly?* Two fingers, one at each end of the list, answer this without ever trying most of the pairs.

Two pointers are two indices that walk through an array, each only ever moving in one direction. The trick is always the same: a short argument shows that whatever a pointer leaves behind can never be part of an answer. This chapter uses it twice: pointers moving towards each other in a sorted array, and a *window* whose two ends both move right.

2.2.1 Pairs with a target sum

The direct method tries every pair of positions: $n(n - 1)/2$ of them. For the five prices that is 10 pairs; for 200,000 prices it is about 2×10^{10} , far too many.

Sorting gives us information the direct method ignores. Put a pointer `lo` on the smallest value and `hi` on the largest, and look at $a[lo] + a[hi]$. Three things can happen:

- **The sum equals the target.** We have found a pair. Stop.
- **The sum is too small.** $a[hi]$ is the largest value still available, so $a[lo]$ plus anything else is even smaller. $a[lo]$ cannot be in any answer: move `lo` one step right.
- **The sum is too large.** Symmetrically, $a[lo]$ is the smallest value available, so $a[hi]$ plus anything is too large. Move `hi` one step left.

Each step either finishes or throws away one position for good, so after at most $n - 1$ steps the pointers meet.

2.2.2 Which pointer can be discarded

On the gift-card example the search takes two steps. The first sum is $1 + 9 = 10$, one short of 11. Pairing 1 with 7, 4 or 2 instead would be shorter still, so the value 1 is finished with.

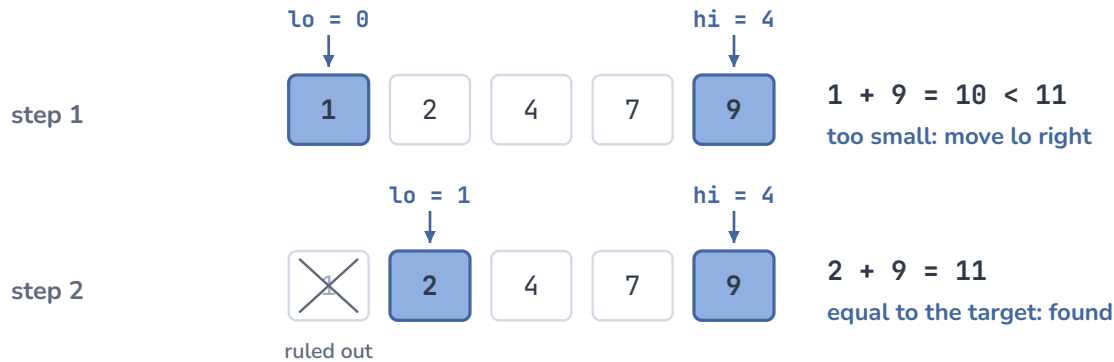


Figure 2.2.1 — If the sum is too small, pairing the same left value with a smaller right value cannot help. The crossed-out cell is eliminated for good.

A larger picture shows what each step really discards. Write every pair sum in a table, row i and column j for $i < j$. Because the array is sorted, sums grow from left to right along each row and from top to bottom down each column. The pointers (lo , hi) always sit at the top-right corner of the part of the table that is still alive. Here the target is 6, so both kinds of move appear.

	$a[1] = 2$	$a[2] = 4$	$a[3] = 7$	$a[4] = 9$	
$a[0] = 1$	3	5	8	10	1: $10 > 6$, so the whole $a[4]$ column is too large
$a[1] = 2$		6	9	11	2: $8 > 6$, so the $a[3]$ column is too large
$a[2] = 4$			11	13	3: $5 < 6$, so the whole $a[0]$ row is too small
$a[3] = 7$				16	4: $2 + 4 = 6$, found

sums grow to the right along a row and downward in a column

Figure 2.2.2 — Each comparison removes a whole row or a whole column of candidate pairs. Grey sums are ruled out without being computed; dashed cells were visited in the numbered order.

