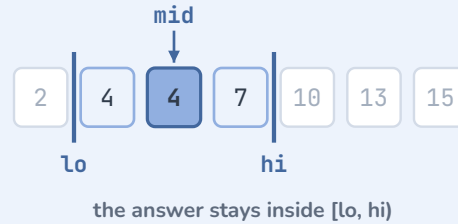


Binary search

Chapter 2.3 · Overall level: Core



I am thinking of a whole number from 1 to 1,000. After each guess I say “higher” or “lower”. *How many guesses do you need?* Never more than 10, if every guess cuts the remaining range in half.

Binary search is that guessing game played on a sorted array, or on any question whose answer switches from “no” to “yes” exactly once. It looks simple, and it is famous for off-by-one bugs. This chapter uses one careful form, a half-open interval with a written invariant, so that every version you write is correct for the same reason.

2.3.1 Searching for a boundary

Take the sorted array $a = [2, 4, 4, 7, 10]$ and the question: *where is the first value that is at least 4?* The value 4 appears twice, and the answer must be the first one, index 1 (position 2 when counting from 1).

Ask the yes/no question “is $a[i] \geq 4$?” at every index. Because the array is sorted, all the “no” answers come first and all the “yes” answers after them. So we are not looking for a value; we are looking for the **boundary** where no turns into yes.

	0	1	2	3	4	
$a[i]$	2	4	4	7	10	sorted, so every “no” comes before every “yes”: we want the first yes
$a[i] \geq 4$?	F	T	T	T	T	

Figure 2.3.1 — In a sorted array the answers to “ $a[i] \geq x$?” form a block of F followed by a block of T. The search finds the first T.

If every value is smaller than x , there is no T at all. Then the boundary sits just after the last index, at n . Returning n for “not found” is natural: it is the place where x would be inserted to keep the array sorted.

2.3.2 The half-open search

Keep two numbers lo and hi and treat them as *boundaries* between cells, not as cells: the answer is known to lie in the half-open range $[lo, hi)$, which includes lo and stops just before hi . Start with $lo = 0$ and $hi = n$, so that the answer n (“none”) is also allowed. Then repeat while the range is not empty:

- Pick the middle index $mid = (lo + hi) / 2$, rounded down. It always lies inside $[lo, hi)$.
- If $a[mid] \geq x$, index mid qualifies, so the first qualifying index is mid or to its left: set $hi = mid$.
- Otherwise $a[mid]$ and everything to its left are too small: set $lo = mid + 1$.

When $lo = hi$, the range is empty and lo is the answer. Here is the search for $x = 4$, with the full array always visible.

