# Delloite

## 1.Reverse a String

Given a string s, write a function to reverse it. The function should return a new string that is the reverse of s.

### Input

- A single string s with 1≤length of s≤10^5

### Output

- A single string representing the reversed version of s.

### Constraints

- The solution should be efficient in time complexity, ideally O(n), where n is the length of the string.

Sample Input:

```
Unset
Input: "abc"
```

Sample Output:

```
Unset
Output: "cba"
```

Solutions:

```
C/C++


string reverseString(const string& s) {
    string reversed = s;
    reverse(reversed.begin(), reversed.end());
    return reversed;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input: "hello"
Output: "olleh"
```

Test Case 2:

```
Unset
Input: "12345"
Output: "54321"
```

Test Case 3:

```
Unset
Input: "a"
Output: "a"
```

Test Case 4:

```
Unset
Input: "aabb"
Output: "bbaa"
```

Test Case 5:

```
Unset
Input: "xyyz"
Output: "zyyx"
```

## 2. Check for Palindrome String

Given a string `s`, determine if it is a palindrome. A palindrome is a string that reads the same backward as forward. The function should return `true` if `s` is a palindrome, and `false` otherwise.

### Input

- A single string `s` containing alphanumeric characters (1 ≤ length of `s` ≤ 10^5).

### Output

- `true` if `s` is a palindrome, `false` otherwise.

### Constraints

- Consider only lower case characters .
- The solution should be efficient in time complexity, ideally O(n), where n is the length of the string.
-

Sample Input:

```
Unset
Input: "abc"
```

Sample Output:

```
Unset
Output: false
```

Solutions:

```
C/C++


bool isPalindrome(const string& s) {
    int left = 0;
    int right = s.size() - 1;

    while (left < right) {
        if (s[left] != s[right]) return false;
        left++;
        right--;
    }
    return true;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input: "hello"
Output: false
```

Test Case 2:

```
Unset
Input: "racecar"
Output: true
```

Test Case 3:

```
Unset
Input: "a"
Output: true
```

Test Case 4:

```
Unset
Input: "aabb"
Output: false
```

Test Case 5:

```
Unset
Input: "xyyx"
Output: true
```

## 3.Find Maximum Element in an Array

Given an array `arr` of integers, write a function that returns the maximum element in the array.

### Input

- An integer array `arr` of size n (1 ≤ n≤ 10^5), where each element in `arr` is an integer.

### Output

- An integer representing the maximum element in the array.

## Constraints

- The array will contain at least one element.
- The solution should run in O(n) time complexity.

Sample Input:

```
Unset
Input: [3, 5, 9, 1, 7]
```

Sample Output:

```
Unset
Output: 9
```

Solutions:

```
C/C++


int findMaxElement(const vector<int>& arr) {
    int maxElement = INT_MIN;
    for (int num : arr) {
        if (num > maxElement) {
            maxElement = num;
        }
    }
    return maxElement;
}
```

Hidden Test Case

Test Case 1:

```
Unset
Input: [10, 20, 30, 40]
Output: 40
```

Test Case 2:

```Unset
Input: [-1, -5, -3, -4]
Output: -1
```

Test Case 3:

```Unset
Input: [100]
Output: 100
```

Test Case 4:

```Unset
Input: [50, 60, 10, 20, 60]
Output: 60
```

Test Case 5:

```Unset
Input: [50, 60, 10, 20, 60, 90, 90]
Output: 90
```

## 4.Fibonacci Sequence

Given an integer n, write a function that returns the first n numbers in the Fibonacci sequence.

The Fibonacci sequence is defined as:

- F(0)=0
- F(1)=1
- F(n)=F(n−1)+F(n−2) for n≥2

Input

- An integer n (1 ≤ n≤ 30), which represents the number of terms to generate in the Fibonacci sequence.

Output

- A list of integers representing the first n numbers in the Fibonacci sequence.

Constraints

- The solution should have a time complexity of O(n).

Sample Input:

```
Unset
Input: 2
```

Sample Output:

```
Unset
Output: [0, 1]
```

Solutions:

```
C/C++
vector<int> generateFibonacci(int n) {
    vector<int> fibonacci;
    if (n >= 0) fibonacci.push_back(0);
    if (n >= 1) fibonacci.push_back(1);

    for (int i = 2; i <= n; i++) {
        fibonacci.push_back(fibonacci[i - 1] + fibonacci[i - 2]);
    }

    return fibonacci;
}
```

Hidden Test Case :

Test Case 1:

```
Unset
Input: 5
Output: [0, 1, 1, 2, 3]
```

Test Case 2:

```
Unset
Input: 6
Output: [0, 1, 1, 2, 3, 5]
```

Test Case 3:

```
Unset
Input: 7
Output: [0, 1, 1, 2, 3, 5, 8]
```

Test Case 4:

```
Unset
Input: 3
Output: [0, 1, 1]
```

Test Case 5:

```
Unset
Input: 1
Output: [0]
```

## 5.Merge Two Sorted Arrays

### Problem Statement

Given two sorted arrays, `arr1` and `arr2`, write a function that merges them into a single sorted array. The resulting array should also be sorted in non-decreasing order.

### Input

- `arr1`: A sorted array of integers (0 ≤ length of arr1 ≤ 3* 10^4).
- `arr2`: A sorted array of integers (0 ≤ length of arr ≤ 3*10^4).

### Output

- A single sorted array that contains all elements from both `arr1` and `arr2`.

### Constraints

- Both `arr1` and `arr2` are already sorted in non-decreasing order.