The corner is the largest sum in its row and the smallest in its column. If it is too small, its whole row is too small; if it is too large, its whole column is too large. Either way one line of the table disappears, and the four comparisons above stand in for all ten pairs.

Why no pair is skipped

The method is correct because of an **invariant**, a statement that is true before every step:

If some pair of different positions sums to the target, then some such pair has both positions in the range $lo \dots hi$.

At the start the range is the whole array, so the invariant holds. Suppose it holds and $a[lo] + a[hi] < x$. Any pair that uses position lo together with some j in the range has $a[lo] + a[j] \leq a[lo] + a[hi] < x$, because $a[j] \leq a[hi]$. So no answer inside the range uses lo , and removing it keeps the invariant. The case $a[lo] + a[hi] > x$ is the mirror image for hi .

If the loop ends with $lo = hi$, the range holds a single position, which cannot form a pair with itself, so the invariant says there was no answer at all. Duplicates need no special care: $[2, 2]$ with target 4 is found on the first step, because the two 2s sit at different positions.

Pseudocode

Opposing pointers: $O(n)$ after sorting

```

lo ← 0, hi ← n - 1
while lo < hi:
    s ← a[lo] + a[hi]
    if s = x: answer YES
    if s < x: lo ← lo + 1           ▶ a[lo] is too small for every partner
    else:    hi ← hi - 1          ▶ a[hi] is too large for every partner
answer NO

```

If the input is not sorted, sort it first (Chapter 1.5) for a total of $O(n \log n)$. If the problem asks for the positions in the original input, sort (value, position) pairs.

2.2.3 Code in four languages

Each program solves the first practice problem, **Target pair**. The first line holds n and the target x , the second line n integers in nondecreasing order. The program prints YES if two different positions sum to x , otherwise NO.

Sample input	Sample output
5 11 1 2 4 7 9	YES

The values lie between -10^9 and 10^9 , so a sum of two, like the target, lies between -2×10^9 and 2×10^9 . That fits in a signed 32-bit `int` (limit 2,147,483,647), but only just. The C++ and Java programs keep values, target and sums in 64-bit integers, so a slightly wider range could never overflow silently.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n, x = int(data[0]), int(data[1])
    a = list(map(int, data[2:2 + n]))
    lo, hi = 0, n - 1
    while lo < hi:
        s = a[lo] + a[hi]
        if s == x:
            print("YES")
            return
        if s < x:
            lo += 1
        else:
            hi -= 1
    print("NO")

main()
```

C++17

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    long long x;
    cin >> n >> x;
    vector<long long> a(n);
    for (auto &v : a) cin >> v;
    int lo = 0, hi = n - 1;
    while (lo < hi) {
        long long s = a[lo] + a[hi];    // up to 2e9: needs 64 bits
        if (s == x) {
            cout << "YES\n";
            return 0;
        }
        if (s < x) lo++;
        else hi--;
    }
    cout << "NO\n";
    return 0;
}
```

