Binary Search

Binary Search — Complete C++ Guide (basic → advanced)

Overview (short)

Binary search is an algorithm to find a target value in a sorted domain by repeatedly halving the search interval. It's O(log n) time and is useful far beyond simple array lookups — it is also used for finding thresholds, optimizing monotonic functions, solving integer-continuous search problems ("binary search on answer"), and many competitive programming tasks.

---

1. Binary Search: Theory

1.1 Intuition

If an array `A` is sorted, and we need to find `x`, we can compare `x` with the middle element `A[mid]`:

- If `x == A[mid]` → found.
- If `x < A[mid]` → it can only be in left half.
- If `x > A[mid]` → it can only be in right half.

By discarding half the search space each step, we reach logarithmic time.

1.2 Complexity

- Time: O(log n) comparisons (base-2).
- Space: O(1) for iterative version; O(log n) recursion depth for recursive version.

1.3 Correctness outline (loop invariant)

Maintain an invariant: the target is always in the search interval `[low, high]` (or known to be absent). With careful updates ensuring the interval shrinks and invariant holds, termination happens when interval becomes size 0 (not found) or you return (found). Using `low <= high` inclusive intervals is common.

1.4 Pitfalls

- Integer overflow when computing `mid = (low + high)/2`. Use `low + (high - low) / 2`.
- Off-by-one errors with inclusive vs exclusive bounds.
- Infinite loops when updating `low`/`high` incorrectly (e.g. `low = mid` instead of `mid + 1`).
- Not handling duplicates when searching for first/last occurrence.

---

2. Basic C++ Implementations

2.1 Iterative binary search (classic)

Theory: inclusive interval `[l, r]`, while `l <= r` compute `mid`. Standard safe `mid` formula avoids overflow.

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

// Returns index of target in sorted array arr, or -1 if not found.
int binary_search_iterative(const vector<int>& arr, int target) {
    int l = 0, r = (int)arr.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2; // safe from overflow
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) l = mid + 1;
        else r = mid - 1;
    }
    return -1;
}

// Example usage:
int main() {
    vector<int> a = {1, 3, 4, 6, 8, 10};
    cout << binary_search_iterative(a, 6) << '\n'; // prints 3
    cout << binary_search_iterative(a, 2) << '\n'; // prints -1
}
```

Walkthrough:
Array = `[1,3,4,6,8,10]`, target = 6: `l=0,r=5,mid=2 -> arr[2]=4 < 6 => l=3` then `mid=3`, `arr[3]=6` found.

2.2 Recursive binary search

Theory: same invariant but expressed via recursive calls.

```cpp
int binary_search_recursive(const vector<int>& arr, int target, int l, int r) {
    if (l > r) return -1;
    int mid = l + (r - l) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] < target) return binary_search_recursive(arr, target, mid + 1, r);
    return binary_search_recursive(arr, target, l, mid - 1);
}
```

Recursion depth = O(log n). Use iterative in performance-critical contexts to avoid recursion overhead.

---

3. Variations and Practical Patterns

We now cover many practical binary-search variations used frequently.

3.1 `lower_bound` and `upper_bound` semantics

- `lower_bound(arr, x)`: first index `i` such that `arr[i] >= x`. (If all < x, returns n).
- `upper_bound(arr, x)`: first index `i` such that `arr[i] > x`. (If none, returns n).
  These are the building blocks for counting occurrences: `count = upper_bound - lower_bound`.

Implement lower_bound (iterative)

```
int lower_bound_idx(const vector<int>& arr, int x) {
    int l = 0, r = (int)arr.size(); // note: r is n (exclusive)
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] < x) l = mid + 1;
        else r = mid;
    }
    return l; // l in [0..n]
}
```

Why exclusive `r`? This idiom avoids some off-by-one errors and directly finds the first position where `arr[pos] >= x`.

Implement upper_bound

```
int upper_bound_idx(const vector<int>& arr, int x) {
    int l = 0, r = (int)arr.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] <= x) l = mid + 1;
        else r = mid;
    }
    return l;
}
```

Example: count occurrences

```
int count_occurrences(const vector<int>& arr, int x) {
    return upper_bound_idx(arr, x) - lower_bound_idx(arr, x);
}
```

3.2 First and Last Occurrence (in array with duplicates)

- To find **first** occurrence: use `lower_bound` logic but check equality.
- To find **last** occurrence: you can use `upper_bound - 1` if found.

