→ revision needed
→ too hard
→ easy

Prob – 1] Largest Element in an Array

✅ **Problem Statement:**

You are given an array of integers. You need to **find and return the largest (maximum) element** in the array.

🔢 **Example:**

Input:

arr = [5, 7, 2, 9, 1]

Output:

9

} unsorted ∴ O(N)
if sorted → O(1)
coz last index element would be answer

✅ **Approach: Linear Search (Traverse Entire Array)**

🔍 **Idea:**

- Initialize a variable max with the first element of the array.
- Traverse through the array.
- At each step, if the current element is greater than max, update max.
- After traversing the entire array, max will hold the largest value.

✅ **Code:**

```
int largest(vector<int> &arr) {
  int max = arr[0];

  for (int i = 0; i < arr.size(); i++) {
    if (max < arr[i]) {
      max = arr[i];
    }
  }
```

```
int largest(vector<int> arr){
    int large = INT_MIN/arr[0];
    for(int i=0; i<arr.size(); i++){
        if(arr[i] > largest)
            large = arr[i];
    }
    return largest;
}
```

→ do this if there can be single element in array

→ we can use built in function of sorting : [arr.begin(), arr.end()]
& then return last element

```
    return max;
}
```

🔁 Time and Space Complexity:

Type	Value
Time	O(N) — Linear traversal
Space	O(1) — Constant space

🔁 Dry Run Example:

arr = [5, 7, 2, 9, 1]

max = 5

i = 0: arr[i]=5 → max=5

i = 1: arr[i]=7 → 7 > 5 → max=7

i = 2: arr[i]=2 → 2 < 7 → max=7

i = 3: arr[i]=9 → 9 > 7 → max=9

i = 4: arr[i]=1 → 1 < 9 → max=9

Return 9 ✅

📌 Things to Remember:

- Always initialize max with the **first element**, not zero (if array can contain negative numbers).
- Can also use built-in functions like *max_element(arr.begin(), arr.end()) in C++ STL.

- Edge Case: If the array has only one element, return that element directly.

Avoid using variable names like max in competitive coding unless you're not using <algorithm> (since max() is a standard function).

Prob – 2] Second Largest Element in an Array without sorting

✅ **Problem Statement:**

Given an array of integers, find the **second largest element** in the array.

If there is no such element (i.e., all elements are the same or only one element exists), return -1.

🔢 **Example:**

Input:

arr = [4, 1, 6, 7, 7]

Output:

6

✅ **Approach: Two Pass Linear Traversal**

🔍 **Idea:**

1. **First Pass:** Find the largest element.
2. **Second Pass:** Find the largest element that is **not equal to the largest** (i.e., second largest).

✅ **Code:**

```
int getSecondLargest(vector<int> &arr) {
  int n = arr.size();
  if (n == 0 || n == 1) {
    return -1;
  }
```

```
    int largest = INT_MIN;
    int slargest = -1;

    // First pass: find the largest element
    for (int i = 0; i < n; i++) {
        largest = max(largest, arr[i]);
    }

    // Second pass: find the second largest element
    for (int i = 0; i < n; i++) {
        if (arr[i] != largest && arr[i] > slargest) {
            slargest = arr[i];
        }
    }

    return slargest;
}
```

Time and Space Complexity:

Type	Value
Time	O(N) — Two linear passes
Space	O(1) — Constant space

Dry Run Example:

Input:

```cpp
arr = [10, 5, 10, 8]
```

First Pass:

- `largest = 10`

Second Pass:

- `arr[0] = 10` → skip
- `arr[1] = 5` → `slargest = 5`
- `arr[2] = 10` → skip
- `arr[3] = 8` → `8 > 5` → `slargest = 8`

✅ Final Output: `8`

📌 **Things to Remember:**

- Always handle edge cases:
- Empty array or single-element array → return -1.
- All elements same → second largest doesn't exist.
- Don't initialize slargest with INT_MIN if you plan to return -1 for not found.
- **Alternative approach:** You can find second largest in **one pass** by keeping track of both largest and second_largest in a single loop.
- Avoid printing inside utility functions (cout << "-1" is not good practice), instead return values.

✅ **Optimized Approach: Single-Pass Traversal**

🔍 **Idea:**

Traverse the array **once**, keeping track of both:

- largest: The largest element found so far.

- second_largest: The second largest, updated when a new largest is found or a value lies between the current largest and second largest.