Java 17

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken();
        int n = (int) in.nval;
        in.nextToken();
        long x = (long) in.nval;
        long[] a = new long[n];
        for (int i = 0; i < n; i++) {
            in.nextToken();
            a[i] = (long) in.nval;
        }
        int lo = 0, hi = n - 1;
        while (lo < hi) {
            long s = a[lo] + a[hi];
            if (s == x) {
                System.out.println("YES");
                return;
            }
            if (s < x) lo++;
            else hi--;
        }
        System.out.println("NO");
    }
}
```

JavaScript (Node)

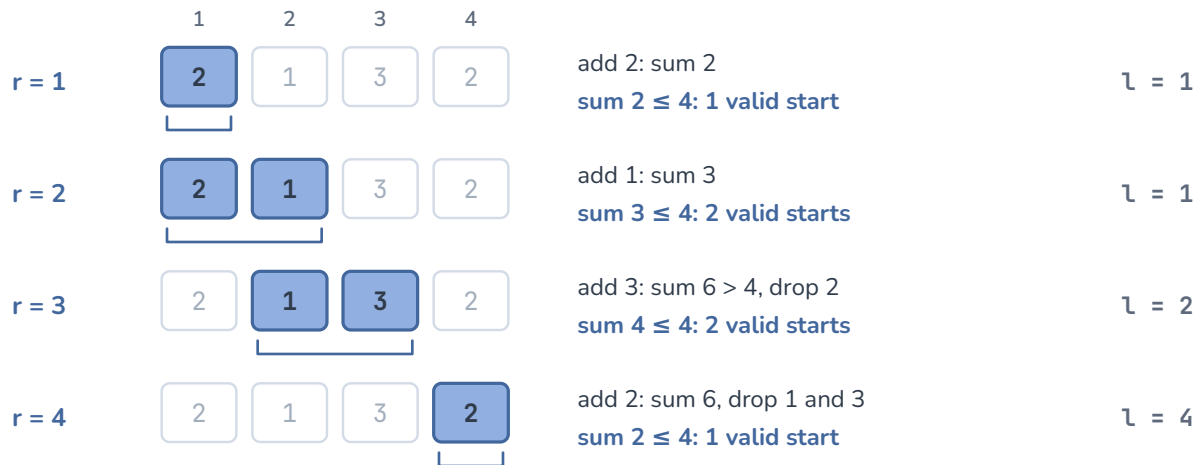
```
const data = require("fs").readFileSync(0, "utf8")
    .trim().split(/\s+/).map(Number);
const n = data[0], x = data[1];
const a = data.slice(2, 2 + n);
let lo = 0, hi = n - 1, found = false;
while (lo < hi) {
    const s = a[lo] + a[hi];
    if (s === x) {
        found = true;
        break;
    }
    if (s < x) lo++;
    else hi--;
}
console.log(found ? "YES" : "NO");
```

JavaScript numbers are exact integers below $2^{53} \approx 9 \times 10^{15}$, so sums up to 2×10^9 need no special care.

2.2.4 A window over positive values

The second use of two pointers looks at consecutive blocks. Take $a = [2, 1, 3, 2]$ and a budget $K = 4$. Which consecutive blocks have sum at most K ? There are 10 blocks, and checking each one separately repeats a lot of addition.

Instead, keep a *window* $a[l..r]$ and its running sum. Move the right end r forward one step at a time, adding the new value. If the sum is now over budget, move the left end l forward, subtracting values, until the sum fits again. After this shrinking, l is the **earliest start** that fits for this right end.



Valid windows: $1 + 2 + 2 + 1 = 6$. Longest valid window: length 2.

Figure 2.2.3 — For positive values, removing the leftmost element can only decrease the sum. The dark cells form the window after shrinking; grey cells are outside it.

Two facts make this correct, and both need every value to be positive.

1. **Shrinking helps.** Removing the leftmost value lowers the sum, so if $a[l..r]$ fits, every shorter block $a[l'..r]$ with $l' > l$ fits too. The valid starts for a right end form one unbroken stretch $l..r$.
2. **The left end never moves back.** If $a[l..r]$ is over budget, then $a[l..r + 1]$ is over budget too, since it adds a positive value. So a start thrown away for r is useless for every later right end, and l can stay where it is.

So the longest valid block ending at r has length $r - l + 1$, and the longest valid block overall is the largest of these: 2 in the example.

Negative values break the rule

With a negative value, removing an element can *increase* the sum, and both facts fail. Take $a = [5, -4, 3]$ with $K = 4$. At $r = 1$ the block $[5]$ has sum $5 > 4$, so the window throws position 1 away for good. But the whole array $[5, -4, 3]$ has sum 4 and fits: the window can never find it and reports length 2 instead of 3. Problems with signed values need a different tool, such as prefix sums (Chapter 2.1) combined with sorting or searching, and are outside this chapter.

2.2.5 Counting every valid window

Suppose we must count *all* valid blocks rather than find the longest. After shrinking, the valid blocks ending at r are exactly those starting at $l, l + 1, \dots, r$: there are $r - l + 1$ of them, not just one.

Right end r	Window after shrinking	Sum	Valid starts	Blocks counted
1	[2]	2	1	[2]
2	[2, 1]	3	1, 2	[2, 1], [1]
3	[1, 3]	4	2, 3	[1, 3], [3]
4	[2]	2	4	[2]

Adding $r - l + 1$ for each r gives $1 + 2 + 2 + 1 = 6$. Every block has exactly one right end, so it is counted exactly once, at that end. The total can be as large as $n(n + 1)/2 \approx 2 \times 10^{10}$ for 200,000 values, which needs a 64-bit counter.

Pseudocode

Sliding window for positive values: $O(n)$