```
int first_occurrence(const vector<int>& arr, int x) {
    int idx = lower_bound_idx(arr, x);
    if (idx < arr.size() && arr[idx] == x) return idx;
    return -1;
}

int last_occurrence(const vector<int>& arr, int x) {
    int idx = upper_bound_idx(arr, x) - 1;
    if (idx >= 0 && idx < arr.size() && arr[idx] == x) return idx;
    return -1;
}
```

Walkthrough: `arr=[1,2,2,2,3], x=2` → `lower_bound=1`, `upper_bound=4` → first=1, last=3.

3.3 Binary Search on Monotonic Boolean Functions (predicate)

This is the generalization: given predicate `P(x)` that is false, false, ..., true, true (monotonic), find smallest `x` where `P(x)` is true. Useful in parametric search.

Template:

```
// Find first x in [lo, hi] such that P(x) is true. Must have P monotonic.
long long first_true(long long lo, long long hi, function<bool(long long)> P) {
    while (lo < hi) {
        long long mid = lo + (hi - lo) / 2;
        if (P(mid)) hi = mid;
        else lo = mid + 1;
    }
    return lo; // caller must check P(lo)
}
```

Example: smallest `k` such that `k*k >= N` (integer square root ceiling): `P(k) = (k*k >= N)`.

3.4 Search in Rotated Sorted Array

Array is rotated — `arr = sorted array` rotated at some pivot. Example: `[4,5,6,7,0,1,2]`. Binary search variant uses comparisons to decide which half is sorted.

```cpp
int search_in_rotated(const vector<int>& nums, int target) {
    int l = 0, r = nums.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (nums[mid] == target) return mid;
        // left half is sorted
        if (nums[l] <= nums[mid]) {
            if (nums[l] <= target && target < nums[mid]) r = mid - 1;
            else l = mid + 1;
        } else { // right half is sorted
            if (nums[mid] < target && target <= nums[r]) l = mid + 1;
            else r = mid - 1;
        }
    }
    return -1;
}
```

Key idea: Determine which side is strictly sorted, then check if `target` lies in that range.

3.5 Binary Search on Answer (Parametric Search)

When the search domain is the answer space (e.g., maximize/minimize value meeting condition), formulate a boolean predicate `feasible(x)` and binary-search the smallest/largest feasible `x`.

Example: Given `n` students and `m` tasks with times, find minimum maximum time per student to finish tasks with contiguous assignment. `P(t)` = "can we assign tasks so each student has ≤ t time?" Binary-search `t`.

3.6 Binary Search on Floating Values

When searching continuous values, iterate fixed number of times or until interval length < epsilon.

```cpp
double binary_search_double(double lo, double hi, function<bool(double)> P, int
iterations=100) {
    for (int it = 0; it < iterations; ++it) {
        double mid = lo + (hi - lo) / 2.0;
        if (P(mid)) hi = mid;
        else lo = mid;
    }
    return lo; // or hi depending on invariants
}
```

Use-case: root-finding for monotonic function; find threshold where `f(x) >= target`.

3.7 Search in "Infinite" Sorted Array

If array length unknown but indexing supported, first exponentially increase `r` until `arr[r] >= target` or out-of-bounds, then binary search in `[l..r]`. For practical code with vector, you simulate via bounds-checking.

3.8 Binary Search on Answer — optimization problems

- Minimizing maximum distance, finding minimal days to finish work, capacity problems.
- Convert to monotonic `feasible(x)` and binary-search.

---

4. Proofs & Formal Reasoning (Sketches)

4.1 Proof of termination

Every iteration reduces the size of the search interval by at least one (for integers), because `l` becomes `mid+1` or `r` becomes `mid-1` (inclusive variant), and `mid` is between `l` and `r`. Thus interval integer length decreases → termination.

4.2 Correctness via invariant

Invariant: at each loop start, if the target exists it lies in `[l,r]`. Initially true. Updates maintain the invariant. If we exit with `l>r` the interval is empty → target absent.

---

5. Common Problem Patterns (with code & explanation)

I'll present patterns + code + example input & step-by-step.

Pattern A — Basic search

Problem: Find an element in sorted array.
Solution: standard iterative binary search (see earlier).

Pattern B — Find boundary (lower_bound/upper_bound)

Problem: Count occurrences of `x`.
Solution: `count = upper_bound(x) - lower_bound(x)`. Code above.

Pattern C — Smallest element ≥ x (lower_bound) — has code above.

Pattern D — First element greater than target (upper_bound) — has code above.

Pattern E — Find peak (bitonic array / mountain)

If array increases then decreases. Peak index found by comparing `mid` with neighbors — this is binary search on derivative.

```cpp
int peakIndexInMountainArray(const vector<int>& arr) {
    int l = 0, r = arr.size() - 1;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] < arr[mid + 1]) l = mid + 1;
        else r = mid;
    }
    return l;
}
```

Explanation: `arr[mid] < arr[mid+1]` means slope increasing → peak to right.

Pattern F — Find sqrt floor (integer)

Find `floor(sqrt(x))` using binary search.

