Q0: Binary Search

---

1. Problem Understanding

- Binary Search is an algorithm used to efficiently search for a value in a sorted array or in any monotonic search space.
- It works by repeatedly dividing the search interval into half.
- It applies to:
    - Searching for an element
    - Finding first/last occurrence
    - Finding boundaries
    - Solving questions where answer lies in a range
    - Questions where the condition behaves monotonically (true/false pattern)
- Binary search is NOT just for arrays â€" it's for searching on answers (Binary Search on Answer).

---

2. Constraints

- Array must be sorted (or monotonic condition must exist).

- Monotonic means the values move only in one direction (only increasing or only decreasing). Nothing zig-zag.

- Time limit often requires O(log n) or O(n log n) solutions.

- Values may be very large, but binary search handles it easily due to logarithmic complexity.

- **What is a Monotonic Array?**

    - An array is monotonic if it satisfies one of the following:
    - 1ï¸□âƒ£ Monotonic Increasing
        - Each element is greater than or equal to the previous.
            - Example:
                - 1 2 2 3 4 7 9
            - Here:
                - arr[i] <= arr[i+1]
    - 2ï¸□âƒ£ Monotonic Decreasing
        - Each element is less than or equal to the previous.
        - Example:
            - 9 7 7 4 3 1
        - Here:
            - arr[i] >= arr[i+1]
    - âœ"ï¸□ Why Binary Search Needs a Monotonic Condition?
        - Binary search works only if we can discard half of the search space every time.
        - That is possible only if:
            - Values consistently increase OR

        - Values consistently decrease
        - So we know which direction contains the target.
        - If array is like:
            - 3 5 2 8 1
            - Not sorted
            - Not monotonic
            - Values go up and down â†' cannot apply binary search

---

3. Edge Cases

- These are the edge cases that break 80% of beginners' code:
    - mid = (l + r) / 2 â†' may overflow; use
    - mid = l + (r - l) / 2.
    - Infinite loop due to wrong update (like using l = mid instead of l = mid + 1).
    - Using wrong comparison (< vs <=).
    - Arrays with duplicate elements (must choose left/right boundary carefully).
    - l surpassing r (stop condition).
    - Off-by-one errors in returning the boundary.
    - When doing binary search on answers â†' ensure the predicate is monotonic.

---

4. Examples

```
ðŸ§ª Examples (Simple)

Searching 7 in [1,3,4,7,9]
Sorted â†' can apply binary search.

Finding floor square root of 40 â†' answer lies in integer range [0..40].

Finding minimum capacity to ship packages â†' monotonic condition exists.
```

---

5. Approaches

Approach 1: Classic Binary Search on Sorted Array

Idea:

- Search for exact element.

Java Code:

```
public static int binarySearch(int[] arr, int target) {
    int l = 0, r = arr.length - 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;
    }
```

ðŸ' Intuition Behind the Approach:

- Each comparison eliminates half the array â†' logarithmic speed.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(log n) because we cut search space by half every iteration.
- ðŸ'¾ Space Complexity
  - O(1) iterative.

Approach 2: Lower Bound (first index â‰¥ target)

Idea:

- Find the first position where element is >= target.

Java Code:

```
public static int lowerBound(int[] arr, int target) {
    int l = 0, r = arr.length - 1;
    int ans = arr.length;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] >= target) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}
```

Approach 3: Upper Bound (first index > target)

Java Code:

```
public static int upperBound(int[] arr, int target) {
    int l = 0, r = arr.length - 1;
    int ans = arr.length;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] > target) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}
```

Approach 4: Search Insert Position

Java Code:

```java
public static int searchInsert(int[] arr, int target) {
    int l = 0, r = arr.length - 1;
    int ans = arr.length;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] >= target) {
            ans = mid;
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return ans;
}
```

Approach 5: Binary Search on Answer (VERY IMPORTANT)

Idea:

- Used in many DSA problems:
    - âœ” Allocate minimum pages
    - âœ” Aggressive cows
    - âœ” Koko eating bananas
    - âœ” Painters partition
    - âœ” Minimum capacity to ship packages
    - âœ” Find smallest divisor
    - âœ” Minimize maximum distance to gas station
    - âœ” etc.
- âœ” Idea
    - Guess an answer â†' check if it's valid â†' move left or right.
    - The key is the predicate must be monotonic:
    - If x works â†' all values > x might work
- Or vice versa

Java Code:

```java
public static int searchAnswer(int low, int high) {
    int ans = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (isPossible(mid)) {
            ans = mid;      // mid is valid â†' try left
            high = mid - 1;
        } else {
            low = mid + 1;  // mid invalid â†' go right
        }
    }
    return ans;
}
```

Approach 6: Binary Search on Real Numbers

Idea:

- Used for:
  - Square root with precision
  - Koko eating bananas (if using double)
  - Gas station placement
- Key difference:
  - Loop until precision achieved instead of l <= r.