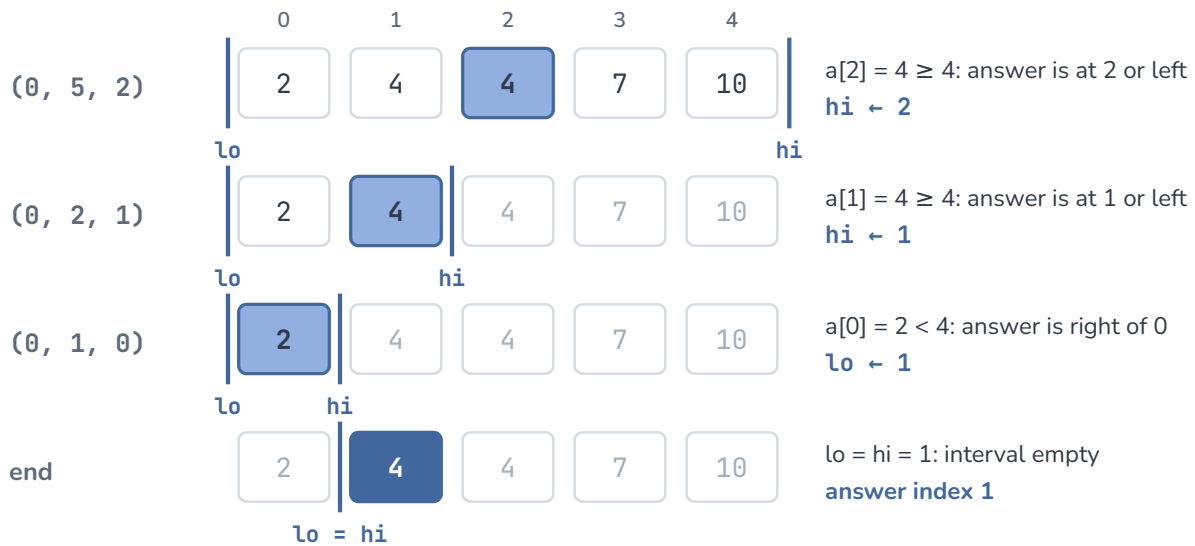


Figure 2.3.2 — The first qualifying position remains inside $[lo, hi)$, until the interval is empty. Blue bars are the boundaries lo and hi ; grey cells are outside the interval; the dark cell is mid .

The search for $x = 11$ shows the not-found case: (0, 5, 2) has $4 < 11$, so $lo = 3$; (3, 5, 4) has $10 < 11$, so $lo = 5$. Now $lo = hi = 5 = n$: no value qualifies.

2.3.3 Why it works: the invariant

Each update is justified by a statement that is true before every step of the loop:

Every index before `lo` fails ($a[i] < x$), and every index from `hi` on qualifies ($a[i] \geq x$).

At the start nothing lies before 0 and nothing lies from n on, so the statement is true. If $a[mid] \geq x$, then `mid` and, by sorting, every later index qualifies, so $hi = mid$ keeps it true. If $a[mid] < x$, then `mid` and every earlier index fails, so $lo = mid + 1$ keeps it true.

Every update shrinks the interval. Because $lo \leq mid < hi$, setting $hi = mid$ makes hi smaller and setting $lo = mid + 1$ makes lo larger. Each step roughly halves the length $hi - lo$, so the loop ends after about $\log_2 n + 1$ steps: 18 steps for 200,000 values.

The answer at the end. When $lo = hi$, the invariant says everything before lo fails and everything from lo on qualifies. That is exactly the definition of the first qualifying index, or n if there is none.

Pseudocode

Lower bound: $O(\log n)$ per question

```
lower_bound(a, x):                                ▶ first index i with a[i] ≥ x, or n
    lo ← 0, hi ← n
    while lo < hi:
        mid ← lo + (hi - lo) / 2                  ▶ rounded down; lo ≤ mid < hi
        if a[mid] ≥ x: hi ← mid
        else: lo ← mid + 1
    return lo
```

Writing $lo + (hi - lo) / 2$ instead of $(lo + hi) / 2$ gives the same value but never overflows, which matters in C++ and Java when lo and hi are large numbers rather than array indices.

Two updates that break the search

Small changes ruin it. With $lo = mid$ instead of $mid + 1$, the state $lo = 0, hi = 1$ gives $mid = 0$ again and again: the interval never shrinks and the loop never ends. With $hi = mid - 1$, a qualifying mid is thrown away: for $x = 10$ the search ends at index 3, but $a[3] = 7$ is less than 10.

2.3.4 Lower and upper bounds

Change the question to “is $a[i] > x$?” and the same loop finds the first value *strictly* greater than x . This is the **upper bound**. Between the two boundaries lie exactly the copies of x : indices before the lower bound hold values below x , indices from the upper bound on hold values above it.

x	lower bound (first $\geq x$)	upper bound (first $> x$)	copies of x
4	1	3	$3 - 1 = 2$
3	1	1	0
10	4	5	1
11	5 (= n , none)	5	0
1	0	0	0

Most libraries provide both searches. Use them when they exist, and know what they return:

Language	Lower bound	Upper bound
Python	<code>bisect_left(a, x)</code> from <code>bisect</code>	<code>bisect_right(a, x)</code>
C++	<code>lower_bound(a.begin(), a.end(), x) - a.begin()</code>	<code>upper_bound(...) - a.begin()</code>
Java	write your own: <code>Arrays.binarySearch</code> finds <i>some</i> copy, not the first	write your own
JavaScript	write your own	write your own

2.3.5 Code in four languages

Each program solves the first practice problem, **First qualifying position**. The first line holds n and q , the second line the n sorted values, then q lines with one x each. For each x the program prints the 1-based position of the first value at least x , or 0 if there is none.