```cpp
long long integer_sqrt(long long x) {
    long long l = 0, r = x;
    while (l <= r) {
        long long mid = l + (r - l) / 2;
        if (mid*mid == x) return mid;
        else if (mid*mid < x) l = mid + 1;
        else r = mid - 1;
    }
    return r; // r is floor(sqrt(x))
}
```

Careful: `mid*mid` can overflow for `long long` if `x` large — use `mid <= x / mid` check instead.

Pattern G — Search in 2D sorted matrix (each row sorted, and first element of row > last of previous row)

Flattened index binary search:

```cpp
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    if (matrix.empty()) return false;
    int rows = matrix.size();
    int cols = matrix[0].size();
    int l = 0, r = rows*cols - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        int val = matrix[mid / cols][mid % cols];
        if (val == target) return true;
        if (val < target) l = mid + 1;
        else r = mid - 1;
    }
    return false;
}
```

---

6. Advanced Techniques & Tips

6.1 Templates for reusability

A generic `binary_search_on_predicate`:

```cpp
template <typename T, typename Pred>
T binary_search_first(T lo, T hi, Pred p) {
    while (lo < hi) {
        T mid = lo + (hi - lo) / 2;
        if (p(mid)) hi = mid;
        else lo = mid + 1;
    }
    return lo;
}
```

`Pred` must represent monotonic predicate.

6.2 Avoiding overflow & precision issues

- Use `mid = l + (r - l) / 2`.
- For `mid*mid <= x` use `mid <= x / mid` to avoid overflow.
- For floating, iterate fixed times or check `hi - lo < eps`.

6.3 Choosing inclusive `[l,r]` vs half-open `[l,r)`

- Inclusive `[l,r]` often used for integer indices with `while (l <= r)`.
- Half-open `[l,r)` often used for `lower_bound` idiom (`while (l < r)`).

- Be consistent. Convert problems to one form and stick to that style.

6.4 Binary search on answer vs direct formula

If problem has monotonic property, prefer predicate approach. Example: scheduling, partitioning, feasibility checks.

6.5 Debugging tips

- Print `l, r, mid` during development on sample arrays.
- Use small arrays with duplicates and edge cases: empty, one element, two elements, all equal, strictly increasing.

---

7. Complete C++ File — Reference (many utilities)

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

int binary_search_iterative(const vector<int>& arr, int target) {
    int l = 0, r = (int)arr.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) l = mid + 1;
        else r = mid - 1;
    }
    return -1;
}

int lower_bound_idx(const vector<int>& arr, int x) {
    int l = 0, r = (int)arr.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] < x) l = mid + 1;
        else r = mid;
    }
    return l;
}

int upper_bound_idx(const vector<int>& arr, int x) {
    int l = 0, r = (int)arr.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
```

```cpp
        if (arr[mid] <= x) l = mid + 1;
        else r = mid;
    }
    return l;
}

int first_occurrence(const vector<int>& arr, int x) {
    int idx = lower_bound_idx(arr, x);
    if (idx < arr.size() && arr[idx] == x) return idx;
    return -1;
}

int last_occurrence(const vector<int>& arr, int x) {
    int idx = upper_bound_idx(arr, x) - 1;
    if (idx >= 0 && idx < (int)arr.size() && arr[idx] == x) return idx;
    return -1;
}

int search_in_rotated(const vector<int>& nums, int target) {
    int l = 0, r = nums.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (nums[mid] == target) return mid;
        if (nums[l] <= nums[mid]) {
            if (nums[l] <= target && target < nums[mid]) r = mid - 1;
            else l = mid + 1;
        } else {
            if (nums[mid] < target && target <= nums[r]) l = mid + 1;
            else r = mid - 1;
        }
    }
    return -1;
}

int peakIndexInMountainArray(const vector<int>& arr) {
    int l = 0, r = arr.size() - 1;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] < arr[mid + 1]) l = mid + 1;
        else r = mid;
    }
    return l;
}
```

```cpp
long long integer_sqrt(long long x) {
    long long l = 0, r = x;
    while (l <= r) {
        long long mid = l + (r - l) / 2;
        if (mid <= x / mid) l = mid + 1; // mid*mid <= x
        else r = mid - 1;
    }
    return r;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    vector<int> a = {1,2,2,2,3,4,6,8};
    cout << "binary_search_iterative(6) -> " << binary_search_iterative(a,6) <<
"\n";
    cout << "first_occurrence(2) -> " << first_occurrence(a,2) << "\n";
    cout << "last_occurrence(2) -> " << last_occurrence(a,2) << "\n";
    cout << "count_occurrences(2) -> " << (upper_bound_idx(a,2)-
lower_bound_idx(a,2)) << "\n";
    vector<int> rot = {4,5,6,7,0,1,2};
    cout << "search_in_rotated(0) -> " << search_in_rotated(rot,0) << "\n";
    vector<int> mountain = {0,2,4,7,5,3,1};
    cout << "peakIndexInMountainArray -> " << peakIndexInMountainArray(mountain) <<
"\n";
    cout << "integer_sqrt(17) -> " << integer_sqrt(17) << "\n";
    return 0;
}
```