✅ **Optimized Code (One Pass):**

```
int getSecondLargest(vector<int> &arr) {
    int n = arr.size();
    if (n < 2) return -1;  // Less than two elements

    int largest = INT_MIN;
    int slargest = -1;

    for (int i = 0; i < n; i++) {
        if (arr[i] > largest) {
            slargest = largest;
            largest = arr[i];
        } else if (arr[i] != largest && arr[i] > slargest) {
            slargest = arr[i];
        }
    }

    return slargest;
}
```

old value of largest added

🔁 **Dry Run:**

Input:

arr = [12, 35, 1, 10, 34, 1]

Iteration steps:

- i = 0: largest = 12, slargest = -1

- i = 1: largest = 35, slargest = 12
- i = 2: skip
- i = 3: 10 > 12? No, 10 > slargest(12)? No → skip
- i = 4: 34 > slargest(12) → slargest = 34
- i = 5: skip

✓ Final answer: 34

🧠 **Key Points to Remember:**

- **Faster** than the two-pass solution (still O(n), but fewer iterations).
- Be careful with the else if — ensures slargest isn't equal to largest.
- Use INT_MIN only when all array elements are guaranteed to be ≥ it, else initialize slargest = -1 for validity.

🧮 **Complexity:**

Type Value Time O(N) — Single pass Space O(1) — Constant space

Prob – 3] Check if the array is sorted

🧩 **Problem Statement:**

Check if Array is Sorted and Rotated

You're given an array nums. Determine whether it is possible to obtain a **non-decreasing sorted array** by **rotating** the original array **some number of times (including 0)**.

✅ **Code:**

```
bool check(vector<int>& nums) {
  int count = 0;
  int n = nums.size();

  for (int i = 0; i < n; i++) {
    if (nums[i] > nums[(i + 1) % n]) {
```

```
            count++;
        }
    }

    return count <= 1;
}
```

🧠 **Approach:**

- An array that is **sorted and rotated** will have **at most one place** where the order breaks (i.e., nums[i] > nums[i+1]).
- Since rotation makes the last element come before the first, we check the condition **in a circular manner** using (i + 1) % n.
- Count the number of such **"drops"**.
- If there's **0 or 1** such drop, the array is sorted and rotated.
- If **more than 1**, the array is not valid.

🧪 **Dry Run Example:**

Example 1:

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

- Drops:
- 5 > 1 → count = 1
- ✅ count = 1, so return true

Example 2:

nums = [2, 1, 3, 4]

- Drops:
- 2 > 1 → count = 1
- 1 < 3 → ok
- 3 < 4 → ok

- 4 < 2 (last to first) → not valid → count = 2 ❌
- Return false

🧮 **Time & Space Complexity:**

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

📝 **Points to Remember:**

- ✅ Use % n to handle circular comparison.
- 🌀 Array is considered sorted and rotated if there's **only one or zero** places where nums[i] > nums[i+1].
- 🚫 Do **not** sort or rotate manually—just count the **disorder points**.
- This works even when the rotation count is 0 (i.e., already sorted array).

Prob – 4] Remove duplicates from Sorted array

🧩 **Problem Statement:**

Remove Duplicates from Sorted Array

Given a **sorted array nums**, remove the duplicates in-place such that each element appears only once and return the new length.

Do not allocate extra space; modify the input array in-place with O(1) extra memory.

✅ **Code:**

```
int removeDuplicates(vector<int>& nums) {
  if (nums.empty()) return 0;  // Edge case: empty array

  int j = 0;  // Pointer for unique elements
  for (int i = 1; i < nums.size(); i++) {
    if (nums[i] != nums[j]) {
```

```
            j++;
            nums[j] = nums[i];  // Overwrite duplicate
        }
    }

    return j + 1;  // Length of the array with unique elements
}
```

🧠 **Approach (Two Pointer Technique):**

- Since the array is **sorted**, all duplicates are **adjacent**.
- Use two pointers:
- i: iterates through all elements.
- j: tracks the position of the last unique element.
- If nums[i] != nums[j], we've found a new unique value → increment j and write nums[i] to nums[j].

🧪 **Dry Run Example:**

Input:

`nums = [1, 1, 2, 2, 3]`

Process:

- i = 1: nums[1] == nums[0] → skip
- i = 2: nums[2] ≠ nums[0] → j = 1, nums[1] = 2
- i = 3: nums[3] == nums[1] → skip
- i = 4: nums[4] ≠ nums[1] → j = 2, nums[2] = 3

Modified nums: `[1, 2, 3, _, _]`

Return: `3` (length of unique array)

🧮 **Time & Space Complexity:**

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

📝 **Points to Remember:**

- ✅ Array must be **sorted** for this method to work correctly.
- 🚫 Do **not** use extra space.
- nums will be **modified in-place** up to the new length.
- You can use the returned length to iterate only through unique elements.

Prob – 5] Left Rotate an array by one place

🧩 **Problem Statement:**

Rotate Array by k Positions

Given an integer array nums, rotate the array to the right by k steps.