Sample input	Sample output
5 3	2
2 4 4 7 10	0
4	1
11	
1	

The search returns a 0-based index i in $0..n$; the program prints $i + 1$, or 0 when $i = n$. Each program writes the search by hand so that the four versions match the pseudocode line for line.

Python 3

```
import sys

def lower_bound(a, x):
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] >= x:
            hi = mid
        else:
            lo = mid + 1
    return lo

def main():
    data = sys.stdin.buffer.read().split()
    n, q = int(data[0]), int(data[1])
    a = list(map(int, data[2:2 + n]))
    out = []
    for k in range(q):
        i = lower_bound(a, int(data[2 + n + k]))
        out.append(i + 1 if i < n else 0)
    print("\n".join(map(str, out)))

main()
```

C++17

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

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    vector<int> a(n);
    for (int &v : a) cin >> v;
    while (q--) {
        int x;
        cin >> x;
        int i = lowerBound(a, x);
        cout << (i < n ? i + 1 : 0) << '\n';
    }
}
```

Java 17

```
import java.io.*;

public class Main {
    static int lowerBound(int[] a, int x) {
        int lo = 0, hi = a.length;
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] >= x) hi = mid;
            else lo = mid + 1;
        }
        return lo;
    }

    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();
        int q = (int) in.nval;
        int[] a = new int[n];
        for (int i = 0; i < n; i++) {
            in.nextToken();
            a[i] = (int) in.nval;
        }
        StringBuilder out = new StringBuilder();
        for (int k = 0; k < q; k++) {
            in.nextToken();
            int i = lowerBound(a, (int) in.nval);
            out.append(i < n ? i + 1 : 0).append('\n');
        }
        System.out.print(out);
    }
}
```

Values and queries lie between -10^9 and 10^9 , so 32-bit integers suffice; the search only compares, it never adds two values. In JavaScript, `(lo + hi) >> 1` is safe because indices stay below 2^{31} .

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8")
  .trim().split(/\s+/).map(Number);
const n = data[0], q = data[1];
const a = data.slice(2, 2 + n);

function lowerBound(x) {
  let lo = 0, hi = a.length;
  while (lo < hi) {
    const mid = (lo + hi) >> 1;
    if (a[mid] >= x) hi = mid;
    else lo = mid + 1;
  }
  return lo;
}

const out = [];
for (let k = 0; k < q; k++) {
  const i = lowerBound(data[2 + n + k]);
  out.push(i < n ? i + 1 : 0);
}
console.log(out.join("\n"));
```

2.3.6 Searching over the answer

Binary search does not need an array. It needs a yes/no question about a number that switches from “no” to “yes” once. Many optimisation problems hide such a question: instead of computing the best value directly, ask “is the value T good enough?”

Two machines make items in parallel. Machine 1 finishes one every 2 minutes and machine 2 one every 3 minutes. When have they made 5 items in total? By time T , a machine with period t has finished $\lfloor T/t \rfloor$ items, so the question “is T enough?” is a short scan:

$$\text{feasible}(T): \lfloor T/2 \rfloor + \lfloor T/3 \rfloor \geq 5$$

time T	0	1	2	3	4	5	6	7	8
items $T/2 + T/3$	0	0	1	2	3	3	5	5	6
at least 5 ?	F	F	F	F	F	F	T	T	T

first feasible time: 6

Figure 2.3.3 — More available time cannot reduce the number of produced items, so feasibility is monotone. $T = 5$ gives $2 + 1 = 3$ items; $T = 6$ gives $3 + 2 = 5$.