Approach 7: Binary Search on Rotated Sorted Array

Idea:

- âœ" Search in rotated sorted array
- âœ" Find minimum in rotated sorted array
- Concept:
- Left half or right half is always sorted â€" determine where target lies.

Approach 8: Binary Search with Duplicates

Idea:

- First occurrence
- Last occurrence
- Count occurrences using:
- last - first + 1
- Requires adjusting boundaries carefully.

---

6. Variants / Follow-Ups

- Floor
- Ceil
- Square root
- Cube root
- kth missing positive
- Peak element
- Allocate minimum pages
- Maximum average subarray (variant)
- Aggressive cows
- Minimum time to complete tasks
- Koko eating bananas
- Search in mountain array

---

7. Tips & Observations

- Always check monotonicity before applying binary search.
- For duplicates â†’ use lower/upper bound logic.
- For rotated arrays â†’ identify sorted half.
- For BS on answer â†’ define the range properly.
- Watch for infinite loops by using mid = l + (r - l) / 2.
- Always test with edge cases: single element, all equal, target not found, target smaller/larger than all.
- **Complexity**
    - â□±ï¸□ Time Complexity (Final)
        - All array-based BS â†’ O(log n)
        - BS on answer â†’ O(log(range))
    - ðŸ’¾ Space Complexity (Final)
        - O(1) for iterative.

Q135: Find Minimum in Rotated Sorted Array

---

1. Problem Understanding

- You are given a sorted array that has been rotated between 1 and n times.
- Examples:
- Original: [1,2,3,4,5]
- Rotated: [3,4,5,1,2]
- Your task:

- Find the minimum element in the rotated array.
- Must run in O(log n) â†’ Binary Search.

---

2. Constraints

- 1 <= n <= 5000
- Numbers can be negative.
- All elements are unique
- Must use binary search â†’ no linear scan.

---

3. Edge Cases

- Array not rotated â†’ minimum at index 0.
- Example: [1,2,3,4]
- Rotation by n times also gives original array.
- Minimum might be at the boundaries.
- Example: [2,3,4,5,1]
- Very small arrays: size 1, size 2.
- Already sorted means return nums[0].

---

4. Examples

```
Example 1

Input:

[3,4,5,1,2]


Output:

1

Example 2

Input:

[4,5,6,7,0,1,2]


Output:

0
```

---

5. Approaches

Approach 1: Brute Force (Linear Scan)

Idea:

- The minimum of any array is simply the smallest element.
- Directly scan the array.

ðŸ’ Intuition Behind the Approach:

- A sorted rotated array still contains a minimum somewhere.
- You inspect all elements to find it.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n)
  - Because scanning all elements.
- ðŸ’¾ Space Complexity
  - O(1)
  - Only one variable used.

Approach 2: Slightly Better (Check Adjacent Break Point)

Idea:

- In a rotated array, the minimum occurs at the break point where:
- nums[i] < nums[i-1]
- Scan and check only such break.

Java Code:

```java
public static int findMin(int[] nums) {
    int n = nums.length;
    for (int i = 1; i < n; i++) {
        if (nums[i] < nums[i - 1]) return nums[i];
    }
    return nums[0];
}
```

ðŸ’ Intuition Behind the Approach:

- The array is sorted except one pivot point.
- Minimum sits right after the point where order breaks.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n) worst case
  - If array not rotated.
- ðŸ’¾ Space Complexity

    - O(1)

Approach 3: Binary Search â€” O(log n)

Idea:

- Use binary search to find the pivot where rotation happened.
- Observations:
    - If nums[mid] > nums[right], the minimum is to the right.
    - Else â†' minimum is to the left or at mid.

Steps:

- Set l = 0, r = n-1.
- While l < r:
    - Compute mid.
    - If nums[mid] > nums[r] â†' search right half.
    - Else â†' search left half (including mid).
- At the end l = index of minimum.

Java Code:

```java
public static int findMin(int[] nums) {
    int l = 0, r = nums.length - 1;

    while (l < r) {
        int mid = l + (r - l) / 2;

        if (nums[mid] > nums[r]) {
            l = mid + 1;     // min is to the right
        } else {
            r = mid;         // min is at mid or left side
        }
    }

    return nums[l]; // l is the index of minimum
}
```

ðŸ' Intuition Behind the Approach:

- When nums[mid] > nums[r], right half is unsorted â†' min lies there.
- When nums[mid] <= nums[r], right half is sorted â†' min cannot be in sorted region except possibly mid.
- Binary search narrows search space towards the pivot point.
- This is a classic Binary Search on answer space, not on presence.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(log n)

    - Because each iteration halves the search space.
- ðŸ'¾ Space Complexity
    - O(1)
    - Only two pointers.