→ I know this question but I have different recursive approach need to learn this

✅ **Approach 1: Using Extra Space**

```
void rotate(vector<int>& nums, int k) {
    int n = nums.size();
    k = k % n;
    vector<int> rotated(n);

    for (int i = 0; i < n; i++) {
        rotated[(i + k) % n] = nums[i];
    }

    for (int i = 0; i < n; i++) {
        nums[i] = rotated[i];
    }
}
```

✔ **Dry Run:**

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

k = 2

Output: [4,5,1,2,3]

```
void rotate(arr, k){
 int n = arr.size();
   k = (k % n);
 vector<int> rotated(n);

for (int i → n) {
   rotated[(i+k) % n] = arr[i]
}

for (int i → n)
    nums[i] = rotated[i]
```

---

my recursive approach

```
void rotates( vector &arr)
    int temp = arr[arr.size()-1]

 for (int i → n-1) {
      arr[i+1] = arr[i]
   }
     arr[n-1] = temp;
}
```

```
void rotate (vector → arr)
 k = k % n;
 while(k) {
   rotate(arr);
    k--;
  }
}
```

```
nums = [1, 2, 3, 4, 5]
k = 2 → k % 5 = 2

Step 1:
rotated[2] = 1
rotated[3] = 2
rotated[4] = 3
rotated[0] = 4
rotated[1] = 5

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

Step 2:
nums = [4, 5, 1, 2, 3]
```

⏱️ **Time Complexity: O(n)**

🧠 **Space Complexity: O(n) (extra space)**

🔄 Approach 2: In-Place Rotation (Optimized)

This method reverses parts of the array to rotate without using extra space.

```
void reverse(vector<int>& nums, int start, int end) {
  while (start < end) {
    swap(nums[start], nums[end]);
    start++;
    end--;
  }
}
```

how to reverse logic

→ reversal algorithm

```
void rotate(vector<int>& nums, int k) {
    int n = nums.size();
    k = k % n;

    // Step 1: Reverse whole array
    reverse(nums, 0, n - 1);

    // Step 2: Reverse first k elements
    reverse(nums, 0, k - 1);

    // Step 3: Reverse the remaining elements
    reverse(nums, k, n - 1);
}
```

easiest approach

✔ **Dry Run:**

Input: nums = [1, 2, 3, 4, 5], k = 2

Step 1: reverse(nums, 0, 4) → [5, 4, 3, 2, 1]

Step 2: reverse(nums, 0, 1) → [4, 5, 3, 2, 1]

Step 3: reverse(nums, 2, 4) → [4, 5, 1, 2, 3]

⏱ **Time Complexity: O(n)**

💾 **Space Complexity: O(1)** ✅ **Optimized!**

💡 **Things to Remember:**

- Use extra space approach if simplicity matters.
- Use reverse-based approach when **space optimization is required**.
- Always do k = k % n before rotating to handle large k.

- This is a common interview question and helps test your understanding of array manipulation and optimization.

code understood revision required

Prob – 6] Move Zeros to end

🧩 **Problem Statement:**

Move all zeroes to the end of the array while maintaining the relative order of non-zero elements.

You must do it **in-place** without making a copy of the array and with **minimum number of operations**.

✅ **Approach: Two-Pointer Swap Technique**

```
void moveZeroes(vector<int>& nums) {
  int n = nums.size();
  int j = 0; // Pointer for placing non-zero elements

  for (int i = 0; i < n; i++) {
    if (nums[i] != 0) {
      swap(nums[i], nums[j]); // Move non-zero element to the front
      j++;
    }
  }
}
```

✔️ **Dry Run Example**

Input:

`nums = [0, 1, 0, 3, 12]`

Execution Steps:

- i = 0: nums[0] == 0 → do nothing
- i = 1: nums[1] == 1 → swap(nums[1], nums[0]) → nums = [1, 0, 0, 3, 12], j = 1
- i = 2: nums[2] == 0 → do nothing
- i = 3: nums[3] == 3 → swap(nums[3], nums[1]) → nums = [1, 3, 0, 0, 12], j = 2
- i = 4: nums[4] == 12 → swap(nums[4], nums[2]) → nums = [1, 3, 12, 0, 0], j = 3

Final Output:

`[1, 3, 12, 0, 0]`

⏱️ **Time Complexity:**

- **O(n)** — One traversal of the array

💾 **Space Complexity:**

- **O(1)** — In-place algorithm

💡 **Key Points to Remember:**

- This is a **stable** rearrangement (non-zero elements maintain relative order).
- Efficient for large datasets since it only uses a **single traversal** and **no extra space**.
- Do **not** move zeros unnecessarily (important for reducing swaps).
- j tracks the **index where the next non-zero should go**.
- Use swap only when i != j to avoid unnecessary operations (optional optimization).

Prob – 7] Linear Search

🧩 **Problem Statement:**

Given a sorted array arr and an integer k, determine if k is present in the array.