---

8. Examples with step-by-step walkthroughs

Example 1: lower_bound on `arr=[1,3,4,6,8], x=5`

- l=0,r=5(mid computed over [0,5) scheme)
- mid=2 → arr[2]=4 < 5 → l=3
- mid=4 → arr[4]=8 >=5 → r=4
- mid=3 → arr[3]=6 >=5 → r=3
- now lr3 → end → lower_bound=3 → position to insert 5 to keep sorted.

Example 2: rotated array `arr=[6,7,1,2,3,4,5], target=3`

- l=0,r=6,mid=3 (arr[3]=2) → left half not strictly sorted? left (6..2) check: arr[l]=6>arr[mid]=2 → right half (mid..r) is sorted? arr[mid]=2<=arr[r]=5 -> right sorted. target 3 falls in [2,5] -> l = mid+1 = 4.
- Continue until found.

---

9. Common Mistakes and Fixes

- **Overflow**: use `l + (r - l) / 2`.
- **Equality mistake in `upper_bound` vs `lower_bound`**: check `<=` vs `<`.
- **Infinite loop**: if you do `l = mid` or `r = mid` with inclusive intervals, you may not shrink. Either use `mid+1`/`mid-1` or switch to half-open.
- **Bad predicate monotonicity**: when using predicate-based search, ensure monotonicity — otherwise binary search is invalid.

---

10. Useful Exercises (with hints / short approach)

1. **Find smallest element in rotated sorted array**: Use pivot detection: find index `i` with `arr[i] > arr[i+1]`. Edge cases small arrays.
   - Hint: Use `while(l<r)` with `if(arr[mid] > arr[r]) l = mid+1; else r = mid;`.
2. **Find peak in bitonic array**: see `peakIndexInMountainArray`.
3. **Kth smallest in two sorted arrays**: Use binary search on partition index. (Harder — classic median-of-two-arrays problem.)
4. **Minimum number of pages (Allocation Problem)**: Binary search on answer (max pages per student).
5. **Aggressive cows / placing routers**: Binary search maximum feasible minimum distance.
6. **Find first bad version**: classic predicate `isBadVersion(mid)`.
7. **Search a string in array of words padded with empty strings**: when mid points to empty, move mid to nearest non-empty neighbor.

---

11. Checklist for implementing binary search

- [ ] Decide inclusive or half-open interval and stick to it.
- [ ] Use safe `mid` computation.
- [ ] Write invariants as comments.
- [ ] Ensure updates shrink interval.
- [ ] Test on edge cases (empty, single element, all equal, target below/above).
- [ ] Consider duplicates if needed.
- [ ] Consider overflow for multiplications.

12. Final Advanced Examples (short)

12.1 Binary search to minimize maximum load (example)

Problem: Given tasks with times `t[i]` and `k` workers who can take contiguous tasks, minimize maximum time assigned.

Approach: `feasible(maxTime)` checks scanning tasks greedily whether we can split into ≤ k workers such that no worker sum > maxTime. Binary-search `maxTime` in range `[max(t), sum(t)]`.

12.2 Kth smallest in two sorted arrays (outline)

Binary search on how many elements to take from A (0..k) ensuring partition property using `A[i-1] <= B[j]` and `B[j-1] <= A[i]`. Edge-heavy but O(log n).

13. Quick Reference: Common snippets

- inclusive search:

```
while (l <= r) {
  mid = l + (r-l)/2;
  if (cond) return mid;
  else if (arr[mid] < target) l = mid + 1; else r = mid - 1;
}
```

- lower_bound:

```
while (l < r) {
  mid = l + (r-l)/2;
  if (arr[mid] < x) l = mid + 1;
  else r = mid;
}
return l;
```

- binary search on predicate (first true):

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

14. Exercises & Solutions (selected)

Exercise: Given sorted `arr`, find smallest element > x (strictly greater).
Solution: `idx = upper_bound_idx(arr, x); if idx == n -> none; else arr[idx]`.

Exercise: Find index of pivot (min element) in rotated sorted array with duplicates.
Idea: duplicates complicate: if `arr[mid] == arr[r]`, can't decide — do `r--` (linear worst-case). Otherwise proceed normally.

---

15. Recommended reading and next steps

- Study standard library: `std::lower_bound`, `std::upper_bound` (in `<algorithm>`).
- Practice problems: LeetCode binary search tag, SPOJ, Codeforces tasks with parametric search.
- Learn partition-based problems (median of two arrays) and Kth-element via selection algorithms.