---

6. Justification / Proof of Optimality

- The array is partially sorted.
- Binary search is the perfect fit when:
    - There is a monotonic pattern.
    - There is a pivot.
    - We can eliminate half of the array on each decision.
- The condition nums[mid] > nums[r] is the key that tells us which side the pivot lies on.

---

7. Variants / Follow-Ups

- Find index of minimum instead of value.
- Find number of times rotated â†' index of min is the rotation count.
- Rotated array with duplicates â†' trickier binary search.
- Search element in rotated sorted array.

---

8. Tips & Observations

- Always compare with the rightmost element for easier logic.
- If whole array is already sorted â†' return nums[0].
- If duplicates exist:
    - Need to shrink boundaries carefully.
- Rotated sorted array problems ALWAYS revolve around:
    - Detecting pivot
    - Searching on monotonic segments

Q136: Square root of a number

---

1. Problem Understanding

- You are given a number x.
- You must return:
- If x is a perfect square â†' its exact square root
- Otherwise â†' floor(âˆšx)
- You must not use the built-in sqrt() function.
- Time complexity must be O(log N), which hints binary search.

---

2. Constraints

- 1 â‰¤ x â‰¤ 10^7
- Input is a single integer
- Output must be an integer (floor value)

---

3. Edge Cases

- x = 1 â†' answer = 1
- Very small values: x = 2, 3 â†' answer = 1
- Large inputs up to 10^7
- Perfect squares: 4, 9, 16, 25 â†' return exact root
- Overflow of mid * mid â†' use long to prevent overflow

---

4. Examples

```
Example 1
Input: 5
Output: 2

Example 2
Input: 36
Output: 6

Example 3
Input: 2
Output: 1
```

---

5. Approaches

Approach 1: Brute Force Approach (Linear Search)

Idea:

- Try every number from 1 to x and check the last i such that i*i â‰¤ x.

Steps:

- Loop from i=1 to i*i â‰¤ x.
- Track the last valid number.
- Return it.

Java Code:

```
public static int mySqrt(int x) {
    int ans = 0;
```

```
        for (long i = 1; i * i <= x; i++) {
            ans = (int) i;
        }
        return ans;
    }
```

ðŸ' Intuition Behind the Approach:

- Directly simulate what square root means.
- Simple but slow.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(âˆšN)
  - Because loop runs until i*i â‰¤ x.
- ðŸ'¾ Space Complexity
  - O(1)
  - Only counters used.

Approach 2: Using Math Properties (Decreasing Search Range)

Idea:

- Instead of checking all numbers from 1 to x, we can optimize by stopping early once i*i crosses x.
- (Slightly better than brute but still worst-case âˆšN)

Steps:

- Same as brute but break early.

Java Code:

```
    public static int mySqrt(int x) {
        long i = 1;
        while (i * i <= x) i++;
        return (int)(i - 1);
    }
```

ðŸ' Intuition Behind the Approach:

- Stop as soon as you've gone past the root.
- Still linear in âˆšN but less work than brute.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(âˆšN)
- ðŸ'¾ Space Complexity
  - O(1)

Approach 3: Binary Search (Required O(log N))

Idea:

- The real square root lies between 1 and x.
- Binary search this space for the greatest mid such that:
- mid * mid â‰¤ x

Steps:

- Let low = 1, high = x
- Compute mid
- If mid*mid â‰¤ x, store it and move right (search for bigger)
- Else move left
- Return last stored answer

Java Code:

```
public static int mySqrt(int x) {
    if (x == 0) return 0;

    long s = 1;
    long e = x;
    long ans = 1;

    while (s <= e) {
        long mid = (s + e) / 2;

        if (mid * mid <= x) {
            ans = mid;      // mid is valid
            s = mid + 1;    // try for bigger
        } else {
            e = mid - 1;    // mid too big
        }
    }
    return (int) ans;
}
```

ðŸ’ Intuition Behind the Approach:

- Square root grows slowly, so binary search reduces the search space by half every step.
- We want the largest number whose square is â‰¤ x â†' therefore move right on valid mid.
- This ensures correctness for both perfect and non-perfect squares.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(log N)
  - Because binary search repeatedly halves the range.
- ðŸ'¾ Space Complexity

    - O(1)
    - Only variables used.

---

6. Justification / Proof of Optimality

- Binary search is optimal because the search range is monotonic (i*i increases as i increases).
- Unlike brute force (âˆšN steps), binary search does it extremely fast in log N steps.
- Works well for values up to 10^7 easily within constraints.