Return 1 if found, otherwise return 0.

simple linear search ka code hai

✅ **Approach: Linear Search**

🔧 **Code:**

```
bool searchInSorted(vector<int>& arr, int k) {
   int n = arr.size();
   for (int i = 0; i < n; i++) {
      if (arr[i] == k)
         return 1;  // Found
   }
   return 0;  // Not found
}
```

🧠 **Dry Run Example**

Input:

arr = [1, 3, 5, 7, 9], k = 5

Execution:

- i = 0 → arr[0] = 1 ≠ 5
- i = 1 → arr[1] = 3 ≠ 5
- i = 2 → arr[2] = 5 == 5 → return 1

Output: 1

⏱️ **Time Complexity:**

- **O(n)** in the worst case — when the element is at the end or not present.

💾 **Space Complexity:**

- **O(1)** — constant space, no extra memory used.

⚡ Optimized Approach (Binary Search):

Since the array is sorted, you can **optimize** it using **binary search**:

```
bool searchInSorted(vector<int>& arr, int k) {
  int low = 0, high = arr.size() - 1;

  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid] == k) return 1;
    else if (arr[mid] < k) low = mid + 1;
    else high = mid - 1;
  }
  return 0;
}
```

→ agar sorted array ho toh yeh approach use kare

🔁 Time Complexity (Binary Search): O(log n)

- 💡 Use this optimized version **only if the array is sorted** (which it is in this problem).

📝 Points to Remember:

- Always check if the array is sorted to consider binary search.
- For small arrays, linear search might be as fast or even better due to lower overhead.
- Binary search reduces time drastically for large arrays.
- In interviews, **mention both** approaches and **justify your choice**.

Prob – 8] Find the Union

🧩 Problem Statement:

Given: Two arrays a[] and b[] (not necessarily sorted).

Task: Return the **union** of the two arrays in sorted order without duplicates.

✅ **Approach: Using set in C++**

💡 **Idea:**

A set in C++ stores **unique** elements in **sorted** order automatically. So, inserting all elements from both arrays into a set directly gives the union.

🔧 **Code:**

```
vector<int> findUnion(vector<int> &a, vector<int> &b) {
    set<int> s; //declare

    for (int i = 0; i < a.size(); i++) {
        s.insert(a[i]); //copy
    }

    for (int i = 0; i < b.size(); i++) {
        s.insert(b[i]); //copy
    }

    return vector<int>(s.begin(), s.end());
}
```

yeh bhi easy hai

① declare set

② copy elements from first vector

③ copy from next vector

④ convert set to vector

↳ converting to vector

🧠 **Dry Run Example**

Input:

a = {1, 2, 4, 5}, b = {2, 3, 5, 6}

Execution:

- set after inserting a: {1, 2, 4, 5}
- set after inserting b: {1, 2, 3, 4, 5, 6}

Output: vector = {1, 2, 3, 4, 5, 6}

⏱️ **Time Complexity:**

- Inserting each element into the set takes **O(log n)**.
- Let m = a.size() and n = b.size(), total complexity:
- **O((m + n) * log(m + n))**

💾 **Space Complexity:**

- **O(m + n)** to store the union in the set.

⚡ Alternative Approach (Two Pointer for Sorted Arrays):

If arrays a and b are sorted, a **two-pointer** technique can be used in **O(m + n)** time and **O(1)** extra space (excluding output):

```
vector<int> findUnion(vector<int> &a, vector<int> &b) {
  vector<int> result;
  int i = 0, j = 0;
  while(i < a.size() && j < b.size()) {
    if (a[i] < b[j]) result.push_back(a[i++]);
    else if (a[i] > b[j]) result.push_back(b[j++]);
    else {
      result.push_back(a[i]);
      i++; j++;
    }
  }
  while(i < a.size()) result.push_back(a[i++]);
  while(j < b.size()) result.push_back(b[j++]);

  // Remove duplicates
  result.erase(unique(result.begin(), result.end()), result.end());
  return result;
```

}

📝 **Points to Remember:**

- set automatically handles both uniqueness and sorting.
- Use unordered_set if sorting is not required — gives better time performance.
- Two-pointer is best for **already sorted arrays**.
- set approach is clean, short, and safe for any type of input.

Prob – 9] Find missing number in an array

🧩 **Problem Statement**

Given: An array nums containing n distinct numbers from the range 0 to n.

Task: Return the **one number that is missing** from the array.

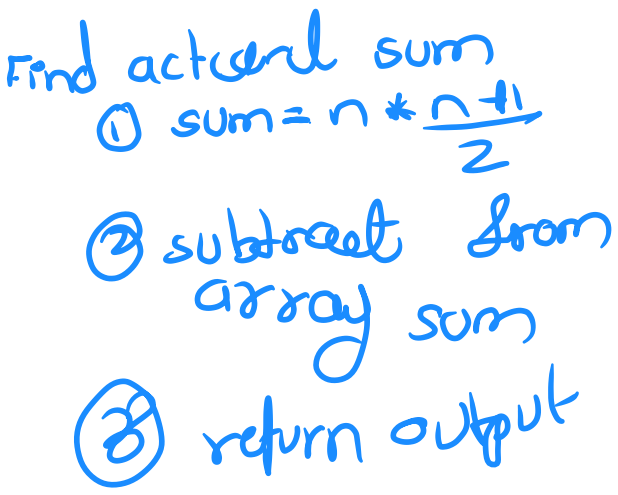
✅ **Approach: Mathematical Formula (Sum Method)**

💡 **Idea:**

- The sum of first n natural numbers is n * (n + 1) / 2.
- Subtract the **actual sum** of the array from the **expected total sum** to find the missing number.

