ChatGPT

Binary Search Deep Research for FAANG Interviews in JavaScript

Binary search overview

Binary search is a divide-and-conquer technique that repeatedly halves a search interval to locate a target (or a boundary) in a **sorted** or otherwise **monotonic** search space, achieving logarithmic time by eliminating half the remaining candidates each step. [1]

In FAANG-style interviews, binary search shows up far beyond "find `target` in sorted array." The interview move is to recognize when **the answer space is ordered** (indices, values, time, or a feasible/infeasible predicate) and then enforce an invariant while shrinking the search range. [2]

Binary search is especially good for: - Searching in **sorted arrays** and variants like "first $\geq x$" / "last $\leq x$". [3]
- Problems with an "**if X works, then any bigger X works**" (or the reverse) feasibility structure, often called searching on a **monotonic predicate** / "binary search on the answer." [4]
- "Almost sorted" structures like **rotated sorted arrays**, where you can still guarantee one half is sorted each step. [5]
- Partition-style problems like finding the median of two sorted arrays without merging, where you binary search a **partition index**. [6]

(These categories map directly onto the LeetCode set you listed.) [7]

A repeatable setup process

Here's a compact interview process (max five steps) that works across almost all binary-search problems:

Step A — Identify the ordered thing.
Are you searching **indices** in a sorted array? A **value range** (like speed/time/capacity)? A **time-ordered list**? A **partition index**? Binary search requires an order or a monotonic true/false predicate. [8]

Step B — Decide what you're returning.
- Exact index?
- Boolean exists?
- Minimum feasible value? (leftmost true)
- Maximum feasible value? (rightmost true)
This determines whether you implement `while (lo <= hi)` (exact search) or `while (lo < hi)` (boundary search). [9]

Step C — Define the invariant and predicate.
Write the key sentence:
- "The target, if it exists, is always in `[lo..hi]`." (exact search)
- "All values `< lo` are infeasible, all values `>= hi` are feasible." (first-true search)
This is the heart of correctness. [10]

Step D — Implement boundary updates carefully.
Pick `mid`, compare, and eliminate a half while preserving the invariant. Most bugs are "off-by-one" boundary mistakes, not concept mistakes. [11]

Step E — Validate edge cases with a tiny dry run.
Always validate: empty-ish structures, `k = 1`, `k = n`, target smaller/larger than all elements, and the "already sorted / not rotated" case for rotation problems. [12]

A quick warm-up example

Example array: `nums = [-1, 0, 3, 5, 9, 12]`, `target = 9` (classic index search). This is the canonical "binary search in sorted array" setting. [13]

Array being searched (indices shown):

```
idx:   0   1   2   3   4   5
nums: -1   0   3   5   9   12
```

Dry run (exact search style):

```
lo=0, hi=5
mid=2 -> nums[mid]=3  < 9  => lo = mid+1 = 3

lo=3, hi=5
mid=4 -> nums[mid]=9 == 9  => return 4
```

Logarithmic behavior comes from halving the remaining search interval each iteration. [14]

Core binary-search patterns you should know

You listed three core concepts; they're absolutely central. In FAANG interviews, there are really **four** "main" binary search patterns that recur (your three + the partition variant). [15]

Boundary search for a minimum or maximum

This is the “find a minimum element” / “leftmost true” / “lower_bound” family: - Find minimum feasible `x` where predicate becomes true. [10]
- Find first index `i` where `arr[i] >= target` (a standard `lower_bound`-style boundary). [16]

You’ll use this pattern in Koko (min feasible speed), in TimeMap (rightmost timestamp ≤ query time), and in rotated-minimum. [17]

Binary search on an answer range

Instead of searching an array, you search an integer range like `[1..max]` and use a monotonic feasibility predicate: - If `k` bananas/hour works, any `k' > k` also works (monotonic). [18]

This is the core idea in Koko Eating Bananas. [19]