---

7. Variants / Follow-Ups

- Calculate ceil of âˆšx
- Return true/false if x is a perfect square
- Return âˆšx without using multiplication (Newtonâ€™s Method)
- âˆšx for long or big integers

---

8. Tips & Observations

- Always use long for mid * mid to avoid integer overflow.
- When searching for largest valid, use:
    - if (mid*mid <= x) â†’ move right
- For a perfect square, binary search will naturally find it.
- Classic question in binary search category â€” must be mastered.

Q137: Search A 2D Matrix

---

1. Problem Understanding

- You are given a matrix where:
    - Each row is sorted from left â†’ right.
    - The first element of each row is greater than the last element of previous row.
    - This means the entire matrix behaves like one sorted array.
- Goal:
    - Search if x exists.
    - Return true / false.

---

2. Constraints

- 1 <= m, n <= 1000
- m * n â‰¤ 10^6 â†’ binary search is perfect.
- Values range from -10^4 to 10^4.

---

3. Edge Cases

- x smaller than smallest element â†' false
- x larger than largest element â†' false
- 1Ã—1 matrix
- Single row or single column
- Large matrix â†' must use O(log(m*n))

---

4. Examples

```
Example 1
Matrix:
1  3  5  7
10 11 16 20
23 30 34 60
x = 10 â†' true

Example 2
x = 12 â†' false
```

---

5. Approaches

Approach 1: Brute Force (Check Every Element)

Idea:

- Scan all rows and all columns until element is found.

Java Code:

```java
public static boolean searchMatrix(int[][] mat, int x) {
    int m = mat.length, n = mat[0].length;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (mat[i][j] == x) return true;
        }
    }
    return false;
}
```

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(m*n) â€" checks every cell
  - Why: No skipping possible.

- ðŸ'¾ Space Complexity
    - O(1) â€" no extra space

Approach 2: Row Selection + Binary Search (Better)

Idea:

- Identify which row x could be in:
    - Check first/last elements of each row.
- Binary search that row.

Java Code:

```java
public static boolean searchMatrix(int[][] mat, int x) {
    int m = mat.length, n = mat[0].length;

    for (int i = 0; i < m; i++) {
        if (x >= mat[i][0] && x <= mat[i][n-1]) {
            int l = 0, r = n-1;
            while (l <= r) {
                int mid = l + (r - l) / 2;
                if (mat[i][mid] == x) return true;
                else if (mat[i][mid] < x) l = mid + 1;
                else r = mid - 1;
            }
            return false;
        }
    }
    return false;
}
```

ðŸ' Intuition Behind the Approach:

- Each row is sorted, so we can binary search inside the correct row.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - Worst-case: O(m + log n)
    - Why: Scan rows â†' then binary search inside one row.
- ðŸ'¾ Space Complexity
    - O(1)

Approach 3: Treat as Sorted 1D Array (Optimal â€" O(log(m*n)))

Idea:

- Because:
    - Last element of row i < first element of row i+1
    - Matrix behaves like a single sorted array of length m*n.

- Map 1D index to matrix index:
  - row = mid / n
  - col = mid % n
- Binary search on 1D indexed search space.

Java Code:

```
public static boolean searchMatrix(int[][] mat, int x) {
    int m = mat.length, n = mat[0].length;

    int l = 0, r = m * n - 1;

    while (l <= r) {
        int mid = l + (r - l) / 2;

        int row = mid / n;
        int col = mid % n;

        if (mat[row][col] == x) return true;
        else if (mat[row][col] < x) l = mid + 1;
        else r = mid - 1;
    }

    return false;
}
```

ðŸ’ Intuition Behind the Approach:

- Youâ€™re performing binary search on a flattened version of the matrix.
- Because matrix is globally sorted, this works flawlessly.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(log(m*n))
  - Why: Classic binary search on total number of elements.
- ðŸ'¾ Space Complexity
  - O(1) â€" constant space.

---

6. Justification / Proof of Optimality

- Brute force is too slow for large m*n.
- Row-wise selection reduces search but still touches many rows.
- Flattened binary search guarantees minimal comparisons and utilizes full sorted property â€" best approach.

---

7. Variants / Follow-Ups

- Search in a matrix where each row AND column is sorted (different problem, different approach).
- Find the number of occurrences of x.
- Return position instead of boolean.

---

8. Tips & Observations

- Whenever a matrix has this row-start > previous-row-end pattern â†’ treat like 1D sorted list.
- Binary search remains the strongest pattern if global monotonicity exists.

Q138: Find First and Last Position of Element in Sorted Array

---

1. Problem Understanding

- Given a sorted (non-decreasing) array nums, find the first and last index of a given target.
- If target doesnâ€™t appear, return [-1, -1].
- We must do it in O(log n), so binary search is required.