```

l ← 1, sum ← 0
for r from 1 to n:
    sum ← sum + a[r]                ▶ extend the window to the right
    while sum > K:                  ▶ shrink until the window is valid
        sum ← sum - a[l]
        l ← l + 1
    ▶ now a[l..r] is the longest valid block ending at r (empty if l = r + 1)
    use r - l + 1                  ▶ its length, or the number of valid starts

```

If a single value is bigger than K , the loop shrinks right past it: l becomes $r + 1$, the window is empty, and $r - l + 1 = 0$ blocks end there. That is why an empty window must be allowed during the scan.

2.2.6 Windows with few distinct values

The window idea works for any condition that behaves like the budget: if a block is valid, every block inside it is valid. “At most k distinct values” is such a condition. Removing an element can never add a new value, so shrinking still helps, and adding one can only add values.

The window now needs a way to know how many distinct values it holds. Keep a frequency table `count`, a dictionary from value to how many times it occurs in the window (the tallies of Chapter 1.4), and a number `distinct`:

- **Add a value v :** increase `count[v]`. If it went from 0 to 1, v is new to the window: increase `distinct`.
- **Remove a value v :** decrease `count[v]`. If it reached 0, the last copy has left: decrease `distinct` and delete the key, so the dictionary only holds values in the window.

Here is the trace for `a = [5, 1, 5, 2, 2, 1]` with at most $k = 2$ distinct values. Shrinking happens while `distinct > 2`.

<code>r</code>	<code>a[r]</code>	Removed while shrinking	<code>count</code> after shrinking	<code>l</code>	Length
1	5	–	{5: 1}	1	1
2	1	–	{5: 1, 1: 1}	1	2
3	5	–	{5: 2, 1: 1}	1	3
4	2	5 (count 2 → 1), then 1 (count 1 → 0)	{5: 1, 2: 1}	3	2
5	2	–	{5: 1, 2: 2}	3	3
6	1	5 (count 1 → 0)	{2: 2, 1: 1}	4	3

At `r = 4` notice that removing the first 5 did not reduce `distinct`, because another 5 was still inside; only removing the 1 did. The longest valid block has length 3: [5, 1, 5] at positions 1–3, [5, 2, 2] at 3–5 and [2, 2, 1] at 4–6 all qualify.

2.2.7 Why the total work is linear

The window loop has a `while` inside a `for`, which looks like $O(n^2)$. It is not. Count pointer moves instead of loop nesting: `r` moves forward n times in total, and every pass through the inner loop moves `l` forward once. Since `l` only moves forward and never passes `r + 1`, it moves at most n times *over the whole run*, however those moves are spread out. So the inner loop runs at most n times in total and the method is $O(n)$. In the example of Figure 2.2.3, `l` moved three times in all.

The opposing-pointer search is linear for the same reason: every step moves `lo` right or `hi` left, and the gap `hi - lo` starts at $n - 1$ and shrinks by one each step.

2.2.8 Common mistakes

- **Using the wrong monotonicity assumption.** The window rule needs “shrinking keeps a block valid”. With negative values in a sum window it is false; with a condition like “sum at least K ” the direction reverses. Check the property before you write the loop.
- **Counting one start per right end.** When counting valid blocks, add $r - l + 1$ for each r , not 1.
- **Forgetting to decrement a removed frequency.** If count is not lowered when l moves, values that left the window still look present and `distinct` never goes down.
- **Pairing a position with itself.** Loop while $l_0 < h_i$, not $l_0 \leq h_i$; otherwise $[1, 3, 5]$ with target 10 reports $5 + 5$.