Rotated sorted array reasoning

Rotation breaks global sortedness, but a critical property holds: **at any mid, at least one half is still sorted**, which lets you decide which half to discard. [5]

This supports: - Finding the minimum in a rotated sorted array (pivot/minimum). [20]
- Searching for a target in a rotated sorted array in `O(log n)`. [21]

Partition-based binary search across two arrays

This is the special “median of two sorted arrays” trick: binary search a partition in the smaller array so that the left partitions across both arrays contain exactly half the elements, and all left elements are ≤ all right elements. [6]

This is not the same as “rotated array,” but it *feels similar* because you’re searching for a “correct split point.” [22]

Step-by-step walkthroughs for the problem set

These problems are commonly practiced on entity["company","LeetCode","coding interview platform"]; I’ll treat the statements/constraints as given and focus on interview-ready reasoning and JavaScript implementations. [7]

Binary Search

Problem summary and difficulty

Given a sorted ascending array with unique elements, return the index of `target` or `-1` if not found; expected logarithmic time. The main difficulty is off-by-one errors and keeping consistent boundaries. [23]

What to identify and clarify before coding
Confirm: - Sorted ascending? (yes)
- Duplicates? (typically unique in this one)
- Return index or boolean? (index / `-1`) [23]

Apply the setup process
- Step A: Ordered thing = indices of a sorted array. [24]
- Step B: Return = exact index.
- Step C: Invariant = "if target exists, it's inside `[lo..hi]`."
- Step D: Update boundaries by comparing `nums[mid]`.
- Step E: Dry run a hit and a miss.

Step-by-step dry run (showing the array)
Example: `nums = [-1,0,3,5,9,12]`, `target = 2`

Array:

```
idx:   0   1   2   3   4    5
nums: -1   0   3   5   9   12
```

Iterations:

```
lo=0 hi=5 mid=2 nums[mid]=3  -> 3 > 2 => hi=1
lo=0 hi=1 mid=0 nums[mid]=-1 -> -1 < 2 => lo=1
lo=1 hi=1 mid=1 nums[mid]=0  -> 0 < 2 => lo=2
stop (lo=2 > hi=1) => return -1
```

JavaScript solution

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
function search(nums, target) {
  let lo = 0, hi = nums.length - 1;

  while (lo <= hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (nums[mid] === target) return mid;
    if (nums[mid] < target) lo = mid + 1;
    else hi = mid - 1;
  }
```

```
  return -1;
}
```

Time and space analysis
Time: `O(log n)` comparisons by halving the interval each iteration. [14]
Space: `O(1)` auxiliary space. [25]

Search a 2D Matrix

Problem summary and difficulty
You're given an `m x n` matrix where each row is sorted, and the first element of each row is greater than the last of the previous row. Return whether `target` exists, in `O(log(m*n))`. The main "aha" is to treat it as a single sorted 1D array via index mapping. [26]

What to identify and clarify before coding
Confirm: - Row-major global ordering property holds (first of row > last of previous). [27]
- Return boolean (`true/false`). [27]
- Constraints allow `m,n` up to around 100 in standard versions, so memory is not an issue; but you want logarithmic anyway. [28]

Apply the setup process
- Step A: Ordered thing = conceptual sorted 1D array of length `m*n`. [26]
- Step B: Return = boolean exists.
- Step C: Invariant = target, if present, lies within the conceptual index range.
- Step D: Map `mid` → `(r, c)` with `r = Math.floor(mid / n)`, `c = mid % n`.
- Step E: Dry-run mapping to ensure indices match.

Step-by-step dry run (showing the matrix and the conceptual array)
Example:

```
matrix =
[
  [ 1,  3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 60]
]
target = 16
m=3, n=4, total=12
```

Conceptual 1D order (indices 0..11):

```
idx:   0  1  2  3   4   5   6   7   8   9  10  11
val:   1  3  5  7  10  11  16  20  23  30  34  60
```

Iterations (binary search over idx range):

