Step 4: Solution

**4. Binary Search**

Solved

Easy

Topics

Companies

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** nums = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**Constraints:**

- $1 <= nums.length <= 10^4$
- $-10^4 < nums[i], target < 10^4$
- All the integers in nums are **unique**.
- nums is sorted in ascending order.

**Problem Summary**

You are given:

- A **sorted array** nums (ascending order)
- An integer target

Your task:

- Return the **index of target** if it exists
- Otherwise, return **-1**
- **Time complexity must be O(log n)**

**Key Idea (Binary Search)**

Because the array is **sorted**, we can:

- Repeatedly **divide the search space into half**
- Compare target with the **middle element**

**Binary Search Logic (Step-by-Step)**

1. Set two pointers:
   - l = 0 (start)
   - r = n - 1 (end)
2. While l <= r:
   - Find middle index:
   - mid = l + (r - l) / 2
   - If nums[mid] == target → return mid
   - If nums[mid] < target → search right half (l = mid + 1)
   - Else → search left half (r = mid - 1)

3. If loop ends → target not found → return -1

```
class Solution {

public int search(int[] nums, int target) {

int l = 0, r = nums.length - 1;

while (l <= r) {

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

if (nums[mid] == target) {

return mid;

} else if (nums[mid] < target) {

l = mid + 1;

} else {

r = mid - 1;

}

}

return -1;

}

}
```

**Example Dry Run**

**Example 1**

**nums = [-1,0,3,5,9,12], target = 9**

l=0, r=5 → mid=2 → nums[2]=3 < 9 → l=3

l=3, r=5 → mid=4 → nums[4]=9 == target

Output = 4

**Example 2**

**nums = [-1,0,3,5,9,12], target = 2**

Target not found
Output = -1

**Complexity Analysis**

- **Time Complexity:** O(log n)
- **Space Complexity:** O(1)

**2529. Maximum Count of Positive Integer and Negative Integer**

Solved

Easy

Topics

Companies

Hint

Given an array nums sorted in **non-decreasing** order, return *the maximum between the number of positive integers and the number of negative integers.*

- In other words, if the number of positive integers in nums is pos and the number of negative integers is neg, then return the maximum of pos and neg.

**Note** that 0 is neither positive nor negative.

**Example 1:**

**Input:** nums = [-2,-1,-1,1,2,3]

**Output:** 3

**Explanation:** There are 3 positive integers and 3 negative integers. The maximum count among them is 3.

**Example 2:**

**Input:** nums = [-3,-2,-1,0,0,1,2]

**Output:** 3

**Explanation:** There are 2 positive integers and 3 negative integers. The maximum count among them is 3.

**Example 3:**

**Input:** nums = [5,20,66,1314]

**Output:** 4

**Explanation:** There are 4 positive integers and 0 negative integers. The maximum count among them is 4.

**Constraints:**

- 1 <= nums.length <= 2000
- -2000 <= nums[i] <= 2000
- nums is sorted in a **non-decreasing order**.

```
class Solution {

public int maximumCount(int[] nums) {

int pos = 0, neg = 0;

for (int x : nums) {

if (x > 0) pos++;

else if (x < 0) neg++;

}

return Math.max(pos, neg);

}

}
```

**Complexity**

- **Time:** O(n)
- **Space:** O(1)

**4.4 Search Insert Position**

Solved

Easy

Topics

Companies

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [1,3,5,6], target = 5

**Output:** 2

**Example 2:**

**Input:** nums = [1,3,5,6], target = 2

**Output:** 1

**Example 3:**

**Input:** nums = [1,3,5,6], target = 7

**Output:** 4

**Constraints:**

- 1 <= nums.length <= $10^4$
- $-10^4$ <= nums[i] <= $10^4$
- nums contains **distinct** values sorted in **ascending** order.
- $-10^4$ <= target <= $10^4$