---

2. Constraints

- Array length: 0 to 1e5
- Values range: -1e9 to 1e9
- Array is sorted
- Must achieve O(log n)

---

3. Edge Cases

- n = 0 â†’ return [-1, -1]
- Target is smaller than first element
- Target is greater than last element
- All elements are the same as target â†’ output [0, n-1]
- Target appears once
- Target appears multiple times

---

4. Examples

```
Example 1

Input

6 8
```

```
5 7 7 8 8 10


Output

3 4

Example 2

Input

6 6
5 7 7 8 8 10


Output

-1 -1
```

---

5. Approaches

Approach 1: Brute Force (Linear Scan)

Idea:

- Iterate whole array once to find first occurrence and last occurrence.

Steps:

- Loop from left to right â†' find first match
- Loop from right to left â†' find last match

Java Code:

```
public int[] searchRange(int[] nums, int target) {
    int first = -1, last = -1;

    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == target) {
            first = i;
            break;
        }
    }

    for (int i = nums.length - 1; i >= 0; i--) {
        if (nums[i] == target) {
            last = i;
            break;
        }
    }
```

```
        return new int[]{first, last};
    }
```

ðŸ’ Intuition Behind the Approach:

- We check the entire array because itâ€™s unspecific where target may occur.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(n) because we scan the array twice.
- ðŸ’¾ Space Complexity
    - O(1) because no extra space.

Approach 2: Two Binary Searches (Optimal)

Idea:

- Use binary search twice:
    - once to find the first occurrence
    - once to find the last occurrence
- Because array is sorted, binary search ensures O(log n).

Steps:

- Steps
- Find first position
    - Normal binary search
    - When you find nums[mid] == target, move left to see if earlier occurrence exists
    - So set high = mid - 1
- Find last position
    - When nums[mid] == target, move right
    - So set low = mid + 1
- Return [first, last]

Java Code:

```java
public int[] searchRange(int[] nums, int target) {
    int first = findFirst(nums, target);
    int last = findLast(nums, target);
    return new int[]{first, last};
}

private int findFirst(int[] nums, int target) {
    int low = 0, high = nums.length - 1, ans = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (nums[mid] == target) {
            ans = mid;
```

```
                high = mid - 1; // go left
            } else if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return ans;
    }

    private int findLast(int[] nums, int target) {
        int low = 0, high = nums.length - 1, ans = -1;
        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] == target) {
                ans = mid;
                low = mid + 1; // go right
            } else if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return ans;
    }
```

ðŸ' Intuition Behind the Approach:

- Binary search normally stops when the target is found, but we modify it:
  - For first occurrence, we force the search to continue left until no earlier index contains the target.
  - For last occurrence, we force search right.
- This preserves the O(log n) speed but still finds boundaries.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - Two binary searches â†' O(log n)
  - Why?
    - Each binary search halves the array repeatedly, so two searches still log n + log n = log n.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Only variables used; no extra data structures.

---

6. Justification / Proof of Optimality

- Binary search ensures fastest possible runtime (O(log n)), and by slightly modifying conditions, we can locate the exact boundaries of the target.

---

7. Variants / Follow-Ups

- Find first occurrence only
- Find last occurrence only
- Count occurrences â†’ last - first + 1
- Works for rotated sorted arrays with modifications

---

8. Tips & Observations

- Always prefer binary search if array is sorted and you need positions.
- Use the mid calculation trick:
- mid = low + (high - low) / 2 to avoid overflow.
- Searching for first â†’ move left
- Searching for last â†’ move right

Q139: Count 1 in sorted binary array

---

1. Problem Understanding

- You are given a binary, sorted in non-increasing order array:
  - 1 1 1 ... 1 0 0 0
- You must find the count of 1s.
- Because all 1s come first and all 0s later, the problem becomes finding the boundary between 1 â†’ 0.
- Binary search helps find this boundary in O(log n).

---

2. Constraints

- 1 <= N <= 10^6
- Array contains only 0 and 1
- Array is sorted in non-increasing order
- Must be efficient (binary search recommended)

---

3. Edge Cases

- All 1s â†’ count = N
- All 0s â†’ count = 0
- Only one element
- Transition happens at index 0
- Transition happens at index N-1

---

4. Examples

```
Example 1

Input

8
1 1 1 1 1 0 0 0


Output

5

Example 2

Input

4
1 1 1 1


Output

4
```

---

5. Approaches

Approach 1: Linear Scan (Brute Force)

Idea:

- Traverse from the left and count until you see the first 0.