```
lo=0 hi=11 mid=5
mid=5 -> r=1 c=1 -> matrix[1][1]=11 < 16 => lo=6

lo=6 hi=11 mid=8
mid=8 -> r=2 c=0 -> matrix[2][0]=23 > 16 => hi=7

lo=6 hi=7 mid=6
mid=6 -> r=1 c=2 -> matrix[1][2]=16 == 16 => return true
```

JavaScript solution

```javascript
/**
 * @param {number[][]} matrix
 * @param {number} target
 * @return {boolean}
 */
function searchMatrix(matrix, target) {
  const m = matrix.length;
  if (m === 0) return false;
  const n = matrix[0].length;
  if (n === 0) return false;

  let lo = 0;
  let hi = m * n - 1;

  while (lo <= hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    const r = Math.floor(mid / n);
    const c = mid % n;
    const value = matrix[r][c];

    if (value === target) return true;
    if (value < target) lo = mid + 1;
    else hi = mid - 1;
  }
  return false;
}
```

Time and space analysis
Time: `O(log(m*n))` because you binary search the conceptual array. [29]
Space: `O(1)` auxiliary; you do not need to actually flatten. [27]

Koko Eating Bananas

Problem summary and difficulty
Find the minimum integer eating speed `k` such that Koko can eat all piles within `h` hours. The difficulty is that you are not searching an array index—you're searching an **answer range** with a feasibility function "canFinish(k)." [30]

What to identify and clarify before coding
Confirm: - `k` must be an integer and ≥ 1. [31]
- In one hour, she chooses one pile and eats up to `k` from it (so hours for a pile is `ceil(pile/k)`). [32]
- Monotonicity: if speed `k` works, any larger speed also works. [33]
- Bounds: `lo = 1`, `hi = max(piles)` is a safe upper bound (can always finish by eating the largest pile in one hour, and smaller piles in ≤ 1 hour each). [31]

Apply the setup process
- Step A: Ordered thing = integer speed range `[1..maxPile]`. [34]
- Step B: Return = minimum feasible `k` (boundary search / first true). [35]
- Step C: Predicate `canFinish(k)` is monotonic. [36]
- Step D: If feasible, move left; else move right.
- Step E: Dry run with small piles.

Step-by-step dry run (showing the piles and the searched "range")
Example: `piles = [3,6,7,11]`, `h = 8` is a standard illustrative case. [37]

Piles array (fixed throughout):

```
piles: [3, 6, 7, 11], h=8
```

Search space (implicit "array" of k values):

```
k in [1..11]
```

Feasibility: - `hours(k) = ceil(3/k)+ceil(6/k)+ceil(7/k)+ceil(11/k)`

Iterations:

```
lo=1 hi=11 mid=6
hours(6)=ceil(3/6)=1 + ceil(6/6)=1 + ceil(7/6)=2 + ceil(11/6)=2 => 6 hours
```

```
6 <= 8 => feasible => hi=6

lo=1 hi=6 mid=3
hours(3)=1 + 2 + 3 + 4 = 10
10 > 8 => infeasible => lo=4

lo=4 hi=6 mid=5
hours(5)=1 + 2 + 2 + 3 = 8
8 <= 8 => feasible => hi=5

lo=4 hi=5 mid=4
hours(4)=1 + 2 + 2 + 3 = 8
8 <= 8 => feasible => hi=4

stop when lo==hi==4 => answer 4
```

JavaScript solution

```javascript
/**
 * @param {number[]} piles
 * @param {number} h
 * @return {number}
 */
function minEatingSpeed(piles, h) {
  let lo = 1;
  let hi = 0;
  for (const p of piles) hi = Math.max(hi, p);

  function canFinish(k) {
    let hours = 0;
    for (const p of piles) {
      hours += Math.floor((p + k - 1) / k); // ceil(p/k) for ints
      if (hours > h) return false; // early exit
    }
    return hours <= h;
  }