🔧 **Code:**

```
int missingNumber(vector<int>& nums) {
    int n = nums.size();
    int total = n * (n + 1) / 2;
    int sum = 0;
    for (int i = 0; i < nums.size(); i++) {
        sum += nums[i];
    }
    return total - sum;
```



}

🔍 Dry Run Example

Input: nums = [3, 0, 1]

- n = 3
- Expected sum: 3 * (3 + 1) / 2 = 6
- Actual sum: 3 + 0 + 1 = 4
- Missing number = 6 - 4 = 2 ✅

⏱️ Time Complexity

- **O(n)** – Single pass to compute the sum.

💾 Space Complexity

- **O(1)** – No extra space used.

⚡ Alternative Approaches:

1. **XOR Approach** (Also O(n), O(1)): → imp approach

```
int missingNumber(vector<int>& nums) {
    int xor1 = 0, xor2 = 0;
    int n = nums.size();
    for (int i = 0; i < n; i++) {
        xor1 ^= nums[i];
        xor2 ^= i;
    }
    xor2 ^= n;
    return xor1 ^ xor2;
}
```

① xor 0 ^ arr[i]
② xor2 ^= ①
continously store xor of elements present in vector
↑ index
continously stores xor of 0→size-1
③ then xor2 with n
④ [xor1 ^ xor2] returns missing num

Uses XOR logic: A number XORed with itself is 0, so only the missing number survives.

Hashing/Boolean Array: O(n) time and O(n) space — not optimal when space is a concern.

📝 **Key Points to Remember:**

- This is an optimized solution using the formula for the sum of the first n natural numbers.
- Works only if numbers are distinct and in range [0, n].
- Be careful of **integer overflow** for very large n — consider using long long if needed.
- Alternative XOR method is also space-efficient and avoids potential overflows.

Prob – 10] Maximum Consecutive Ones

🧩 **Problem Statement**

Given: A binary array nums (consisting only of 0s and 1s).

Task: Return the **maximum number of consecutive 1s** in the array.

✅ **Approach: Linear Scan (One Pass)**

💡 **Idea:**

- Traverse the array and keep a counter for consecutive 1s.
- Whenever a 0 is encountered, update the maximum if needed and reset the counter.

🔧 **Code:**

```
int findMaxConsecutiveOnes(vector<int>& nums) {
  int count = 0;  // Tracks current streak of 1s
  int ans = 0;    // Stores max streak

  for(int i = 0; i < nums.size(); i++) {
    if(nums[i] == 1)
      count++;  // Increment count for a 1
```

```
    else {
      ans = max(ans, count); // Update max
      count = 0;          // Reset on 0
    }
  }

  return max(ans, count); // Final check after loop
}
```

🔍 Dry Run Example

Input: nums = [1, 1, 0, 1, 1, 1]

- i = 0 → 1 → count = 1
- i = 1 → 1 → count = 2
- i = 2 → 0 → ans = max(0, 2) = 2 → reset count
- i = 3 → 1 → count = 1
- i = 4 → 1 → count = 2
- i = 5 → 1 → count = 3
- → End of loop → Final max = max(2, 3) = **3**

Output: 3

⏱️ Time Complexity

- **O(n)** – One traversal of the array.

💾 Space Complexity

- **O(1)** – No extra space used.

📌 Points to Remember:

- Always update ans **after encountering a zero** to store max streak so far.
- Final return uses max(ans, count) to ensure **trailing ones are considered**.

- Works for arrays containing all 1s or all 0s.
- Initialize both count and ans correctly to avoid logic errors.

Prob – 11] Find the number that appears once, and other numbers twice.

✅ **Problem Statement:**

Given an array of integers where every element appears **twice** except for **one**, find that single element.

🔹 **Brute Force Approach**

💡 **Idea:**

Check frequency of every element using a nested loop. If frequency is 1, return it.

🔁 **Steps:**

1. Traverse each element using outer loop.
2. Count its frequency using an inner loop.
3. If frequency == 1, return the element.