- **32-bit overflow.** Pair sums reach 2×10^9 , window sums 2×10^{11} and window counts 2×10^{10} ; use 64-bit integers in C++ and Java.

2.2.9 Cost, and where two pointers stop working

Both patterns take $O(n)$ time after any sorting, and $O(1)$ extra memory, apart from the frequency dictionary, which holds at most one key per distinct value in the window. Dictionary operations are expected $O(1)$, so the distinct-values window is expected $O(n)$.

Situation	What to use instead
Sum window, but some values are negative	Prefix sums (Chapter 2.1) with sorting or searching
Pair target in unsorted data, positions needed	Sort (value, position) pairs, or a dictionary of seen values (Chapter 1.4)
Need the smallest value inside every window	A monotonic deque: an optional extension beyond this chapter

Exercises

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

1. ● Why is moving each pointer at most n times $O(n)$, even though the window loop has a `while` inside a `for`?
2. ● Give a reason why the sum window needs positive (or nonnegative) values.
3. ● Trace opposing pointers on $[-3, 0, 2, 5, 8]$ with target 7. List each pair of positions examined and its sum.
4. ●● For $[3, 1, 2, 4]$ and $K = 6$, find the left end after shrinking for each right end. How many valid blocks are there, and what is the longest?
5. ●● What goes wrong for $[1, 3, 5]$ and target 10 if the loop condition is $l_0 \leq h_i$?
6. ●● Find the longest block of $[7, 7, 3, 3, 3, 7]$ with at most one distinct value, and give count just after $r = 6$ is processed.

Answers and explanations

Read these after attempting the exercises.

1. The inner loop moves l forward, and l never moves back, so all its moves together number at most n across the whole run. The movements accumulate over the entire scan, not per right end: at most $2n$ pointer moves in total.
2. Removing a negative number makes the sum *larger*, so shrinking may not restore validity, and a block that is too large may become valid when it is extended. Both facts of Section 2.2.4 fail.
3. (0, 4): $-3 + 8 = 5 < 7$, move lo. (1, 4): $0 + 8 = 8 > 7$, move hi. (1, 3): $0 + 5 = 5 < 7$, move lo. (2, 3): $2 + 5 = 7$, found.
4. $r = 1$: [3], sum 3, $l = 1$. $r = 2$: [3, 1], sum 4, $l = 1$. $r = 3$: [3, 1, 2], sum 6, $l = 1$. $r = 4$: sum 10, drop 3 and 1, [2, 4], sum 6, $l = 3$. Counts $1 + 2 + 3 + 2 = 8$ valid blocks; the longest has length 3.
5. The sums $1 + 5 = 6$ and $3 + 5 = 8$ are too small, so lo reaches position 3 while hi is still 3. With $l_o \leq h_i$ the loop tests $5 + 5 = 10$ and wrongly answers YES, using one position twice.
6. The longest is 3 (positions 3–5, three 3s). At $r = 6$ the 7 enters, the window [3, 3, 3, 7] has 2 distinct values, and shrinking removes all three 3s, deleting the key: $\text{count} = \{7: 1\}$, $l = 6$.

Practice this topic

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

Target pair

[Apply](#)

Determine whether two different positions in a nondecreasing array sum to x .

Longest affordable run

[Connect](#)

Find the greatest length of a consecutive block whose sum is at most k . All values are positive. Print 0 if no nonempty block fits.

Count affordable runs

[Reason](#)

Count all nonempty consecutive blocks whose sum is at most k . All values are positive.

At most k signals

[Stretch](#)

Find the longest consecutive block containing at most k distinct values.