  // first true in [lo..hi]
  while (lo < hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (canFinish(mid)) hi = mid;
    else lo = mid + 1;
  }
  return lo;
}
```

Time and space analysis
Time: `O(n log M)` where `n = piles.length` and `M = max(piles)`, because each feasibility check is `O(n)` and you do `O(log M)` checks. This is the standard complexity for "binary search on answer" with a linear predicate. [38]
Space: `O(1)` auxiliary. [39]

Find Minimum in Rotated Sorted Array

Problem summary and difficulty
Given a rotated sorted array of unique elements, return the minimum element in `O(log n)`. The difficulty is learning how to compare `nums[mid]` with an anchor (`nums[right]`) to decide which half contains the minimum. [20]

What to identify and clarify before coding
Confirm: - Array was originally sorted ascending and rotated. [40]
- Unique elements (no duplicates), which makes the comparisons decisive. [20]
- Return value (the min element), not its index. [41]

Apply the setup process
- Step A: Ordered structure is "two increasing runs"; the minimum is the pivot point. [20]
- Step B: Return = minimum value (boundary-ish search).
- Step C: Invariant: minimum is always in `[lo..hi]`.
- Step D: Compare `nums[mid]` to `nums[hi]`: - If `nums[mid] > nums[hi]`, min is strictly right of mid. - Else, min is at mid or left of mid. [20]
- Step E: Include "already sorted" case (no rotation), which returns `nums[0]`. [42]

Step-by-step dry run (showing the array)
Example: `nums = [4,5,6,7,0,1,2]`

Array:

```
idx:  0 1 2 3 4 5 6
nums: 4 5 6 7 0 1 2
```

Iterations:

```
lo=0 hi=6 mid=3 nums[mid]=7 nums[hi]=2
7 > 2 => min in (3..6] => lo=4

lo=4 hi=6 mid=5 nums[mid]=1 nums[hi]=2
1 <= 2 => min in [4..5] => hi=5

lo=4 hi=5 mid=4 nums[mid]=0 nums[hi]=1
0 <= 1 => min in [4..4] => hi=4
```

```
stop lo==hi==4 => nums[4]=0
```

JavaScript solution

```javascript
/**
 * @param {number[]} nums
 * @return {number}
 */
function findMin(nums) {
  let lo = 0, hi = nums.length - 1;

  // If already sorted (not rotated), first element is min.
  if (nums[lo] <= nums[hi]) return nums[lo];

  while (lo < hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (nums[mid] > nums[hi]) {
      lo = mid + 1;
    } else {
      hi = mid;
    }
  }
  return nums[lo];
}
```

Time and space analysis
Time: `O(log n)` by halving the interval each step, which is required by the problem. [43]
Space: `O(1)` auxiliary. [42]

Search in Rotated Sorted Array

Problem summary and difficulty
Return the index of `target` in a possibly rotated sorted array with unique values, or `-1`. The key difficulty is deciding which half is sorted and whether target lies inside it. [44]

What to identify and clarify before coding
Confirm: - Unique values (no duplicates) — simplifies logic. [45]
- Need `O(log n)` time. [45]
- Return index. [46]

Apply the setup process
- Step A: Ordered structure = at least one sorted half at each split. [47]
- Step B: Return = exact index (exact search).
- Step C: Invariant = target, if present, remains in `[lo..hi]`.

- Step D: Determine which side is sorted, then decide whether to keep that side. 5
- Step E: Dry run example where target is in the right “rotated” portion.

Step-by-step dry run (showing the array)
Example: `nums = [4,5,6,7,0,1,2]`, `target = 0`

Array:

```
idx:  0 1 2 3 4 5 6
nums: 4 5 6 7 0 1 2
```

Iterations:

```
lo=0 hi=6 mid=3 nums[mid]=7
Left half [4,5,6,7] is sorted (nums[lo] <= nums[mid])
Is target=0 in [nums[lo]=4 .. nums[mid]=7]? No
=> discard left => lo=mid+1=4

lo=4 hi=6 mid=5 nums[mid]=1
Left half [0,1] is sorted (nums[lo]=0 <= nums[mid]=1)
Is target=0 in [0..1]? Yes
=> keep left => hi=mid-1=4

lo=4 hi=4 mid=4 nums[mid]=0 => found => return 4
```

JavaScript solution

```javascript
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
function searchRotated(nums, target) {
  let lo = 0, hi = nums.length - 1;

  while (lo <= hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (nums[mid] === target) return mid;

    // Determine which half is sorted
    if (nums[lo] <= nums[mid]) {
      // Left half sorted
      if (nums[lo] <= target && target < nums[mid]) {
```

```
        hi = mid - 1;
      } else {
        lo = mid + 1;
      }
    } else {
      // Right half sorted
      if (nums[mid] < target && target <= nums[hi]) {
        lo = mid + 1;
      } else {
        hi = mid - 1;
      }
    }
  }
  return -1;
}
```

Time and space analysis
Time: `O(log n)` since each iteration discards a half. [48]
Space: `O(1)` auxiliary. [49]

Time Based Key Value Store

Problem summary and difficulty
Design a structure with: - `set(key, value, timestamp)`
- `get(key, timestamp)` returns the value with the **largest timestamp ≤ query timestamp** for that key, or `""` if none.

The key difficulty: you must combine a hashmap (key → list) with binary search over per-key timestamps. A crucial property is that timestamps for each key are strictly increasing, so the list stays sorted by insertion. [50]

What to identify and clarify before coding
Confirm: - Timestamps for `set` are strictly increasing per key (so append keeps order). [51]
- Need nearest value at or before time `t` (a rightmost-≤ boundary). [52]
- Constraints can include up to ~`2 * 10^5` operations, pushing you away from linear scan in `get`. [53]

Apply the setup process
- Step A: Ordered thing = per-key timestamp array (sorted). [54]
- Step B: Return = value at rightmost timestamp ≤ query (boundary search). [55]
- Step C: Predicate: “timestamp <= queryTime” is monotonic over sorted timestamps. [55]
- Step D: Implement `upperBound(queryTime) - 1`.
- Step E: Test: query earlier than first timestamp, between timestamps, and after last.

Step-by-step dry run (showing the array being searched)
Suppose for key `"foo"` you stored:

```
timestamps: [1, 4, 7]
values:     ["bar", "bar2", "bar3"]
query timestamp = 6
```

Binary search for rightmost timestamp ≤ 6:

Array being searched:

```
idx:        0  1  2
timestamps: 1  4  7
```

Iterations (upperBound style: first > 6, then subtract 1):

```
lo=0 hi=3 (note: hi is "exclusive" length=3)
mid=1 -> timestamps[1]=4 <= 6 => lo=2
lo=2 hi=3
mid=2 -> timestamps[2]=7  > 6 => hi=2
stop lo==hi==2 => upperBoundIndex = 2
answerIndex = upperBoundIndex - 1 = 1 => value "bar2"
```

JavaScript solution

```javascript
class TimeMap {
  constructor() {
    /** @type {Map<string, Array<[number, string]>>} */
    this.map = new Map(); // key -> list of [timestamp, value]
  }

  /**
   * @param {string} key
   * @param {string} value
   * @param {number} timestamp
   * @return {void}
   */
  set(key, value, timestamp) {
    if (!this.map.has(key)) this.map.set(key, []);
    this.map.get(key).push([timestamp, value]); // timestamps are increasing
  }

  /**
   * @param {string} key
   * @param {number} timestamp
```

```
   * @return {string}
   */
  get(key, timestamp) {
    const arr = this.map.get(key);
    if (!arr || arr.length === 0) return "";