Steps:

- Initialize count = 0
- Loop from left to right
- If element is 1, increment
- If element is 0, break immediately
- Return count

Java Code:

```java
public int countOnes(int[] arr) {
    int count = 0;
    for (int x : arr) {
        if (x == 1) count++;
        else break;
    }
    return count;
}
```

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(k) where k = number of initial 1s
  - Worst case â†' O(n)
  - Why?
    - You may traverse the entire array if all values are 1.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - No extra memory used.

Approach 2: Binary Search for First 0 (Better)

Idea:

- Count of 1s = index of the first 0.
- Example:
- [1 1 1 0 0] â†' first 0 at index 3 â†' count = 3.

Steps:

- Binary search to find the first index where arr[mid] == 0.
- If no 0 exists â†' return N.
- Else return that index.

Java Code:

```java
public int countOnes(int[] arr) {
    int low = 0, high = arr.length - 1;
    int firstZero = arr.length;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == 0) {
            firstZero = mid;
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }

    return firstZero;
}
```

ðŸ’ Intuition Behind the Approach:

- The first 0 divides the array into:
- [1s] [0s]
- Finding that boundary gives us the count of 1s.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(log n)
  - Because each step halves the search space.
- ðŸ’¾ Space Complexity
  - O(1)
  - Only a few variables used.

Approach 3: Binary Search for Last 1 (Optimal)

Idea:

- Find last index where arr[mid] == 1, then count = mid + 1

Steps:

- Steps
- Binary search over array.
- If arr[mid] == 1, record index and move right to find last 1.
- If arr[mid] == 0, move left.
- At end â†'
  - if last1 = -1, return 0
  - else return last1 + 1

Java Code:

```java
public int countOnes(int[] arr) {
    int low = 0, high = arr.length - 1;
    int lastOne = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == 1) {
            lastOne = mid;
            low = mid + 1; // search right
        } else {
            high = mid - 1;
        }
    }

    return lastOne + 1;
}
```

ðŸ' Intuition Behind the Approach:

- The array looks like this:
  - 1 1 1 â€¦ 1 | 0 0 0
- Binary search targets the rightmost 1 efficiently.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(log n)
  - Boundary search using binary search.
- ðŸ'¾ Space Complexity
  - O(1)

---

6. Justification / Proof of Optimality

- Binary search works perfectly because the array has a monotonic pattern: all 1s first, then all 0s.
- Finding the transition boundary is the fastest way to determine count.

---

7. Variants / Follow-Ups

- Find first 1
- Find first 0 in increasing binary array
- Count zeros
- First and last occurrence of any target
- Works in monotonic boolean functions
- Find first element greater than X

---

8. Tips & Observations

- Always look for monotonicity to apply binary search.
- For non-increasing binary arrays:
  - First 0 â†' count of 1s
  - Last 1 â†' count of 1s
- Binary search boundary problems always need careful condition handling.

Q140: Floor in a Sorted Array

---

1. Problem Understanding

- You are given a sorted array without duplicates and a value x.
- You must find the floor of x:
  - Floor of x = largest element â‰¤ x.
- Return its index (0-based).

- If it doesnâ€™t exist â†’ return -1.

---

2. Constraints

- 1 â‰¤ N â‰¤ 1e5
- Array sorted strictly increasing
- 0 â‰¤ x â‰¤ arr[N-1]

---

3. Edge Cases

- x < smallest element â†’ return -1
- x > largest element â†’ return index of last element
- Exact match exists â†’ return index
- Single-element array
- Missing element but floor exists (example: x=5, array=[1,2,8â€¦])

---

4. Examples

```
Example 1

Input

7 0
1 2 8 10 11 12 19


Output

-1

Example 2

Input

7 5
1 2 8 10 11 12 19


Output

1

Example 3

Input

7 10
1 2 8 10 11 12 19
```

```
Output

3
```

---

5. Approaches

Approach 1: Linear Scan (Brute Force)

Idea:

- Scan from left until elements exceed x.

Steps:

- Loop through array
- Track last index with arr[i] <= x
- Return index

Java Code:

```java
public int floorIndex(int[] arr, int x) {
    int ans = -1;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] <= x) ans = i;
        else break;
    }
    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- Since array is sorted, once you exceed x, you wonâ€™t find a valid floor later.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Might scan whole array.
- ðŸ'¾ Space Complexity
  - O(1)

Approach 2: Binary Search (Optimal)

Idea:

- Use binary search to find last element â‰¤ x.

Steps:

- Initialize low=0, high=n-1, ans=-1
- While low â‰¤ high:
    - Compute mid
    - If arr[mid] â‰¤ x:
        - Record ans = mid
        - Go right â†’ low = mid + 1
    - Else (arr[mid] > x):
        - Go left â†’ high = mid - 1
- Return ans

Java Code:

```
public int floorIndex(int[] arr, int x) {
    int low = 0, high = arr.length - 1;
    int ans = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] <= x) {
            ans = mid;
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return ans;
}
```

ðŸ’ Intuition Behind the Approach:

- This is a boundary search:
- We want the rightmost value that is still â‰¤ x.
- Binary search lets us jump over large sections to find this efficiently.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
    - O(log n)
    - Binary search halves the space each step.
- ðŸ’¾ Space Complexity
- O(1)
- Constant space.

---

6. Justification / Proof of Optimality

- Binary search is ideal because the array is sorted and we need to find a boundary position.
- This ensures the best efficiency (O(log n)).

---

7. Variants / Follow-Ups

- Find ceil of x (smallest â‰¥ x)
- Find first element greater than x
- Lower bound / upper bound problems
- Find last occurrence of a number

---

8. Tips & Observations

- Floor â†' go right when arr[mid] <= x
- Ceil â†' go left when arr[mid] >= x
- Boundary problems always use modified binary search
- Always store ans before moving

Q141: Rotated Sorted Array Search

---

1. Problem Understanding

- You are given an array that was originally sorted in non-decreasing order but then rotated at some pivot. Example:
  - Original: 0 1 2 4 5 6 7
  - Rotated : 4 5 6 7 0 1 2
- Your task:
- Find the index of target B in this rotated sorted array.
- If not present â†' return -1.
- No duplicates exist.
- Goal: O(log n) â†' must use binary search.

---

2. Constraints

- 1 â‰¤ N â‰¤ 1e6
- 1 â‰¤ A[i] â‰¤ 1e9
- Array was sorted then rotated
- All elements are unique
- Must achieve O(log n)

---

3. Edge Cases

- Target is at pivot

- Array is not rotated at all (normal sorted)
- Target smaller than all elements
- Target larger than all elements
- Single-element array
- Pivot at index 0 or last index
- Target at boundaries

---

4. Examples

```
Example 1

Input

8
4 5 6 7 0 1 2 3
4


Output

0

Example 2

Input

4
5 17 100 3
6


Output

-1
```

---

5. Approaches

Approach 1: Linear Scan (Brute Force)

Idea:

- Loop from 0 to N-1
- If element equals target â†' return index
- Else return -1

Approach 2: Find Pivot then Binary Search Both Halves (Better)

Idea:

- In a rotated sorted array, the smallest element is the pivot.
- Example:
    - 4 5 6 7 0 1 2 â†’ pivot is at index 4.
- Steps:
    - Find pivot using binary search
    - Decide whether target is in left side or right side
    - Apply binary search normally in that half

Steps:

- Binary search to find smallest element index (pivot).
- If target lies in interval [arr[pivot], arr[n-1]] â†’ search in right half.
- Else search in left half.

Java Code:

```
public int search(int[] arr, int target) {
    int n = arr.length;

    // Step 1: find pivot (index of smallest element)
    int low = 0, high = n - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] > arr[high]) low = mid + 1;
        else high = mid;
    }
    int pivot = low;

    // Step 2: decide which half to search
    if (target >= arr[pivot] && target <= arr[n - 1]) {
        return binarySearch(arr, pivot, n - 1, target);
    } else {
        return binarySearch(arr, 0, pivot - 1, target);
    }
}

private int binarySearch(int[] arr, int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

ðŸ’ Intuition Behind the Approach:

- A rotated sorted array is two sorted halves.
- Find pivot â†’ pick correct half â†’ normal binary search.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - Pivot search: O(log n)
  - Binary search in correct half: O(log n)
  - Total: O(log n)
- ðŸ'¾ Space Complexity
  - O(1)

Approach 3: Single-Pass Binary Search (Optimal)

Idea:

- We modify binary search logic itself.
- At any mid, one side is always sorted:
- Either:
  - Left half is sorted
  - OR right half is sorted
- We check which half is sorted and decide where to move.

Steps:

- Standard binary search loop
- If arr[mid] == target â†' return mid
- Check which half is sorted
  - If left half sorted (arr[low] â‰¤ arr[mid]):
    - Check if target lies in left sorted range
    - â†' move right or left accordingly
  - Else right half sorted:
    - Check if target lies in right sorted range
    - â†' move accordingly

Java Code:

```java
public int search(int[] arr, int target) {
    int low = 0, high = arr.length - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) return mid;

        // Left half sorted
        if (arr[low] <= arr[mid]) {
            if (target >= arr[low] && target < arr[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
        // Right half sorted
        else {
            if (target > arr[mid] && target <= arr[high])
                low = mid + 1;
            else
                high = mid - 1;
        }
    }

    return -1;
}
```

ðŸ’ Intuition Behind the Approach:

- Every mid splits the array into two halves, and one half is guaranteed to be sorted.
- By checking:
  - whether target lies inside that sorted half
  - or in the unsorted half
- we discard half of the array each time â†’ classic binary search.
- This avoids finding pivot separately.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(log n)
  - We halve the search space every iteration.
- ðŸ’¾ Space Complexity
  - O(1)

---

6. Justification / Proof of Optimality

- Binary search works perfectly here because even after rotation, one part of the array remains sorted at every division.
- This property ensures we can always make the correct decision about which half to search.