This is exactly the picture of Figure 2.3.1, with times instead of indices. Before searching, three things must be settled:

1. **Monotone.** If T is feasible, so is every larger time, because no machine un-makes an item. This must be *proved*, not assumed: it is what makes the F block come before the T block.
2. **A range that surely contains the answer.** The lower end $lo = 0$ is safe. The upper end must be a time known to be feasible: the fastest machine alone makes k items by time $\min(t) \times k$, so $hi = 2 \times 5 = 10$ works. If hi were not feasible, the search could return it anyway.
3. **A fast check.** Each call of `feasible` scans the machines once.

Then run the lower-bound loop on numbers: $(lo, hi, mid) = (0, 10, 5)$ is infeasible (3 items), so $lo = 6$; $(6, 10, 8)$ is feasible, $hi = 8$; $(6, 8, 7)$ is feasible, $hi = 7$; $(6, 7, 6)$ is feasible, $hi = 6$. The answer is 6.

Pseudocode

Search over the answer: $O(\text{cost of check} \times \log(\text{range}))$

```
lo ← smallest possible answer
hi ← an answer known to be feasible
while lo < hi:
    mid ← lo + (hi - lo) / 2
    if feasible(mid): hi ← mid      ▶ mid works: the answer is mid or smaller
    else:             lo ← mid + 1 ▶ mid fails: so does everything below it
return lo
```

2.3.7 Splitting into groups under a capacity

A second classic: split $[4, 3, 2, 6, 1, 5]$ into at most $k = 3$ consecutive groups so that the largest group sum is as small as possible. Turn it into a yes/no question: *can it be split into at most 3 groups if every group sum must be at most c ?*

For a fixed capacity c , a left-to-right scan answers it: keep adding values to the current group, and start a new group only when the next value would push the sum over c . The scan ends every group as late as possible. No other valid split can do better: by induction, after any number of groups the scan has covered at least as many values as that split, so it never needs more groups.

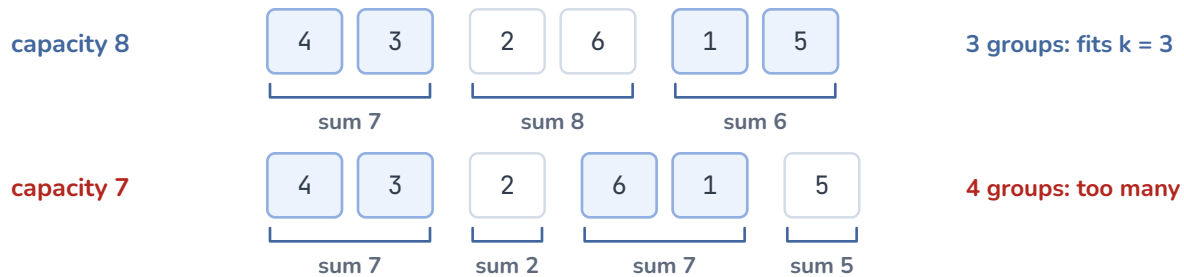


Figure 2.3.4 — Capacity 8 needs 3 groups and capacity 7 needs 4, so with $k = 3$ the smallest possible largest sum is 8.

The answer is monotone: a split that works for capacity c also works for any larger capacity. It lies between $\max(a) = 6$, since every group must hold its largest element, and $\text{sum}(a) = 21$, one group holding everything, which is always feasible. The search visits $(6, 21, 13)$ feasible, $(6, 13, 9)$ feasible, $(6, 9, 7)$ infeasible, $(8, 9, 8)$ feasible, and returns 8.

2.3.8 Cost and number sizes

A lower-bound search costs $O(\log n)$, so q questions on a sorted array cost $O((n + q) \log n)$ including the sort, if one is needed. A search over answers in a range of size R calls the check about $\log_2 R$ times: for $R = 10^{12}$ that is 40 calls, each $O(n)$ above.

The candidate answers themselves can be large: $\min(t) \times k$ reaches $10^6 \times 10^6 = 10^{12}$ and a total of group sums 2×10^{11} . Use 64-bit integers for `lo`, `hi`, `mid` and every running total, and stop adding production once it reaches the target. JavaScript numbers are exact up to 2^{53} , so `Math.floor(T / t)` is safe here.