    // Find first index i where arr[i][0] > timestamp (upper bound), then i-1 is
answer.
    let lo = 0;
    let hi = arr.length; // exclusive

    while (lo < hi) {
      const mid = lo + Math.floor((hi - lo) / 2);
      if (arr[mid][0] <= timestamp) lo = mid + 1;
      else hi = mid;
    }

    const idx = lo - 1;
    return idx >= 0 ? arr[idx][1] : "";
  }
}
```

Time and space analysis
- `set`: amortized `O(1)` append per call, relying on strictly increasing timestamps. [51]
- `get`: `O(log n)` per key-history length due to binary search. [56]
- Space: `O(total number of set calls)` to store all pairs. [57]

Median of Two Sorted Arrays

Problem summary and difficulty
Given two sorted arrays `nums1`, `nums2`, return the median of the combined multiset with overall time complexity `O(log(m+n))` (often implemented as `O(log(min(m,n)))`). The difficulty is that merging is too slow; instead, you binary search a partition in the smaller array to satisfy the ordering constraints across the partition. [58]

What to identify and clarify before coding
Confirm: - Arrays are sorted (can contain duplicates in general). [59]
- Total length may be odd or even; median definition changes accordingly. [60]
- Need sublinear merge-free approach: binary search partition. [61]

Apply the setup process
- Step A: Ordered structure = two sorted arrays; the “ordered thing” you search is the partition index `i` in the smaller array. [22]
- Step B: Return = median value (possibly average of two middle). [60]
- Step C: Predicate (correct partition):
`maxLeftA <= minRightB` AND `maxLeftB <= minRightA`. [62]

- Step D: If `maxLeftA > minRightB`, move partition left; else move right. [22]
- Step E: Dry run with a small pair.

Step-by-step dry run (showing both arrays and the partitions)
Example:

```
A = [1, 3, 8]
B = [7, 9, 10, 11]
Total = 7 => median is the 4th element (1-indexed) of merged order
```

We always binary search the shorter array `A`. [22]

We want left partition size:

```
leftSize = Math.floor((m + n + 1) / 2) = Math.floor((3 + 4 + 1)/2) = 4
```

Array visuals:

```
A: [1, 3, 8]
B: [7, 9, 10, 11]
```

Binary search on `i` = number of elements taken from A into the left half.

Iteration 1:

```
lo=0 hi=m=3
i=1 => A_left=[1] A_right=[3,8]
j=leftSize - i = 4 - 1 = 3 => B_left=[7,9,10] B_right=[11]

maxLeftA = 1
minRightA = 3
maxLeftB = 10
minRightB = 11

Check:
maxLeftA <= minRightB  => 1 <= 11 true
maxLeftB <= minRightA  => 10 <= 3 false  (B-left too big)
=> Need to take MORE from A (move i right): lo = i+1 = 2
```

Iteration 2:

```
lo=2 hi=3
i=2 => A_left=[1,3] A_right=[8]
j=4-2=2 => B_left=[7,9] B_right=[10,11]

maxLeftA = 3
minRightA = 8
maxLeftB = 9
minRightB = 10

Both conditions:
3 <= 10 true
9 <= 8 false
=> still need MORE from A => lo=3
```

Iteration 3:

```
lo=3 hi=3
i=3 => A_left=[1,3,8] A_right=[]
j=4-3=1 => B_left=[7] B_right=[9,10,11]