🔢 **Code:**

```
int getSingleElement(vector<int> &arr) {
  int n = arr.size();
  for (int i = 0; i < n; i++) {
    int num = arr[i];
    int cnt = 0;
    for (int j = 0; j < n; j++) {
      if (arr[j] == num)
        cnt++;
    }
```

```
      if (cnt == 1) return num;
    }
    return -1;
}
```

🧮 **Dry Run:**

Input: [2, 3, 2, 4, 3]

- 2 → appears twice
- 3 → appears twice
- 4 → appears once ✅

⏱️ **Time Complexity: O(n^2)**

🧠 **Space Complexity: O(1)**

🔹 **Better Approach 1: Hashing Using Array**

💡 **Idea:**

Use a frequency array to count occurrences.

🔁 **Steps:**

1. Find the maximum element to size the hash array.
2. Declare vector<int> hash(max+1, 0);
3. Count each element's frequency using hash[arr[i]]++
4. Traverse array and return the element with count 1.

🔢 **Code:**

```
int getSingleElement(vector<int> &arr) {
    int n = arr.size();
```

```
    int maxi = arr[0];
    for (int i = 0; i < n; i++) {
        maxi = max(maxi, arr[i]);
    }
    vector<int> hash(maxi + 1, 0);
    for (int i = 0; i < n; i++) {
        hash[arr[i]]++;
    }
    for (int i = 0; i < n; i++) {
        if (hash[arr[i]] == 1)
            return arr[i];
    }
    return -1;
}
```

📌 **Note:**

- Only works if elements are non-negative and within reasonable size (to avoid memory issues).

✅ **Declaring hash array in C++:**

```
vector<int> hash(size, 0);  // Initializes all values to 0
```

🧾 **Dry Run:**

Input: [2, 3, 2, 4, 3]

hash[2] = 2, hash[3] = 2, hash[4] = 1 → return 4

⏱️ **Time Complexity: O(n + maxElement)**

🧠 **Space Complexity: O(maxElement)**

🔹 **Better Approach 2: Hashing Using Map**

💡 **Idea:**

Use a map to handle all integers (including negatives).

🔁 **Steps:**

1. Declare map<int, int> mpp;
2. Count frequencies with mpp[arr[i]]++
3. Traverse map and return key with value 1.

🔢 **Code:**

```
int getSingleElement(vector<int> &arr) {
   map<int, int> mpp;
   for (int i = 0; i < arr.size(); i++) {
      mpp[arr[i]]++;
   }
   for (auto it : mpp) {
      if (it.second == 1)
         return it.first;
   }
   return -1;
}
```

✅ **Declaring and using map in C++:**

```
map<int, int> mpp;
mpp[5]++; // Adds 5 with value 1 if not present, else increments value
```

🧮 **Dry Run:**

Input: [2, 3, 2, 4, 3]

mpp = {2:2, 3:2, 4:1} → return 4

⏱️ **Time Complexity: O(n log n) (due to map insertion)**

🗃️ **Space Complexity: O(n)**

🔹 **Optimal Approach: XOR Method**

💡 **Idea:**

- a ^ a = 0
- 0 ^ a = a

All pairs cancel out, and only the unique number remains.

🔢 **Code:**

```
int singleNumber(vector<int>& nums) {
   int xorr = 0;
   for(int i = 0 ; i < nums.size(); i++){
      xorr = xorr ^ nums[i];
   }
   return xorr;
}
```

🧮 **Dry Run:**

Input: [2, 3, 2, 4, 3]

2 ^ 3 ^ 2 ^ 4 ^ 3 = (2^2) ^ (3^3) ^ 4 = 0 ^ 0 ^ 4 = 4

⏱️ **Time Complexity: O(n)**

🗃️ **Space Complexity: O(1)** ✅ **Best**

Summary Table:

Approach	Time Complexity	Space Complexity	Works with Negatives?	Notes
Brute Force	$O(n^2)$	O(1)	✅ Yes	Slow for large input
Hashing (Array)	O(n + max)	O(max)	❌ No	Fast, but limited
Hashing (Map)	O(n log n)	O(n)	✅ Yes	Balanced approach
XOR	O(n)	O(1)	✅ Yes	✅ Best & most optimal

Prob – 12] Longest subarray with given sum K(positives)

🧩 Problem Statement

Given: A non-empty array a of integers (which may contain negative elements) and a target sum k.

Task: Find the length of the longest subarray whose sum is exactly equal to k.

Example:

- **Input:** a = [1, -2, 3, 4, 5], k = 9
- **Output:** 2 (The subarray [4, 5] has sum 9.)

✅ Approach:

1. Using 2 Pointers (Optimal Approach)

This approach works efficiently when the array contains **only non-negative** elements. When negative elements are present, this approach can fail because it assumes that moving the left pointer to the right can always reduce the sum.

💡 Idea:

- Use two pointers (left and right) to create a sliding window.
- Expand the window by moving right pointer, and adjust the window by moving the left pointer whenever the sum exceeds k.