---

7. Variants / Follow-Ups

- Search element in rotated array with duplicates
- Search min element in rotated array
- Search max element in rotated array
- Count rotations
- Find pivot index
- Find peak element

---

8. Tips & Observations

- At every mid, one half is sorted â†’ the key insight
- If arr[low] <= arr[mid] â†’ left sorted

- Else â†' right sorted
- Use ranges carefully when comparing target
- Avoid infinite loops by updating low and high properly

Q142: Peak Index in a Mountain Array

1. Problem Understanding

- You are given a mountain array, meaning:
- It strictly increases up to a peak
- Then strictly decreases
- Example:
  - 0 2 5 7 6 3 1 * â†' * peak
- You must find the index of the peak element.
- Mountain array guarantees:
- Exactly one peak
- Peak is not at index 0 or index n-1
- Must solve in O(log n)

---

2. Constraints

- 3 â‰¤ n â‰¤ 1e5
- Values: 0 â‰¤ arr[i] â‰¤ 1e6
- Array is guaranteed to be a perfect mountain
- Must solve in O(log n)

---

3. Edge Cases

- (not many because mountain is guaranteed)
  - Peak at index 1 (smallest valid peak)
  - Peak at index n-2 (largest valid peak)
  - Large values
  - Minimum length (3 elements)

---

4. Examples

```
Example 1

Input

3
0 1 0
```

```
Output

1

Example 2

Input

4
0 2 1 0


Output

1

Example 3

Input

4
0 10 5 2


Output

1
```

---

5. Approaches

Approach 1: Linear Scan (Brute Force)

Idea:

- Scan until the numbers stop increasing.
- The first time you see a drop, the previous index is peak.

Steps:

- Loop from i = 1 to n-1
- If arr[i] < arr[i-1]
  - â†' return i-1
- Because guaranteed mountain â†' no other checks needed.

Java Code:

```
public int peakIndex(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < arr[i - 1]) return i - 1;
    }
    return -1;
}
```

ðŸ' Intuition Behind the Approach:

- Peak is exactly where sequence changes from increasing â†' decreasing.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(n)
- ðŸ'¾ Space Complexity
    - O(1)

Approach 2: Binary Search (Optimal)

Idea:

- Binary search for the point where slope changes:
    - If arr[mid] < arr[mid + 1] â†' we are on the increasing slope
    - â†' peak is on the right
    - â†' move low = mid + 1
    - If arr[mid] > arr[mid + 1] â†' we are on the decreasing slope
    - â†' mid could be the peak
    - â†' move high = mid
- Eventually low == high â†' peak index.

Steps:

- Initialize low = 0, high = n-1
- While low < high:
    - Compute mid
    - If increasing part â†' move right
    - If decreasing part â†' move left
- Return low (same as high)

Java Code:

```java
public int peakIndex(int[] arr) {
    int low = 0, high = arr.length - 1;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] < arr[mid + 1]) {
            // Increasing part: peak is on the right
            low = mid + 1;
        } else {
            // Decreasing part: peak is mid or left side
            high = mid;
        }
    }

    return low; // or high, both same
}
```

ðŸ' Intuition Behind the Approach:

- The mountain array has a monotonic behavior:
    - Increasing â†' Peak â†' Decreasing
- At any mid:
    - If arr[mid] < arr[mid+1], slope is increasing â†' move right
    - If arr[mid] > arr[mid+1], slope is decreasing â†' peak is left or mid
- We narrow down the peak using binary search.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(log n)
    - We cut the search space in half each step.
- ðŸ'¾ Space Complexity
    - O(1)

---

6. Justification / Proof of Optimality

- Binary search works because the array is monotonic on each side of the peak:
    - left half strictly increasing
    - right half strictly decreasing
- This allows you to determine the direction of the peak at every step, ensuring O(log n) time.

---

7. Variants / Follow-Ups

- Find peak in mountain array (LeetCode 852)
- Find peak element in unsorted array (LeetCode 162)
- Find max in bitonic array
- Find first element greater than previous
- Find turning point in unimodal array

---

8. Tips & Observations

- Use mid < mid+1 to detect slope
- Always check arr[mid] < arr[mid+1] FIRST
- low will always converge to the peak
- Never check neighbors outside bounds because mountain is guaranteed

---