maxLeftA = 8
minRightA = +Infinity (A_right empty)
maxLeftB = 7
minRightB = 9

Both:
8 <= 9 true
7 <= +Inf true
=> Correct partition found
Odd total => median = max(maxLeftA, maxLeftB) = max(8,7)=8
```

JavaScript solution

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
function findMedianSortedArrays(nums1, nums2) {
  // Ensure nums1 is the smaller array
  if (nums1.length > nums2.length) return findMedianSortedArrays(nums2, nums1);

  const A = nums1, B = nums2;
```

```
  const m = A.length, n = B.length;

  let lo = 0, hi = m;
  const leftSize = Math.floor((m + n + 1) / 2);

  while (lo <= hi) {
    const i = lo + Math.floor((hi - lo) / 2);  // partition in A
    const j = leftSize - i;                    // partition in B

    const maxLeftA = (i === 0) ? -Infinity : A[i - 1];
    const minRightA = (i === m) ? Infinity : A[i];

    const maxLeftB = (j === 0) ? -Infinity : B[j - 1];
    const minRightB = (j === n) ? Infinity : B[j];

    if (maxLeftA <= minRightB && maxLeftB <= minRightA) {
      // Correct partition
      if ((m + n) % 2 === 1) {
        return Math.max(maxLeftA, maxLeftB);
      }
      return (Math.max(maxLeftA, maxLeftB) + Math.min(minRightA, minRightB)) /
2;
    }

    if (maxLeftA > minRightB) {
      hi = i - 1; // move left
    } else {
      lo = i + 1; // move right
    }
  }

  // Should never reach here if inputs are valid sorted arrays
  throw new Error("Invalid input");
}
```

Time and space analysis
Time: `O(log(min(m,n)))` because you binary search only the smaller array's partition index; this meets the intended logarithmic requirement without merging. [61]
Space: `O(1)` auxiliary. [22]

JavaScript binary search templates and pitfalls

Template for exact index search

Use this when you return an index (or `-1`) and the array is sorted. This is the classic pattern. [24]

```
function binarySearch(nums, target) {
  let lo = 0, hi = nums.length - 1;
  while (lo <= hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (nums[mid] === target) return mid;
    if (nums[mid] < target) lo = mid + 1;
    else hi = mid - 1;
  }
  return -1;
}
```

Template for first true in an integer range

Use this for “minimum feasible value” problems (Koko-style) and boundary searches. [10]

```
function firstTrue(lo, hi, isTrue) {
  while (lo < hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (isTrue(mid)) hi = mid;
    else lo = mid + 1;
  }
  return lo;
}
```

Template for upperBound in a sorted array

TimeMap is a classic “rightmost ≤ x”, which is commonly implemented as `upperBound(x) - 1` (first index > x). [63]

```
function upperBound(arr, x) { // first index i with arr[i] > x
  let lo = 0, hi = arr.length;
  while (lo < hi) {
    const mid = lo + Math.floor((hi - lo) / 2);
    if (arr[mid] <= x) lo = mid + 1;
    else hi = mid;
  }
  return lo;
}
```

Pitfall to watch in JavaScript mid calculation

Many people write `mid = (lo + hi) >> 1`. In JavaScript, bitwise operators coerce numbers into **32-bit integers**, which can be surprising outside typical LeetCode constraints. Using `Math.floor((hi - lo)/2)` avoids that coercion. [64]

Review and key concepts

Binary search is best thought of as an **invariant-maintaining boundary shrinking tool** over something ordered or monotonic, not just “search in a sorted array.” [65]

The key patterns to review for FAANG interviews, aligned to your list: - **Classic index binary search** (LC 704) and “flattened indexing” (LC 74): search indices in a sorted conceptual array. [66]
- **Binary search on answer** (LC 875): search an integer range using a monotonic feasibility predicate; return the minimum feasible value (first true). [67]
- **Rotated arrays** (LC 153, LC 33): exploit the fact that at least one half is sorted and (for min) compare against the right end to find the pivot. [68]
- **Time-ordered lists per key** (LC 981): maintain sorted timestamps by appending due to the strictly increasing constraint, then binary search the rightmost timestamp $\leq$ query. [69]
- **Partition binary search** (LC 4): binary search a partition index in the smaller array to satisfy cross-array ordering and compute the median without merging. [62]

A short checklist you can apply in interviews: - Name the ordered thing (indices, values, partition, time). [11]
- State your invariant out loud. [35]
- Decide if you need exact search (`lo <= hi`) or boundary search (`lo < hi`). [70]
- Dry run two iterations with the actual array/range drawn out (prevents 80% of off-by-one mistakes). [35]

---

[1] [3] [8] [11] [14] [24] [25] [48] [65] Binary Search
https://cp-algorithms.com/num_methods/binary_search.html?utm_source=chatgpt.com