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int lo = 0;
        int hi = nums.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (nums[mid] == target) {
                return mid;  // Target found
            } else if (nums[mid] < target) {
                lo = mid + 1;
            } else {
                hi = mid - 1;
            }
        }
        return lo;  // Insert position if not found
    }
}
```

#### 4.4.1 278. First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

**Example 1:**

**Input:** n = 5, bad = 4

**Output:** 4

**Explanation:**

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

**Example 2:**

**Input:** n = 1, bad = 1

**Output:** 1

**Constraints:**

- $1 <= bad <= n <= 2^{31} - 1$

### 🔍 Problem Summary

You are given versions 1 to n.

- There exists a **first bad version**
- Every version **after it is also bad**
- You can only check a version using:
- isBadVersion(version)
- Your goal: **find the first bad version**
- Minimize the number of API calls

### 💡 Key Insight

This is a **classic Binary Search** problem.

The versions look like:

Good Good Good Bad Bad Bad

Once isBadVersion(x) becomes true, it stays true.

👉 This monotonic behavior allows **binary search**.

### 🧠 Binary Search Logic (Step-by-Step)

1. Set search range:
2. l = 1, r = n
3. While l < r:
    - Find middle:
    - mid = l + (r - l) / 2
    - If mid is **bad**:
        - The first bad version is **at mid or before**
        - Move left:
        - r = mid
    - Else (mid is good):
        - First bad version must be **after mid**
        - Move right:
        - l = mid + 1
4. When loop ends:
    - l == r → this is the **first bad version**

### ☑ Java Code (Optimal – O(log n))

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int l = 1, r = n;

while (l < r) {

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

if (isBadVersion(mid)) {

```
r = mid; // first bad is at mid or before

} else {

l = mid + 1; // first bad is after mid

}

}

return l;

}

}
```

**Example Dry Run**

**Example 1**

n = 5, bad = 4

l=1, r=5 → mid=3 → isBadVersion(3)=false → l=4

l=4, r=5 → mid=4 → isBadVersion(4)=true → r=4

Loop ends → l = 4 ☑

**Example 2**

n = 1, bad = 1

l=1, r=1 → return 1

**Complexity Analysis**

- **Time Complexity:** O(log n) ☑
- **Space Complexity:** O(1)
- **API Calls:** Minimal (logarithmic)

**4.4.2. Minimum Operations to Exceed Threshold Value I**

You are given a **0-indexed** integer array nums, and an integer k.

In one operation, you can remove one occurrence of the smallest element of nums.

Return *the **minimum** number of operations needed so that all elements of the array are greater than or equal to* k.

**Example 1:**

**Input:** nums = [2,11,10,1,3], k = 10

**Output:** 3

**Explanation:** After one operation, nums becomes equal to [2, 11, 10, 3].

After two operations, nums becomes equal to [11, 10, 3].

After three operations, nums becomes equal to [11, 10].

At this stage, all the elements of nums are greater than or equal to 10 so we can stop.

It can be shown that 3 is the minimum number of operations needed so that all elements of the array are greater than or equal to 10.

**Example 2:**

**Input:** nums = [1,1,2,4,9], k = 1

**Output:** 0

**Explanation:** All elements of the array are greater than or equal to 1 so we do not need to apply any operations on nums.

**Example 3:**

**Input:** nums = [1,1,2,4,9], k = 9

**Output:** 4

**Explanation:** only a single element of nums is greater than or equal to 9 so we need to apply the operations 4 times on nums.

**Constraints:**

- 1 <= nums.length <= 50
- 1 <= nums[i] <= $10^9$
- 1 <= k <= $10^9$
- The input is generated such that there is at least one index i such that nums[i] >= k.

```
class Solution {

public int minOperations(int[] nums, int k) {

int cnt = 0;

for (int x : nums) {

if (x < k) cnt++;

}

return cnt;

}

}
```

**Complexity Analysis**

- **Time:** O(n)
- **Space:** O(1)

2.3 🧠 **Problem Understanding**

- Koko has **n piles of bananas**
- piles[i] = number of bananas in the ith pile
- Guards will return in **h hours**
- Koko chooses an **eating speed = k bananas/hour**
- In **one hour**, she eats from **only one pile**
  - If pile has ≥ k bananas → eats k
  - If pile has < k bananas → eats all and stops for that hour
- She wants to **finish all bananas within h hours**
- Goal: **find the minimum possible integer k**

⏱️ **How to Calculate Time for a Given Speed k**

For one pile with x bananas:

hours = ceil(x / k)

In Java, we avoid floating point by using:

(x + k - 1) / k

Total hours = sum of hours for all piles

🔍 **Key Observation**

- Minimum possible speed = **1**
- Maximum possible speed = **max(piles)**
  (Eating faster than the largest pile gives no extra benefit)

👉 The answer lies between **1 and max(piles)**
👉 This range is **sorted**, so we apply **Binary Search on the answer**

🚀 **Binary Search Strategy**

1. Set:
   - low = 1
   - high = max(piles)
2. While low <= high:

- mid = possible eating speed
- Calculate total hours needed at speed mid

3. Decision:
   - If hours <= h → speed is valid → try **smaller speed**
   - Else → speed too slow → **increase speed**

**🧠 Simple Logic to Remember (Interview Trick)**

**Higher speed → fewer hours**
**Lower speed → more hours**