💻 **Code:**

```
int getLongestSubarray(vector<int>& a, long long k) {
    int n = a.size();
    int left = 0, right = 0;
    long long sum = a[0];
    int maxLen = 0;

    while (right < n) {
        while (left <= right && sum > k) {
            sum -= a[left];
            left++;
        }

        if (sum == k) {
            maxLen = max(maxLen, right - left + 1);
        }

        right++;
        if (right < n) sum += a[right];
    }

    return maxLen;
}
```

🧩 **Dry Run Example:**

Input: a = [1, -2, 3, 4, 5], k = 9

- Initialize left = 0, right = 0, sum = 1, maxLen = 0
- **Iteration 1:** right = 0, sum = 1, no update as sum != k

- **Iteration 2:** right = 1, sum = -1, no update as sum != k
- **Iteration 3:** right = 2, sum = 2, no update as sum != k
- **Iteration 4:** right = 3, sum = 6, no update as sum != k
- **Iteration 5:** right = 4, sum = 11, adjust window by moving left
- **Final Result:** maxLen = 2 (Longest subarray is [4, 5])

⏱️ **Time Complexity:**

- **O(n)** – One pass through the array (right pointer only moves once per iteration).
- **Space Complexity:**
- **O(1)** – Only a few variables are used.

📌 **Points to Remember:**

- **Sliding window** can be used efficiently for non-negative elements.
- For negative elements, the sliding window needs adjustment to handle the effect of reducing the sum on shrinking the window.

💡 **Other Approaches to Solve the Problem:**

2. Using Hashing (Prefix Sum Approach)

When negative numbers are involved, the **Hashing (Prefix Sum)** approach becomes more suitable. By keeping track of the prefix sum of elements and using a hashmap, we can efficiently find the longest subarray that sums to k, even if there are negative numbers.

💡 **Idea:**

- Track the cumulative sum (prefix sum).
- Use a hashmap to store the earliest occurrence of a specific sum.
- If the difference between the current sum and k has been encountered before, we can calculate the length of the subarray between the two indices.

💻 **Code:**

```
int getLongestSubarray(vector<int>& a, long long k) {
  unordered_map<long long, int> prefixSumMap;
  long long sum = 0;
  int maxLen = 0;
```

```
    for (int i = 0; i < a.size(); i++) {
        sum += a[i];

        if (sum == k) {
            maxLen = i + 1; // Update maxLen
        } else if (prefixSumMap.find(sum - k) != prefixSumMap.end()) {
            maxLen = max(maxLen, i - prefixSumMap[sum - k]);
        }

        prefixSumMap[sum] = i;
    }

    return maxLen;
}
```

⏱️ **Time Complexity:**

- **O(n)** – One pass through the array. Hash map operations (insert and lookup) are O(1) on average.

💾 **Space Complexity:**

- **O(n)** – Space for storing the prefix sums in the hashmap.

3. Brute-force Approach (Using Two Loops)

This approach generates all possible subarrays using two nested loops and checks their sums. It's inefficient for large arrays but works for small arrays or when a quick, simple solution is needed.

💡 **Idea:**

- Generate all possible subarrays.
- Compute the sum of each subarray and check if it matches k.

💻 **Code:**

```
int getLongestSubarray(vector<int>& a, long long k) {
  int maxLen = 0;
  int n = a.size();

  for (int start = 0; start < n; start++) {
    long long sum = 0;
    for (int end = start; end < n; end++) {
      sum += a[end];
      if (sum == k) {
        maxLen = max(maxLen, end - start + 1);
      }
    }
  }

  return maxLen;
}
```

⏱️ **Time Complexity:**

- **O(n^2)** – Two loops to generate and sum subarrays.

💾 **Space Complexity:**

- **O(1)** – No extra space used.

4. Optimized Brute-force with Two Loops (Using Hash Map)

If we optimize the brute-force approach by using a hash map to store the sums, it would help in reducing the redundant checks, similar to the hashing approach above.

5. Using Dynamic Programming (Optimal for Some Cases)

Dynamic programming is another approach where we calculate the longest subarray for each index and store the results. However, it is generally less efficient than the prefix sum approach for this problem.

🔑 **Key Points to Remember:**

- **2 Pointers** approach is optimal for non-negative arrays.
- **Hashing (Prefix Sum)** is ideal for handling both negative and positive numbers, leveraging the cumulative sum.
- **Brute-force** is easy to implement but inefficient for larger arrays, with a time complexity of **O(n^2)**.

Prob – 13] Longest subarray with sum K (Positives + Negatives)

🧩 **Problem Statement**

Given: A non-empty array a of integers (which may contain negative elements) and a target sum k.

Task: Find the length of the longest subarray whose sum is exactly equal to k.

Example:

- **Input:** a = [1, -2, 3, 4, 5], k = 9
- **Output:** 2 (The subarray [4, 5] has sum 9.)