[2] [4] [9] [10] [15] [33] [35] [38] [39] [55] [67] [70] Binary Search
https://usaco.guide/silver/binary-search?utm_source=chatgpt.com

[5] [12] [21] [45] [47] [49] [68] 33. Search in Rotated Sorted Array - In-Depth Explanation
https://algo.monster/liteproblems/33?utm_source=chatgpt.com

[6] [58] [59] doocs/leetcode - 0004.Median of Two Sorted Arrays
https://github.com/doocs/leetcode/blob/main/solution/0000-0099/0004.Median%20of%20Two%20Sorted%20Arrays/README_EN.md?utm_source=chatgpt.com

[7] [13] [23] [66] leetcode/solution/0700-0799/0704.Binary Search/ ...
https://github.com/doocs/leetcode/blob/main/solution/0700-0799/0704.Binary%20Search/README_EN.md?utm_source=chatgpt.com

[16] [63] std::lower_bound
https://en.cppreference.com/w/cpp/algorithm/lower_bound.html?utm_source=chatgpt.com

[17] [30] leetcode/solution/0800-0899/0875.Koko Eating Bananas ...
https://github.com/doocs/leetcode/blob/main/solution/0800-0899/0875.Koko%20Eating%20Bananas/README_EN.md?utm_source=chatgpt.com

[18] [36] Binary Search Intuition and Predicate Functions
https://www.geeksforgeeks.org/dsa/binary-search-intuition-and-predicate-functions/?utm_source=chatgpt.com

[19] [31] [32] [34] [37] 875. Koko Eating Bananas - In-Depth Explanation
https://algo.monster/liteproblems/875?utm_source=chatgpt.com

20 42 43 153. Find Minimum in Rotated Sorted Array
https://algo.monster/liteproblems/153?utm_source=chatgpt.com

22 61 62 Leetcode 4. Median of Two Sorted Arrays (HARD)
https://medium.com/%40marcinduchinski/leetcode-4-median-of-two-sorted-arrays-5b34385c09b5?utm_source=chatgpt.com

26 27 29 74. Search a 2D Matrix - LeetCode Wiki
https://doocs.github.io/leetcode/en/lc/74/?utm_source=chatgpt.com

28 74. Search a 2D Matrix - In-Depth Explanation
https://algo.monster/liteproblems/74?utm_source=chatgpt.com

40 doocs/leetcode - 0153.Find Minimum in Rotated Sorted Array
https://github.com/doocs/leetcode/blob/main/solution/0100-0199/0153.Find%20Minimum%20in%20Rotated%20Sorted%20Array/README_EN.md?utm_source=chatgpt.com

41 Find Minimum in Rotated Sorted Array
https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/?utm_source=chatgpt.com

44 46 doocs/leetcode - 0033.Search in Rotated Sorted Array
https://github.com/doocs/leetcode/blob/main/solution/0000-0099/0033.Search%20in%20Rotated%20Sorted%20Array/README_EN.md?utm_source=chatgpt.com

50 52 54 981. Time Based Key Value Store - Solution & Explanation
https://neetcode.io/solutions/time-based-key-value-store?utm_source=chatgpt.com

51 69 Time Based Key-Value Store
https://neetcode.io/problems/time-based-key-value-store/question?utm_source=chatgpt.com

53 LeetCode - 150 - Time Based Key-Value Store - Dmytro's Blog
https://dmytros.blog/posts/leetcode-150-time-based-key-value-store/?utm_source=chatgpt.com

56 57 981. Time Based Key-Value Store - LeetCode Solutions
https://walkccc.me/LeetCode/problems/981/?utm_source=chatgpt.com

60 4. Median of Two Sorted Arrays - In-Depth Explanation
https://algo.monster/liteproblems/4?utm_source=chatgpt.com

64 Bitwise AND (&) - JavaScript - MDN Web Docs
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_AND?utm_source=chatgpt.com