```
class Solution {

public int minEatingSpeed(int[] piles, int h) {

int low = 1;

int high = 0;

// Find maximum pile

for (int p : piles) {

high = Math.max(high, p);

}

int answer = high;

while (low <= high) {

int mid = low + (high - low) / 2; // candidate speed

long totalHours = 0;

// Calculate hours needed at speed = mid

for (int p : piles) {

totalHours += (p + mid - 1) / mid; // ceil division

}

if (totalHours <= h) {

answer = mid; // valid speed

high = mid - 1; // try slower

} else {

low = mid + 1; // speed too slow

}

}

return answer;

}

}
```

**Complexity Analysis**

- **Time Complexity:**
  O(n log maxPile)
- **Space Complexity:**
  O(1)

**Example 1**

piles = [3, 6, 7, 11]

h = 8

**Step 1: Define Search Space**

- Minimum speed low = 1
- Maximum speed high = max(piles) = 11
- Answer will be between **1 and 11**

**Step 2: Start Binary Search**

◇ **Iteration 1**

low = 1, high = 11

mid = (1 + 11) / 2 = 6

Calculate hours needed if **k = 6**

| Pile | Bananas | Hours = ceil(pile / 6) |
|---|---|---|
| 3 | 3 | 1 |
| 6 | 6 | 1 |
| 7 | 7 | 2 |
| 11 | 11 | 2 |

Total hours = 1 + 1 + 2 + 2 = 6

✔ 6 <= 8 → Speed **6 is valid**
👉 Try slower speed

answer = 6

high = 5

◇ **Iteration 2**

low = 1, high = 5

mid = (1 + 5) / 2 = 3

Calculate hours if **k = 3**

| Pile | Bananas | Hours |
|---|---|---|
| 3 | 3 | 1 |
| 6 | 6 | 2 |
| 7 | 7 | 3 |
| 11 | 11 | 4 |

Total hours = 1 + 2 + 3 + 4 = 10

❌ 10 > 8 → Speed **too slow**
👉 Increase speed

low = 4

◇ **Iteration 3**

low = 4, high = 5

mid = (4 + 5) / 2 = 4

Calculate hours if **k = 4**

| Pile | Bananas | Hours |
|---|---|---|
| 3 | 3 | 1 |
| 6 | 6 | 2 |
| 7 | 7 | 2 |
| 11 | 11 | 3 |

Total hours = 1 + 2 + 2 + 3 = 8

✔ 8 <= 8 → Speed **4 is valid**
👉 Try slower

answer = 4

high = 3

**Step 3: Stop Condition**

low = 4, high = 3 → loop ends

☑ **Final Answer**

Minimum eating speed = 4

**[69. Sqrt(x)]**

This is **LeetCode 69 – Sqrt(x)**.
Best and expected approach is **Binary Search** ☑

🔑 **Key Idea (Why Binary Search?)**

- Square root is **monotonic**
  If $mid^2 \le x$, then all values < mid are also valid.
- We want the **largest integer k such that $k^2 \le x$**.
- Binary search gives **O(log x)** time → very efficient.

⚠ **Important Edge Cases**

- x = 0 → answer = 0
- x = 1 → answer = 1
- Avoid overflow → use long for mid * mid

🧠 **Binary Search Logic**

Search range:

low = 0

high = x

At every step:

1. mid = (low + high) / 2
2. If mid * mid == x → perfect square → return mid
3. If mid * mid < x → possible answer → move right
4. If mid * mid > x → move left

Final answer will be stored in ans.

☑ **Java Code (Recommended for Exams & Interviews)**