✅ **Approach:**

1. Using 2 Pointers (Optimal Approach)

This approach works efficiently when the array contains **only non-negative** elements. When negative elements are present, this approach can fail because it assumes that moving the left pointer to the right can always reduce the sum.

💡 **Idea:**

- Use two pointers (left and right) to create a sliding window.
- Expand the window by moving right pointer, and adjust the window by moving the left pointer whenever the sum exceeds k.

💻 **Code:**

```
int getLongestSubarray(vector<int>& a, long long k) {
```

```
    int n = a.size();
    int left = 0, right = 0;
    long long sum = a[0];
    int maxLen = 0;

    while (right < n) {
        while (left <= right && sum > k) {
            sum -= a[left];
            left++;
        }

        if (sum == k) {
            maxLen = max(maxLen, right - left + 1);
        }

        right++;
        if (right < n) sum += a[right];
    }

    return maxLen;
}
```

🧩 **Dry Run Example:**

Input: a = [1, -2, 3, 4, 5], k = 9

- Initialize left = 0, right = 0, sum = 1, maxLen = 0
- **Iteration 1:** right = 0, sum = 1, no update as sum != k
- **Iteration 2:** right = 1, sum = -1, no update as sum != k
- **Iteration 3:** right = 2, sum = 2, no update as sum != k

- **Iteration 4:** right = 3, sum = 6, no update as sum != k
- **Iteration 5:** right = 4, sum = 11, adjust window by moving left
- **Final Result:** maxLen = 2 (Longest subarray is [4, 5])

⏱️ **Time Complexity:**

- **O(n)** – One pass through the array (right pointer only moves once per iteration).
- **Space Complexity:**
- **O(1)** – Only a few variables are used.

📌 **Points to Remember:**

- **Sliding window** can be used efficiently for non-negative elements.
- For negative elements, the sliding window needs adjustment to handle the effect of reducing the sum on shrinking the window.

💡 **Other Approaches to Solve the Problem:**

2. Using Hashing (Prefix Sum Approach)

When negative numbers are involved, the **Hashing (Prefix Sum)** approach becomes more suitable. By keeping track of the prefix sum of elements and using a hashmap, we can efficiently find the longest subarray that sums to k, even if there are negative numbers.

💡 **Idea:**

- Track the cumulative sum (prefix sum).
- Use a hashmap to store the earliest occurrence of a specific sum.
- If the difference between the current sum and k has been encountered before, we can calculate the length of the subarray between the two indices.

💻 **Code:**

```
int getLongestSubarray(vector<int>& a, long long k) {
  unordered_map<long long, int> prefixSumMap;
  long long sum = 0;
  int maxLen = 0;

  for (int i = 0; i < a.size(); i++) {
```

```
    sum += a[i];

    if (sum == k) {
      maxLen = i + 1; // Update maxLen
    } else if (prefixSumMap.find(sum - k) != prefixSumMap.end()) {
      maxLen = max(maxLen, i - prefixSumMap[sum - k]);
    }

    prefixSumMap[sum] = i;
  }

  return maxLen;
}
```

⏱️ **Time Complexity:**

- **O(n)** – One pass through the array. Hash map operations (insert and lookup) are O(1) on average.

💾 **Space Complexity:**

- **O(n)** – Space for storing the prefix sums in the hashmap.

3. Brute-force Approach (Using Two Loops)

This approach generates all possible subarrays using two nested loops and checks their sums. It's inefficient for large arrays but works for small arrays or when a quick, simple solution is needed.

💡 **Idea:**

- Generate all possible subarrays.
- Compute the sum of each subarray and check if it matches k.

💻 **Code:**

```
int getLongestSubarray(vector<int>& a, long long k) {
  int maxLen = 0;
```

```
    int n = a.size();

    for (int start = 0; start < n; start++) {
        long long sum = 0;
        for (int end = start; end < n; end++) {
            sum += a[end];
            if (sum == k) {
                maxLen = max(maxLen, end - start + 1);
            }
        }
    }

    return maxLen;
}
```

⏱️ **Time Complexity:**

- **O(n^2)** – Two loops to generate and sum subarrays.

💾 **Space Complexity:**

- **O(1)** – No extra space used.

4. Optimized Brute-force with Two Loops (Using Hash Map)

If we optimize the brute-force approach by using a hash map to store the sums, it would help in reducing the redundant checks, similar to the hashing approach above.

5. Using Dynamic Programming (Optimal for Some Cases)

Dynamic programming is another approach where we calculate the longest subarray for each index and store the results. However, it is generally less efficient than the prefix sum approach for this problem.

🔑 **Key Points to Remember:**

- **2 Pointers** approach is optimal for non-negative arrays.
- **Hashing (Prefix Sum)** is ideal for handling both negative and positive numbers, leveraging the cumulative sum.
- **Brute-force** is easy to implement but inefficient for larger arrays, with a time complexity of **O(n^2)**.