2.3.9 Common mistakes

- **An update that does not shrink the interval.** `lo = mid` can loop forever. With a half-open range the updates are `hi = mid` and `lo = mid + 1`.
- **An upper bound that is not guaranteed feasible.** If `hi` might fail, the search can return a wrong answer. Prove that `hi` works, as with `min(t) × k` or `sum(a)`.
- **Overflow while computing products or totals.** `min(t) × k`, `mid` and production counts exceed 32 bits. Use `long long / long`, and cap running sums at the target.
- **Stopping at any match.** Returning as soon as `a[mid] = x` finds a copy, not the first. Search for the boundary.
- **Forgetting the not-found boundary.** The lower bound may be n . Check before reading `a[n]`.
- **Searching unsorted data or a non-monotone question.** If F and T are mixed, the halving argument is meaningless.

2.3.10 Where binary search stops

The method needs one switch from “no” to “yes”. If feasibility can go back and forth, for example “the profit at price T is at least 100” when profit first rises and then falls, there is no single boundary and binary search may miss the answer. If the array changes between questions, keeping it sorted costs $O(n)$ per insertion; balanced trees and Fenwick trees (Chapter 4.2) handle that. Searching over real numbers needs separate stopping rules and is not covered here; every search in this chapter is over integers.

Binary search over sorted data also pairs well with earlier tools: sort first (Chapter 1.5), or binary search over prefix sums (Chapter 2.1), which are sorted whenever the values are positive.

Exercises

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

1. ● What should a lower-bound search return if every element is smaller than x ?
2. ● Which property must be proved before searching over an answer?
3. ● Trace the lower-bound loop on $[1, 3, 3, 3, 8, 9]$ with $x = 3$. Write (lo, hi, mid) for each step and the final answer.
4. ● Using lower and upper bounds, how many copies of 3 are in $[1, 3, 3, 3, 8, 9]$?
5. ●● Machines with periods 4 and 6 must make 7 items. Give a safe upper bound for the search and find the earliest time.
6. ●● What is the smallest capacity that splits $[5, 2, 4, 3, 6]$ into at most 2 consecutive groups? Give the groups.

Answers and explanations

Read these after attempting the exercises.

1. n , the boundary just after the last index. The program then reports “none” (0 in the practice problem).
2. That the yes/no answer changes in only one direction: once feasible, every larger candidate is feasible too.
3. (0, 6, 3): $a[3] = 3 \geq 3$, $hi = 3$. (0, 3, 1): $a[1] = 3 \geq 3$, $hi = 1$. (0, 1, 0): $a[0] = 1 < 3$, $lo = 1$. Answer index **1**, the first 3.
4. Lower bound 1, upper bound 4 (the first value > 3 is the 8 at index 4), so there are $4 - 1 = 3$ copies.
5. Upper bound $\min(t) \times k = 4 \times 7 = 28$. Items at time T are $\lfloor T/4 \rfloor + \lfloor T/6 \rfloor$: $T = 16$ gives $4 + 2 = 6$, $T = 18$ gives $4 + 3 = 7$, and $T = 17$ gives $4 + 2 = 6$. The earliest time is **18**.
6. The range is 6 to 20. Capacity 10 gives $5 \ 2 \mid 4 \ 3 \mid 6$, three groups; capacity 11 gives $5 \ 2 \ 4 \mid 3 \ 6$, two groups. The answer is **11**, with groups $[5, 2, 4]$ and $[3, 6]$.

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.

First qualifying position

Apply

For each x , find the first position in a nondecreasing array whose value is at least x . Positions are 1-based; print 0 if none exists.

Copies of a reading

Connect

For each x , report how many times it appears in a nondecreasing array.

Production deadline

Reason

There are n machines. Machine i produces one item every t_i time units, starting with zero items at time 0; machines work in parallel. Find the earliest integer time at which at least k items have been produced.

Contiguous load limit

Stretch

Split a positive array into at most k nonempty consecutive groups. Minimize the largest group sum. Every element must belong to exactly one group.