```
class Solution {

public int mySqrt(int x) {

if (x == 0 || x == 1)

return x;

int low = 0, high = x;

int ans = 0;

while (low <= high) {

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

long sq = (long) mid * mid; // avoid overflow

if (sq == x) {

return mid;

} else if (sq < x) {

ans = mid; // store possible answer

low = mid + 1;
```

```
} else {
high = mid - 1;
}
}
return ans;
}
}
```

**Dry Run (x = 8)**

low = 0, high = 8, ans = 0

mid = 4 → 4*4 = 16 > 8 → high = 3

mid = 1 → 1*1 = 1 < 8 → ans = 1, low = 2

mid = 2 → 2*2 = 4 < 8 → ans = 2, low = 3

mid = 3 → 3*3 = 9 > 8 → high = 2

loop ends → return ans = 2

✔ Output = **2**

**Time & Space Complexity**

- **Time:** O(log x)
- **Space:** O(1)

**Exam / Interview One-Line Mnemonic**

**"Find the largest number whose square is ≤ x using binary search."**

**Why long fixes it**

- long range = **±9 × $10^{18}$**
- Safely stores mid * mid even for large values

long sq = (long) mid * mid;

Now:

- No overflow
- Correct comparison
- Correct answer ☑

**Easy Exam Mnemonic**

**"mid² can overflow int, so use long."**

## 367. Valid Perfect Square

Given a positive integer num, return true *if* num *is a perfect square or* false *otherwise*.

A **perfect square** is an integer that is the square of an integer. In other words, it is the product of some integer with itself.

You must not use any built-in library function, such as sqrt.

**Example 1:**

**Input:** num = 16

**Output:** true

**Explanation:** We return true because 4 * 4 = 16 and 4 is an integer.

**Example 2:**

**Input:** num = 14

**Output:** false

**Explanation:** We return false because 3.742 * 3.742 = 14 and 3.742 is not an integer.

**Constraints:**

- $1 <= num <= 2^{31} - 1$

**Key Idea**

A number num is a **perfect square** if there exists an integer k such that:

k * k == num

Since squares grow **monotonically**, we can apply **binary search**.

**Binary Search Strategy**

- Search space: 1 → num
- For each mid:
  - if mid * mid == num → ☑ perfect square
  - if mid * mid < num → search right
  - if mid * mid > num → search left
- Use **long** to avoid overflow (same reason as in sqrt(x))

**☑ Java Code (Recommended)**

```
class Solution {

public boolean isPerfectSquare(int num) {

int low = 1, high = num;

while (low <= high) {

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

long sq = (long) mid * mid; // avoid overflow

if (sq == num) {

return true;

} else if (sq < num) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return false;

}

}
```

**Dry Run 1 (num = 16)**

low = 1, high = 16

mid = 8 → 8*8 = 64 > 16 → high = 7

mid = 4 → 4*4 = 16 == 16 → return true

✔ Output: **true**

**Dry Run 2 (num = 14)**

low = 1, high = 14

mid = 7 → 49 > 14 → high = 6

mid = 3 → 9 < 14 → low = 4

mid = 5 → 25 > 14 → high = 4

mid = 4 → 16 > 14 → high = 3

loop ends → return false

✔ Output: **false**

**Time & Space Complexity**

- **Time:** O(log num)
- **Space:** O(1)

**3.Sum of Square Numbers**

**Given a non-negative integer c, decide whether there're two integers a and b such that $a^2 + b^2 = c$.**

**Example 1:**

**Input: c = 5**

**Output: true**

**Explanation: 1 * 1 + 2 * 2 = 5**

**Example 2:**

**Input: c = 3**

**Output: false**

**Constraints:**

- **$0 <= c <= 2^{31} - 1$**

🔑 **Best Approach: Two Pointers (√c range)**

**Why this works**

- a² and b² are **non-negative**
- Maximum value of a or b can be √c
- Squares are **monotonic**, so we can move pointers smartly

🧠 **Algorithm (Two Pointer Logic)**

1. Set
2. left = 0
3. right = floor(sqrt(c))
4. While left <= right:
    - Compute sum = left² + right²
    - If sum == c → ☑ return true
    - If sum < c → increase left
    - If sum > c → decrease right
5. If loop ends → ❌ return false

⚠ Use **long** to avoid overflow.

☑ **Java Code (Interview Ready)**

class Solution {

public boolean judgeSquareSum(int c) {

long left = 0;

long right = (long) Math.sqrt(c);

```
while (left <= right) {

long sum = left * left + right * right;

if (sum == c) {

return true;

} else if (sum < c) {

left++;

} else {

right--;

}

}

return false;

}

}
```

**Dry Run 1 (c = 5)**

left = 0, right = 2

$0^2 + 2^2 = 4 < 5 \rightarrow$ left = 1

$1^2 + 2^2 = 5 == 5 \rightarrow$ return true

✔ Output: **true**

**Easy Exam Mnemonic**

**"Fix two pointers at 0 and √c, move based on sum of squares."**

**2778. Sum of Squares of Special Elements**

You are given a **1-indexed** integer array nums of length n.

An element nums[i] of nums is called **special** if i divides n, i.e. n % i == 0.

Return *the* ***sum of the squares*** *of all* ***special*** *elements of* nums.

**Example 1:**

**Input:** nums = [1,2,3,4]

**Output:** 21

**Explanation:** There are exactly 3 special elements in nums: nums[1] since 1 divides 4, nums[2] since 2 divides 4, and nums[4] since 4 divides 4.

Hence, the sum of the squares of all special elements of nums is nums[1] * nums[1] + nums[2] * nums[2] + nums[4] * nums[4] = 1 * 1 + 2 * 2 + 4 * 4 = 21.

**Problem Understanding**

- Array is **1-indexed** (important!)
- Length = n
- An index i is **special** if:
- n % i == 0
- For every such index, add:
- nums[i]²

**Core Logic**

1. Find n = nums.length
2. Loop i from 1 to n
3. If n % i == 0:
    - Take element at index i-1 (because Java is 0-indexed)
    - Add its square to answer

### ☑ Java Code (Clean & Exam-Ready)

```
class Solution {

public int sumOfSquares(int[] nums) {

int n = nums.length;

int sum = 0;

for (int i = 1; i <= n; i++) {

if (n % i == 0) {

sum += nums[i - 1] * nums[i - 1];

}

}

return sum;

}

}
```

### 🧨 Dry Run 1

**Input:** nums = [1,2,3,4]
n = 4

| i | n % i | special? | nums[i] | square |
|---|---|---|---|---|
| 1 | 0 | ☑ | 1 | 1 |
| 2 | 0 | ☑ | 2 | 4 |
| 3 | 1 | ❌ | – | – |
| 4 | 0 | ☑ | 4 | 16 |

**Sum = 1 + 4 + 16 = 21 ✔**

### 🧨 Dry Run 2

**Input:** nums = [2,7,1,19,18,3]
n = 6

Special indices: 1, 2, 3, 6

$2^2 + 7^2 + 1^2 + 3^2$

= 4 + 49 + 1 + 9

= 63

✔ Output = **63**

### ⏱ Complexity

- **Time:** O(n)
- **Space:** O(1)

### 📝 Easy Exam Mnemonic

**“If index divides n, square it and add.”**

**1588. Sum of All Odd Length Subarrays**

Given an array of positive integers arr, return *the sum of all possible* ***odd-length subarrays*** *of* arr.

A **subarray** is a contiguous subsequence of the array.

**Example 1:**

**Input:** arr = [1,4,2,5,3]

**Output:** 58

**Explanation:** The odd-length subarrays of arr and their sums are:

[1] = 1

[4] = 4

[2] = 2

[5] = 5

[3] = 3

[1,4,2] = 7

[4,2,5] = 11

[2,5,3] = 10

[1,4,2,5,3] = 15

If we add all these together we get 1 + 4 + 2 + 5 + 3 + 7 + 11 + 10 + 15 = 58

**Approach 1: Brute Force (Beginner Friendly)**

**Idea**

- Generate all subarrays
- Check if length is odd
- Add their sum

**Java Code**

```
class Solution {

public int sumOddLengthSubarrays(int[] arr) {

int n = arr.length;

int ans = 0;

for (int i = 0; i < n; i++) {

int sum = 0;

for (int j = i; j < n; j++) {

sum += arr[j];

int len = j - i + 1;

if (len % 2 == 1) {

ans += sum;

}

}

}

return ans;

}

}
```

**Complexity**

- **Time:** $O(n^2)$
- **Space:** O(1)

Works because $n \leq 100$, but **not optimal**.