Sample Input:

```
Unset
Input:
```

```
arr1 = [1, 3, 5, 7]
arr2 = [2, 4, 6, 8]
```

Sample Output:

```
Unset
Output:
[1, 2, 3, 4, 5, 6, 7, 8]
```

Solutions:

```
C/C++

vector<int> mergeSortedArrays(const vector<int>& arr1, const vector<int>& arr2)
{
    int i = 0, j = 0;
    vector<int> mergedArray;

    // Merge until one of the arrays is exhausted
    while (i < arr1.size() && j < arr2.size()) {
        if (arr1[i] < arr2[j]) {
            mergedArray.push_back(arr1[i]);
            i++;
        } else {
            mergedArray.push_back(arr2[j]);
            j++;
        }
    }

    // Add remaining elements from arr1 if any
    while (i < arr1.size()) {
        mergedArray.push_back(arr1[i]);
        i++;
    }
    // Add remaining elements from arr2 if any
    while (j < arr2.size()) {
        mergedArray.push_back(arr2[j]);
        j++;
    }

    return mergedArray;
}
```

Hidden Test Case :
Test Case 1:

```
Unset
Input:
arr1 = []
arr2 = [2, 4, 6, 8]

Output:
[2, 4, 6, 8]
```

Test Case 2:

```
Unset
Input:
arr1 = [1, 3, 5, 7]
arr2 = []

Output:
[1, 3, 5, 7]
```

Test Case 3:

```
Unset
Input:
arr1 = [1, 2, 3]
arr2 = [4, 5, 6]

Output:
[1, 2, 3, 4, 5, 6]
```

Test Case 4:

```
Unset
Input:
arr1 = [1, 2, 3, 7]
arr2 = [4, 5, 6]

Output:
[1, 2, 3, 4, 5, 6, 7]
```

Test Case 5:

```
Unset
Input:
arr1 = [1, 1, 2, 3]
arr2 = [1, 3, 4, 4]

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

## Problem Statement

## 6.Prime Number Checker

Write a function that takes an integer as input and returns `true` if it is a prime number, and `false` otherwise.

### Input

- An integer `n` (1 ≤ n ≤ 10^6).

### Output

- `true` if `n` is a prime number.
- `false` otherwise.

### Constraints

- nis a positive integer.

Sample Input:

```
Unset
Input:
n = 29
```

Sample Output:

```
Unset
Output:
true
```

Solutions:

```
C/C++


bool isPrime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true; // 2 and 3 are prime numbers

    // Eliminate multiples of 2 and 3
    if (n % 2 == 0 || n % 3 == 0) return false;

    // Check divisors from 5 up to sqrt(n)
    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    }
    return true;
}
```

Hidden Test Case

Test Case 1:

```
Unset
Input:
n = 1

Output:
false
```

Test Case 2:

```
Unset
Input:
n = 16
```

```
Output:
false
```

Test Case 3:

```Unset
Input:
n = 97

Output:
true
```

Test Case 4:

```Unset
Input:
n = 31

Output:
true
```

Test Case 5:

```Unset
Input:
n = 1000003

Output:
true
```

## 7.Find Duplicate Elements in an Array

### Problem Statement

Write a function that takes an array of integers as input and returns a list of all elements that appear more than once in the array. Each duplicate should appear only once in the output, regardless of how many times it appears in the input.

### Input

- An integer array `arr` of size N(1 ≤ N≤ 10^5).

### Output

- An array containing all unique elements that have duplicates in the input array.

### Constraints

- The elements in the array range between −10^5 and 10^5.

Sample Input:

```
Unset
Input:
arr = [4, 3, 2, 7, 8, 2, 3, 1]
```

Sample Output:

```
Unset

Output:
[2, 3]
```

Solutions:

```
C/C++


vector<int> findDuplicates(const vector<int>& arr) {
    unordered_map<int, int> elementCount;
    unordered_set<int> duplicates;

    // Count occurrences of each element
    for (int num : arr) {
        elementCount[num]++;
    }

    // Collect elements that appear more than once
    for (const auto& entry : elementCount) {
        if (entry.second > 1) {
            duplicates.insert(entry.first);
        }
    }

    // Convert set to vector
    return vector<int>(duplicates.begin(), duplicates.end());
}
```

Hidden Test Case
Test Case 1:

```
Unset
Input:
arr = [1, 2, 3, 4, 5, 6]

Output:
[]
```

Test Case 2:

```Unset
Input:
arr = [10, 10, 10, 10]

Output:
[10]
```

Test Case 3:

```Unset
Input:
arr = [7, 3, 5, 3, 5, 6, 7]

Output:
[3, 5, 7]
```

Test case 4:

```Unset
Input:
arr = [100000, -100000, 0, 0, -100000, 100000]

Output:
[100000, -100000, 0]
```

Test Case 5:

```Unset
Input:
arr = [1,2,3]

Output:
```

```
[]
```

## 8.Anagram Checker

Write a function that takes two strings as input and returns `true` if they are anagrams of each other and `false` otherwise. An anagram is when two strings can be rearranged to form one another. Both strings will contain only lowercase alphabets.

### Input

- Two strings `str1` and `str2` (1 ≤ |str1|, |str2| ≤ 10^5).

### Output

- `true` if the strings are anagrams, `false` otherwise.

### Constraints

- Both strings contain only lowercase English letters (`'a'` to `'z'`).

Sample Input:

```
Unset
Input:
str1 = "listen"
str2 = "silent"
```

Sample Output:

```
Unset
Output:
True
```

Solutions Using Hashing:

```C/C++
bool areAnagrams(string& str1,  string& str2) {
    // If lengths differ, they can't be anagrams
    if (str1.size() != str2.size()) return false;

    unordered_map<char, int> charCount;

    // Count characters in the first string
    for (char ch : str1) {
        charCount[ch]++;
    }

    // Decrement count based on the second string
    for (char ch : str2) {
        charCount[ch]--;
        if (charCount[ch] < 0) {
            return false;
        }
    }

    return true;
}
```

Hidden Test Case :

Test case 1:

```Unset
Input:
str1 = "triangle"
str2 = "integral"

Output:
True
```

Test Case 2:

```Unset
Input:
str1 = "apple"
str2 = "pale"

Output:
False
```

Test Case 3:

```Unset
Input:
str1 = "apple"
str2 = "paple"

Output:
True
```

Test Case 4:

```Unset
Input:
str1 = "anagram"
str2 = "nagaram"

Output:
True
```

Test Case 5:

```
Unset
Input:
str1 = "rat"
str2 = "car"

Output:
False
```

## 9.Binary Search

### Problem Statement

Write a function that performs a binary search on a sorted list of integers to find a target value. Return the index of the target if it exists in the list; otherwise, return `-1`.

### Input

- A sorted array `arr` of integers, with size n(1 ≤ n≤ 10^5).
- An integer `target` to search for in the array.

### Output

- The index of the `target` if found in the array; otherwise, `-1`.

### Constraints

- The function should run in O(log n) time complexity.
- The array is sorted in ascending order.

Sample Input:

```
Unset
Input:
arr = [1, 2, 4, 5, 7, 8, 9]
target = 5
```

Sample Ouput:

```
Unset
Output:
3
```

Solutions:

```cpp
C/C++
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
    vector<int> arr = {1, 2, 4, 5, 7, 8, 9};
    int target = 5;
    cout << binarySearch(arr, target) << endl;
    return 0;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
arr = [1, 2, 4, 5, 7, 8, 9]
target = 6

Output:
-1
```

Test Case 2:

```
Unset
Input:
arr = [3, 8, 10, 15, 19, 23, 32]
target = 23

Output:
5
```

Test Case 3:

```
Unset
Input:
arr = [3, 8, 10, 15, 19, 23, 32]
target = 15

Output:
-1
```

Test Case 4:

```
Unset
Input:
```

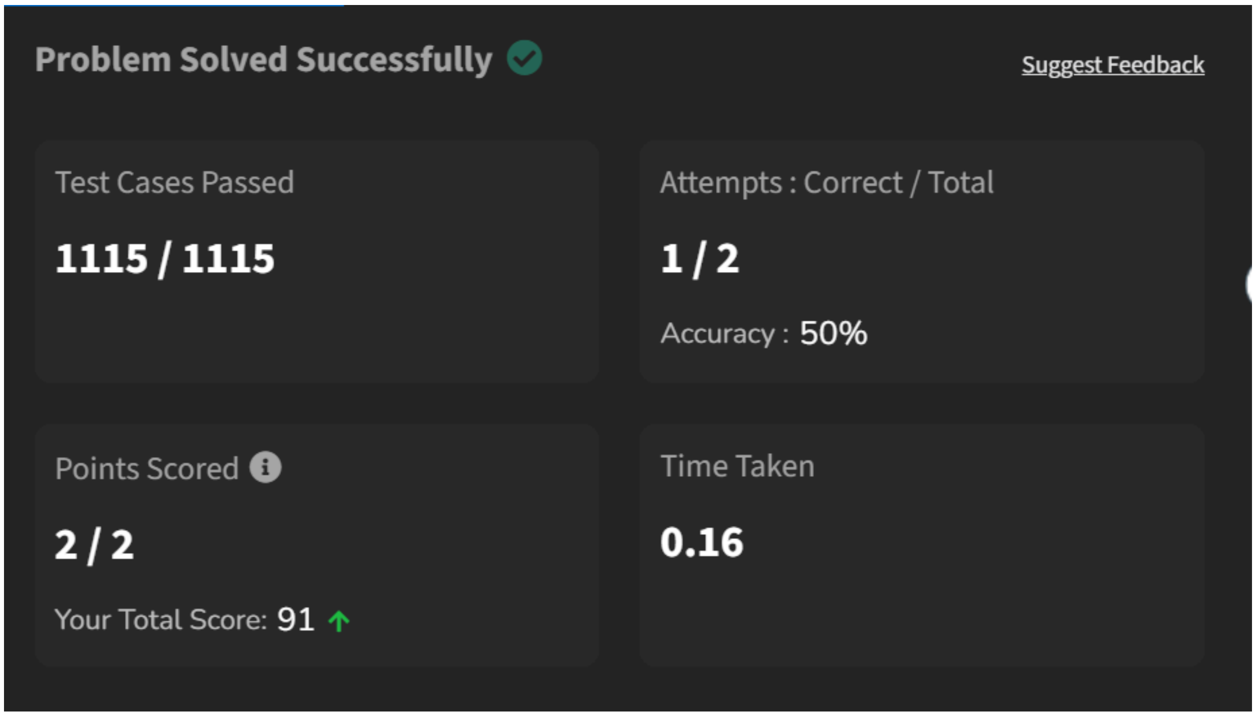
```
arr = [10, 20, 30, 40, 50]
target = 10

Output:
0
```

Test Case 5:

```
Unset
Input:
arr = [10, 20, 30, 40, 50]
target = 50

Output:
4
```



Problem Solved Successfully

Suggest Feedback

Test Cases Passed

1115 / 1115

Attempts : Correct / Total

1 / 2

Accuracy : 50%

Points Scored

2 / 2

Your Total Score: 91

Time Taken

0.16

## 10.Detect a Cycle in a Linked List

You are given the head of a singly linked list. Your task is to write a function that detects if the linked list has a cycle in it. A cycle occurs if a node's next pointer points back to a previous node in the list. If the list has a cycle, return `true`; otherwise, return `false`.

## Input

- A singly linked list represented by its head node.

## Output

- Return `true` if there is a cycle in the linked list; otherwise, return `false`.

## Constraints

- The list can contain up to 10^5 nodes.
- The value of each node is within the range of integers.

Sample Input:

```
Unset
Input:
List: 1 -> 2 -> 3 -> 4 -> null
```

Sample Output:

```
Unset
Output:
No cycle
```

Solutions:

```
C/C++

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

class LinkedListCycleDetector {
public:
    bool hasCycle(ListNode *head) {
        if (head == nullptr) return false;

        ListNode *slow = head;
        ListNode *fast = head;

        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;              // Move slow by 1 step
            fast = fast->next->next;       // Move fast by 2 steps

            if (slow == fast) {
                return true;                // Cycle detected
            }
        }
        return false;                       // No cycle
    }
};
```

Hidden Test Case

Test Case 1:

```
Unset
Input:
List: 1 -> 2 -> 3 -> 4 -> 5 -> 2 (cycle)
Output:
Cycle detected
```

Test Case 2:

```Unset
Input:
List: 1
Output:
No cycle
```

Test case 3:

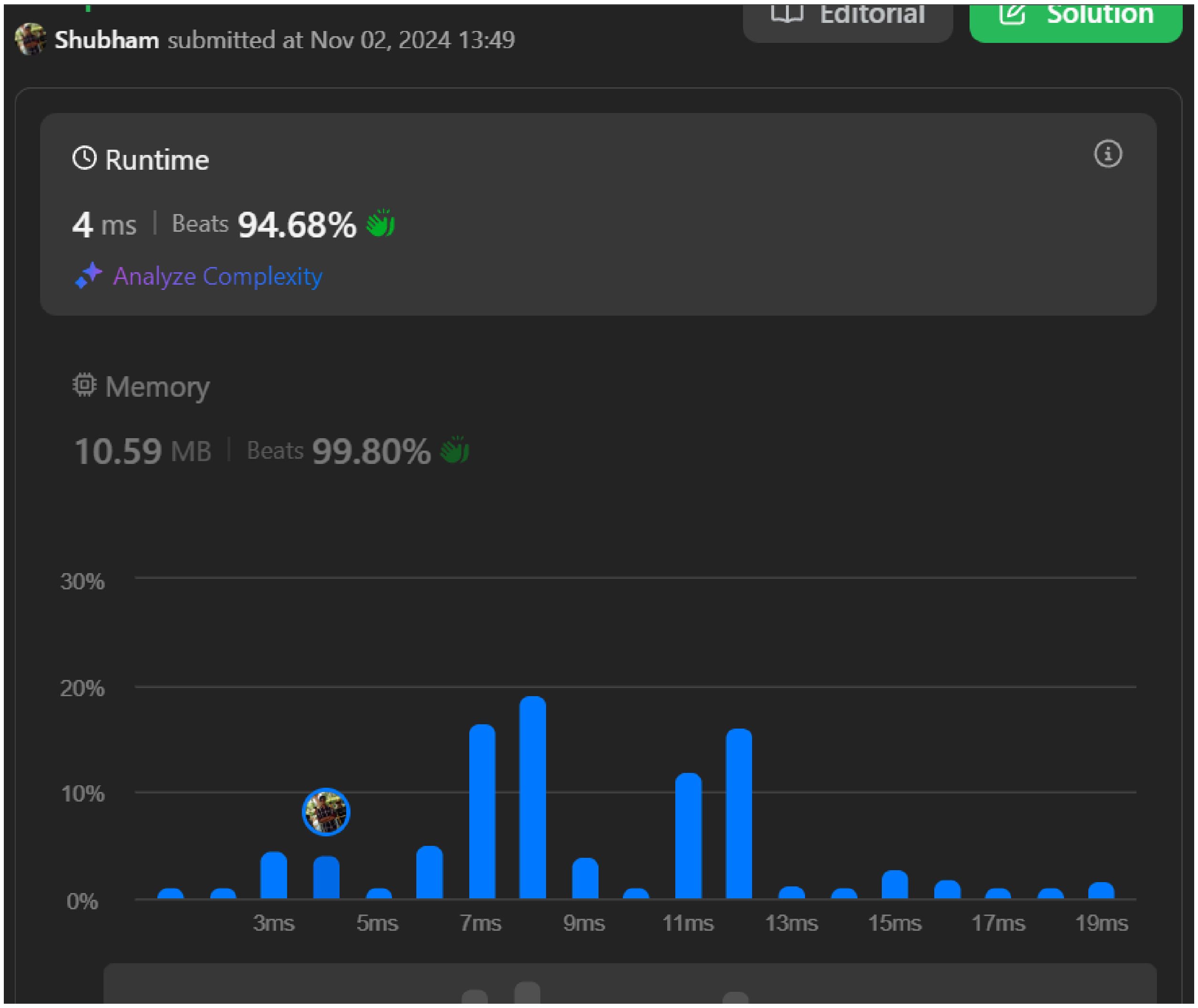
```Unset
Input:
List: 1 -> 2
Output:
No cycle
```

Test Case 4:

```Unset
Input:
List: 1 -> 2 -> 1 (cycle)
Output:
Cycle detected
```

Test Case 5:

```Unset
Input:
List: 1 -> 2 -> 3 -> 4 -> 5 -> null
Output:
No cycle
```


Shubham submitted at Nov 02, 2024 13:49

Runtime

4 ms | Beats 94.68%

Memory

10.59 MB | Beats 99.80%

## 1. Find the Missing Number in a Permutation

You are given an array `arr` of size `n - 1`, containing distinct integers from the range 1 to `n` (inclusive). This array is a permutation of the numbers from 1 to `n`, with one number missing. Your task is to find and return the missing number.

**Input Format**:

- An integer `n`, which is the size of the original permutation (`n = arr.size() + 1`).
- An array `arr` of size `n - 1`, containing distinct integers in the range from `1` to `n`.

**Output Format**:

- Return the missing integer from the permutation.

**Constraints**:

```
- 1 ≤ arr.size() ≤ 10^6
- 1 ≤ arr[i] ≤ n
```

Sample Input:

```
Unset
n = 5
arr = [1, 2, 3, 5]
```

Sample Output:

```
Unset
4
```

Solution:

```
C/C++
Solution

int findMissingNumber(vector<int>&arr, int n) {
    // Calculate the expected sum of numbers from 1 to n
    int expectedSum = n * (n + 1) / 2;

    // Calculate the sum of the given array elements
    int actualSum = 0;
    for (int i = 0; i < n - 1; i++) {
        actualSum += arr[i];
    }

    // The missing number is the difference between expectedSum and actualSum
    return expectedSum - actualSum;
}
```

Hidden Test Cases

Test Case 1:

```
Unset
Input:
n = 6
arr = [6, 4, 2, 5, 1]
Output:
3
```

Test Case 2:

```
Unset
Input:
n = 4
arr = [1, 4, 3]
Output:
2
```

Test Case 3:

```
Unset
Input:
n = 7
arr = [7, 5, 6, 3, 1, 2]
Output:
4
```

Test Case 4:

```
Unset
Input:
n = 3
arr = [1, 3]
Output:
2
```

Test Case 5:

```
Unset
Input:
n = 3
arr = [1, 2]
Output:
3
```

## 3. Find Duplicates in an Array

You are given an array `arr` of integers of length `n`. Your task is to identify all the elements that occur more than once in the array and return them in ascending order. If no element repeats, return an empty list.

**Input Format**:

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` integers.

**Output Format**:

- A list of integers representing all elements that occur more than once in the array, returned in ascending order.
- If no element repeats, return an empty list.

**Constraints**:

- `1 <= n <= 10^5` (length of the array `arr`)
- `-10^4 <= arr[i] <= 10^4` (each element of the array `arr`)

Sample Input:

```
Unset
n = 8
arr = [4, 3, 2, 7, 8, 3, 2, 1]
```

Sample output:

```
Unset
[2, 3]
```

```
C/C++


vector<int> findDuplicates(vector<int> arr) {
    vector<int> duplicates;
    int lastAdded = INT_MIN; // Variable to track the last added duplicate

    // Sort the array to group duplicates together
    sort(arr.begin(), arr.end());

    // Iterate through the array to find duplicates
    for (int i = 0; i < arr.size() - 1; i++) {
        // Check if the current element is equal to the next one and not
already added
        if (arr[i] == arr[i + 1] && lastAdded != arr[i]) {
            duplicates.push_back(arr[i]);
            lastAdded = arr[i]; // Update the last added duplicate
        }
    }

    // If no duplicates were found, add -1 to the result
    if (duplicates.empty()) {
        duplicates.push_back(-1);
    }

    return duplicates;
}
```

Hidden Test Case
Test Case 1:

```
Unset
Input:
n = 8
```

```
arr = [1, 2, 3, 4, 5, 2, 3, 1]
Output:
[1, 2, 3]
```

Test Case 2:

```
Unset
Input:
n = 5
arr = [4, 5, 6, 7, 8]
Output:
[]
```

Test Case 3:

```
Unset
Input:
n = 6
arr = [1, 2, 3, 1, 2, 3]
Output:
[1, 2, 3]
```

Test Case 4:

```
Unset
n = 7
arr = [5, 6, 5, 7, 8, 9, 9]
Output:
[5, 9]
```

Test Case 5:

```
Unset
Input:
n = 4
arr = [10, 20, 30, 10]
Output:
[10]
```

## 4. Order 0s, 1s, and 2s

You are given an array `arr` of length `n` that contains only the integers 0, 1, and 2. Your task is to sort the array in ascending order.

**Input Format**:

- An integer `n` representing the length of the array `arr`.
- An array `arr` of length `n` containing only the integers 0, 1, and 2.

**Output Format**:

- The sorted array `arr` in ascending order.

**Constraints**:

- `1 <= n <= 10^7` (length of the array `arr`)
- `arr[i] ∈ {0, 1, 2}` (each element of the array `arr`

Sample Input:

```
Unset
n = 5
arr = [2, 0, 1, 2, 0]
```

Sample output:

```
Unset
[0, 0, 1, 2, 2]
```

Solutions:

```
Unset
#include <vector>
using namespace std;

class Solution {
public:
    void sortColors(vector<int>& arr, int n) {
        int low = 0;      // Pointer for the next position of 0
        int mid = 0;      // Pointer for the current element
        int high = n - 1; // Pointer for the next position of 2

        while (mid <= high) {
            if (arr[mid] == 0) {
                // Swap arr[mid] with arr[low] and move both pointers
                swap(arr[mid], arr[low]);
                low++;
                mid++;
            } else if (arr[mid] == 1) {
                // Just move the mid pointer
                mid++;
            } else {
                // Swap arr[mid] with arr[high] and move high pointer
                swap(arr[mid], arr[high]);
                high--;
            }
        }
    }
};
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
n = 5
arr = [2, 0, 1, 2, 0]
Output:
[0, 0, 1, 2, 2]
```

Test Case 2:

```
C/C++
Input:
```

```
n = 4
arr = [1, 2, 0, 1]
Output:
[0, 1, 1, 2]
```

Test Case 3:

```
Unset
Input:
n = 6
arr = [2, 2, 2, 1, 1, 0]
Output:
[0, 1, 1, 2, 2, 2]
```

Test Case 4:

```
Unset
Input:
n = 3
arr = [0, 0, 1]
Output:
[0, 0, 1]
```

Test Case 5:

```
Unset
Input:
n = 7
arr = [1, 0, 2, 1, 0, 1, 2]
Output:
[0, 0, 1, 1, 1, 2, 2]
```

## 5. Majority Element Detection

You are given an array `arr` of length `n`. Your task is to find the majority element in the array. A majority element is defined as an element that appears more than `n/2` times in the array. If no majority element exists, return `-1`.

**Input Format**:

- An integer `n` representing the length of the array `arr`.
- An array `arr` of length `n` containing integers.

**Output Format**:

- An integer representing the majority element in the array.
- If no majority element exists, return `-1`.

**Constraints**:

- `1 <= n <= 10^5` (length of the array `arr`)
- `-10^9 <= arr[i] <= 10^9` (each element of the array `arr`)

Sample Input:

```
Unset
n = 7
arr = [2, 2, 1, 1, 2, 2, 3]
```

Sample Output:

```
Unset
2
```

Solutions:

```
C/C++
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int findMajorityElement(int n, vector<int>& arr) {
        unordered_map<int, int> countMap;  // Map to count occurrences of each
element

        // Count occurrences of each element
        for (int num : arr) {
            countMap[num]++;
            // If count exceeds n/2, return the majority element
            if (countMap[num] > n / 2) {
                return num;
            }
        }

        // If no majority element exists, return -1
        return -1;
    }
};
```

Hidden Test Cases:
Test Case 1:

```
Unset
n = 8
arr = [6, 6, 6, 1, 2, 2, 6, 6]
Output:
6
```

Test Case 2:

```
Unset
n = 8
arr = [5, 5, 5, 1, 2, 2, 5, 5]
Output:
5
```

Test Case 3:

```
Unset
Input:
n = 4
```

```
arr = [3, 3, 3, 1]
Output:
3
```

Test Case 4:

```
Unset
Input:
n = 6
arr = [4, 4, 4, 2, 4, 1]
Output:
4
```

Test Case 5:

```
Unset
Input:
n = 5
arr = [1, 1, 2, 2, 3]
Output:
-1
```

## 6.First Equilibrium Point in an Array

You are given an array `arr` of non-negative numbers and an integer `n` representing the length of the array. The task is to find the first equilibrium point in the array, where the sum of all elements before that index is equal to the sum of elements after it. Return the equilibrium point in 1-based indexing, or return `-1` if no such point exists.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` non-negative integers.

**Output Format:**

- An integer representing the 1-based index of the first equilibrium point.

- If no equilibrium point exists, return `-1`.

**Constraints:**

- `1 <= n <= 10^5` (length of the array `arr`)
- `0 <= arr[i] <= 10^9` (value of each element in the array `arr`)

Sample Input:

```
Unset
n = 5
arr = [1, 3, 5, 2, 2]
```

Sample Output:

```
Unset
3
The sum of elements before index 3 (1-based) is 1 + 3 = 4, and the sum of
elements after index 3 is 2 + 2 = 4. Therefore, the equilibrium point is at
index 3.
```

Solutions using prefix sum technique

```
C/C++

int equilibriumPoint(vector<long long> &arr, int n) {
    long long curSum = 0;
    long long totalSum = accumulate(arr.begin(), arr.end(), 0LL);  // Calculate
total sum of the array

    for(int i = 0; i < n; i++) {
        // Check if current sum is equal to the remaining sum after subtracting
arr[i]
        if(curSum == totalSum - curSum - arr[i]) {
            return i + 1;  // Return 1-based index
        }
        curSum += arr[i];  // Update current sum by adding the current element
    }
    return -1;  // Return -1 if no equilibrium point is found
}
```

Hidden Test Cases:

Test Case 1:

```
Unset
Input:
n = 6
arr = [1, 3, 4, 2, 2, 6]
Output:
4
```

Test Case 2:

```
Unset

Input:
n = 4
arr = [1, 2, 3, 4]
Output:
-1
```

Test Case 3:

```
Unset
Input:
n = 6
arr = [1, 2, 1, 3, 1, 2]
Output:
4
```

Test Case 4:

```
Unset

Input:
n = 6
arr = [1, 2, 1, 3, 1, 2]
Output:
-1
```

Test Case 5:

```
Unset
Input:
n = 1
arr = [100]
Output:
1
```

## 7. Longest Subarray with Sum Equal to k

You are given an array `arr` containing `n` integers and an integer `k`. Your task is to determine the length of the longest subarray where the sum of its elements equals the specified value `k`.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` integers.
- An integer `k` representing the target sum.

**Output Format:**

- An integer representing the length of the longest subarray with a sum equal to `k`.

**Constraints:**

- `1 <= n <= 10^5` (length of the array `arr`)
- `-10^9 <= arr[i] <= 10^9` (value of each element in the array `arr`)
- `-10^9 <= k <= 10^9` (value of the target sum)

Sample Input:

```
Unset
n = 5
arr = [1, 2, 3, -2, -1]
k = 3
```

Sample Output:

```
Unset
3
```

Solution with prefix sum:

```
C/C++
int lengthOfLongestSubarray(int arr[], int size, int targetSum) {
    unordered_map<int, int> prefixSumMap;  // To store the first occurrence of
each prefix sum
    int maxLength = 0;  // Maximum length of subarray found
    int currentSum = 0;  // Current prefix sum

    for (int index = 0; index < size; index++) {
        currentSum += arr[index];  // Update the current prefix sum
        int requiredSum = currentSum - targetSum;  // Calculate the required
sum

        // Check if the current prefix sum equals the target sum
        if (currentSum == targetSum) {
            maxLength = max(maxLength, index + 1);  // Update the max length
        }

        // Check if the required prefix sum exists in the map
        if (prefixSumMap.find(requiredSum) != prefixSumMap.end()) {
            maxLength = max(maxLength, index - prefixSumMap[requiredSum]);  //
Update max length based on found prefix
        }

        // Store the first occurrence of the current prefix sum if not already
present
        if (prefixSumMap.find(currentSum) == prefixSumMap.end()) {
            prefixSumMap[currentSum] = index;  // Save the index of the first
occurrence
        }
    }
    return maxLength;  // Return the length of the longest subarray
}
```

Hidden Test Case:

Test Case 1:

```
Unset

Input:

    n = 7

    arr = [1, 2, 3, -2, -1, 4, 5]

    k = 6

Output:

    3
```

Test Case 2:

```
Unset

Input:

    n = 8

    arr = [1, -2, 3, 0, 4, -1, 2, 1]

    k = 5

Output:

    5
```

Test Case 3:

```
Unset

Input:

    n = 6

    arr = [5, 1, -3, 2, 3, 0]

    k = 4

Output:

    4
```

Test Case 4:

```
Unset

Input:

    n = 5

    arr = [10, 5, 2, 7, 1]

    k = 15

Output:

    2
```

Test Case 5:

```
Unset

Input:

    n = 4

    arr = [1, 2, 3, 4]

    k = 10

Output:
```

```
0
```

## 8.Maximum Product Subarray

Given an array `arr[]` that contains both positive and negative integers, as well as zeros, your task is to determine the maximum product that can be obtained from any contiguous subarray of `arr`.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` integers, which may include positive numbers, negative numbers, and zeros.

**Output Format:**

- An integer representing the maximum product that can be obtained from any contiguous subarray of `arr`.

**Constraints:**

- `1 <= n <= 10^5` (length of the array `arr`)
- `-10^6 <= arr[i] <= 10^6` (value of each element in the array `arr`)

Sample Input:

```
Unset
n = 5
arr = [1, 0, 2, -3, 4]
```

Sample Output:

```
Unset
4
```

Solutions using prefix and suffix multiplication approach:

```
C/C++
```

Test Case 1:

```
Unset

Input:

   n = 6

   arr = [6, -3, -10, 0, 2]

Output:

   180
```

Test Case 2:

```
Unset

Input:

   n = 4

   arr = [-2, -3, 0, -2]

Output:

   6
```

Test Case 3:

```
Unset

Input:

   n = 5

   arr = [-1, -2, -3, -4, -5]

Output:

   120
```

Test Case 4:

```
Unset

Input:

   n = 7

   arr = [2, 3, -2, 4, -1, 0, 5]

Output:

   48
```

Test Case 5:

```
Unset

Input:

   n = 3
```

```
    arr = [0, 0, 0]

Output:

    0
```

## 9.Left Rotation of an Array

Given an unsorted array `arr[]` of length `n`, rotate the array to the left (in the counterclockwise direction) by `k` steps, where `k` is a positive integer. Perform the rotation in place, modifying the original array.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An integer `d` representing the number of steps to rotate the array.
- An array `arr` of `n` integers.

**Output Format:**

- The rotated array `arr`, after shifting elements `d` steps to the left.

**Constraints:**

- `1 <= n <= 10^5` (length of the array `arr`)
- `1 <= k <= 10^8` (number of steps to rotate)
- `1 <= arr[i] <= 10^9` (value of each element in the array)

Sample Input:

```
Unset
n = 3
k = 2
arr = [9, 8, 7]
```

Sample Output:

```
Unset
[7, 9, 8]
```

Solution:

```
C/C++
void rotateArr(vector<int>& arr, int k, int n) {
    k %= n;  // In case k is greater than n, reduce it within the bounds
    reverse(arr.begin(), arr.begin() + k);   // Reverse the first k elements
    reverse(arr.begin() + k, arr.end());     // Reverse the remaining elements
    reverse(arr.begin(), arr.end());         // Reverse the entire array
}
```

Hidden Test Case:
Test Case 1:

```
Unset

Input:

   n = 3

   k = 1

   arr = [15, 25, 35]

Output:

   [25, 35, 15]
```

Test Case 2:

```
Unset

Input:
    n = 4

    k = 2

    arr = [4, 3, 2, 1]

Output:

    [2, 1, 4, 3]
```

Test Case 3:

```
Unset

Input:

    n = 5

    k = 5

    arr = [9, 8, 7, 6, 5]

Output:

    [9, 8, 7, 6, 5]
```

Test Case 4:

```
Unset

Input:

    n = 8

    k = 11

    arr = [11, 22, 33, 44, 55, 66, 77, 88]

Output:

    [44, 55, 66, 77, 88, 11, 22, 33]
```

Test Case 5:

```
Unset

Input:

   n = 6

   k = 4

   arr = [1, 2, 3, 4, 5, 6]

Output:

   [5, 6, 1, 2, 3, 4]
```

## 10.Find the Smallest Missing Positive Integer

You are given an integer array `arr[]` of length `n`. Your task is to find the smallest positive integer that is missing from the array.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` integers, which may contain both positive and negative numbers.

**Output Format:**

- A single integer representing the smallest positive integer that is missing from the array.

**Constraints:**

- `1 <= n <= 10^6` (length of the array `arr`)
- `-10^6 <= arr[i] <= 10^6` (value of each element in the array `arr`)

Sample Input:

```
Unset

n = 5

arr = [3, 4, -1, 1]
```

Sample Output:

```
Unset

2
```

Solution:

```
C/C++
int findMissingPositive(vector<int>& arr) {
    int n = arr.size();

    // Rearrange the numbers in the array
    for (int i = 0; i < n; i++) {
        while (arr[i] > 0 && arr[i] <= n && arr[arr[i] - 1] != arr[i]) {
            swap(arr[arr[i] - 1], arr[i]);
        }
    }

    // Identify the smallest missing positive integer
    for (int i = 0; i < n; i++) {
        if (arr[i] != i + 1) {
            return i + 1;
        }
    }

    return n + 1;
}
```

Hidden Test Case:

Test Case 1:

```
Unset

Input:

    n = 8

    arr = [1, 2, 3, 4, 5, 6, 7, 8]

Output:
    9
```

Test Case 2:

```Unset
Input:
    n = 4
    arr = [2, 3, 1, 5]
Output:
    4
```

Test Case 3:

```Unset
Input:
    n = 5
    arr = [-1, -2, -3, -4, -5]
Output:
    1
```

Test Case 4:

```Unset
Input:
    n = 6
    arr = [7, 8, 9, 10, 11, 12]
```

```
Output:

    1
```

Test Case 5:

```
Unset

Input:

    n = 7

    arr = [1, 2, 0, -1, -2, 3, 4]

Output:

    5
```

## 11. K-th Smallest Element in an Array

You are given an integer array `arr[]` and an integer `k`, where `k` is smaller than the size of the array. Your task is to find the k-th smallest element in the array.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` integers.
- An integer `k`, where `1 <= k <= n`.

**Output Format:**

- A single integer representing the k-th smallest element in the array.

**Constraints:**

- `1 <= n <= 10^5` (length of the array `arr`)
- `-10^6 <= arr[i] <= 10^6` (value of each element in the array `arr`)

- `1 <= k <= n` (k-th position)

Sample Input:

```
Unset
n = 5
arr = [3, 1, 5, 2, 4]
k = 3
```

Sample Output:

```
Unset
3
```

Solution using priority queue:

```
C/C++
int findKthSmallest(vector<int>& arr, int k) {
    // Create a max-heap using a priority queue
    priority_queue<int> maxHeap;
    int n = arr.size();

    // Iterate through the array
    for (int i = 0; i < n; i++) {
        maxHeap.push(arr[i]);  // Add the current element to the heap
        if (maxHeap.size() > k) {
            maxHeap.pop();  // Remove the largest element if heap size exceeds
k
        }
    }

    // The top element is the k-th smallest element
    return maxHeap.top();
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
    n = 6
    arr = [7, 10, 4, 3, 20, 15]
    k = 4
Output:
    10
```

Test Case 2:

```
Unset
Input:
    n = 7
    arr = [12, 3, 5, 7, 19, 10, 2]
    k = 2
Output:
    3
```

Test Case 3:

```
Unset
Input:
    n = 4
    arr = [1, 2, 3, 4]
    k = 1
```

```
Output:
    1
```

Test Case 4:

```Unset
Input:
    n = 8
    arr = [8, 9, 10, 5, 6, 4, 3, 2]
    k = 5
Output:
   6
```

Test Case 5:

```Unset
Input:
    n = 5
    arr = [15, 20, 5, 30, 10]
    k = 2


Output:
    10
```

## 12. Find union of two sorted array

You are given two sorted arrays `a[]` and `b[]` , where both arrays may contain duplicate elements. Your task is to return the union of the two arrays, ensuring the result is in sorted order, without duplicates.

**Input Format:**

- An integer `n` representing the length of array `a[]`.
- An integer `m` representing the length of array `b[]`.
- An array `a[]` of `n` sorted integers.
- An array `b[]` of `m` sorted integers.

**Output Format:**

- An array containing the union of the two arrays `a[]` and `b[]`, with unique elements in sorted order.

**Constraints:**

- `1 <= n, m <= 10^5` (length of the arrays)
- `-10^6 <= a[i], b[i] <= 10^6` (values of elements in the arrays)

Sample Input:

```
Unset
n = 5
m = 4
a = [1, 2, 2, 3, 4]
b = [2, 3, 3, 5]
```

Sample output:

```
Unset
[1, 2, 3, 4, 5]
```

Solution using merge algorithm of merge sort

```
C/C++
vector<int> findUnion(vector<int> &a, vector<int> &b) {
    vector<int> ans;
    int i = 0, j = 0;

    // Traverse both arrays simultaneously
    while (i < a.size() && j < b.size()) {
        // Add smaller element or equal element to the result
        if (a[i] < b[j]) {
            if (ans.empty() || ans.back() != a[i]) ans.push_back(a[i]);
            i++;
        } else if (a[i] > b[j]) {
            if (ans.empty() || ans.back() != b[j]) ans.push_back(b[j]);
            j++;
        } else {  // Both elements are equal
            if (ans.empty() || ans.back() != a[i]) ans.push_back(a[i]);
            i++;
            j++;
        }
    }

    // Add remaining elements of array a
    while (i < a.size()) {
        if (ans.empty() || ans.back() != a[i]) ans.push_back(a[i]);
        i++;
    }

    // Add remaining elements of array b
    while (j < b.size()) {
        if (ans.empty() || ans.back() != b[j]) ans.push_back(b[j]);
        j++;
    }

    return ans;
}
```

Hidden Test Case:

Test Case1:

```
Unset

Input:

    n = 4

    m = 4

    a = [10, 20, 20, 40]

    b = [5, 20, 30, 40]

Output:

    [5, 10, 20, 30, 40]
```

Test Case 2:

```
Unset

Input:
n = 3
m = 0
a = [7, 8, 9]
b = []

Output:
[7, 8, 9]
```

Test Case 3:

```
Unset

Input:
n = 3
m = 4
a = [1, 1, 1]
b = [2, 2, 2]
```

```
Output:
[1, 2]
```

Test Case 4:

```
Unset

Input:
n = 0
m = 3
a = []
b = [3, 4, 5]

Output:
[3, 4, 5]
```

Test Case 5:

```
Unset

Input:

    n = 3

    m = 3

    a = [1, 3, 5]

    b = [2, 4, 6]

Output:

    [1, 2, 3, 4, 5, 6]
```

## 12.Find the Single Unique Element in Array

You are given a non-empty array `arr` where every element appears twice except for one element that appears only once. Your task is to identify that single unique element.

You must implement an efficient solution that runs in linear time and uses only constant extra space.

## Input Format:

- An integer `n` representing the size of the array `arr`.
- An array `arr` of size `n` consisting of integers, where each element appears twice except for one element that appears only once.

## Output Format:

- An integer representing the unique element in the array `arr` that appears only once.

## Constraints:

- `1 <= n <= 3 * 10^4` (size of the array `arr`)
- `n` is odd (as there is always one unique element).
- `-3 * 10^4 <= arr[i] <= 3 * 10^4` (value of each element in the array)
- All elements except one occur twice.

Sample Input:

```
Unset
n = 9
arr = [10, 9, 8, 9, 8, 6, 6, 7, 10]
```

Sample output:

```
Unset
7
```

Solution using XOR properties:

```
C/C++
int findUniqueElement(vector<int>& arr) {
    int unique = 0;
    for (int num : arr) {
        unique ^= num;
    }
    return unique;
}
```

Hidden Test Cases:

Test Case 1:

```
Unset
Input:
n = 5
arr = [2, 2, 1, 4, 4]
Output:
1
```

Test Case 2:

```
Unset
Input:
n = 7
arr = [5, 3, 5, 2, 2, 7, 7]
Output:
3
```

Test Case 3:

```
Unset
Input:
n = 9
arr = [10, 9, 8, 9, 8, 6, 6, 7, 10]
Output:
7
```

Test Case 4:

```
Unset
Input:
n = 3
arr = [1, 1, -3]
Output:
-3
```

Test Case 5:

```
Unset
Input:
n = 5
arr = [0, 2, 2, 0, -1]
Output:
-1
```

## 13. Next Greater Permutation

You are given an array arr of integers. Your task is to rearrange the elements of the array into the next lexicographically greater permutation of numbers. If such a permutation is not possible, you should rearrange the array to the lowest possible order (i.e., sort it in ascending order).

### Input Format:

- An integer **n** representing the length of the array **arr**.
- An array **arr** of **n** integers.

### Output Format:

- An array representing the next greater permutation of the input array. If no such permutation exists, return the sorted array.

### Constraints:

- $1 \leq n \leq 10^5$ (length of the array **arr**)
- $-10^9 \leq arr[i] \leq 10^9$ (value of each element in the array **arr**)

Sample Input:

```
Unset
n = 3
arr= [1, 2, 3]
```

Sample Output:

```
Unset
[1, 3, 2]
```

Solutions:

```
C/C++
void findNextPermutation(vector<int>& arr) {
    int n = arr.size();
    int i, j;

    // Step 1: Find the largest index i such that arr[i] < arr[i + 1]
    for (i = n - 2; i >= 0; i--) {
        if (arr[i] < arr[i + 1]) {
            break;
        }
    }

    // Step 2: If such an index exists, find the largest index j greater than i
such that arr[j] > arr[i]
    if (i >= 0) {
        for (j = n - 1; j > i; j--) {
            if (arr[j] > arr[i]) {
                swap(arr[i], arr[j]); // Swap values at i and j
                break;
            }
        }
    }

    // Step 3: Reverse the subarray from arr[i + 1] to the end of the array
    reverse(arr.begin() + i + 1, arr.end());
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:

    n = 3

    arr = [1, 2, 3]

Output:

    [1, 3, 2]
```

Test Case 2:

```
Unset
Input:

    n = 4

    arr = [4, 3, 2, 1]

Output:

    [1, 2, 3, 4]
```

Test Case 3:

```
Unset
Input:

    n = 5

    arr = [1, 1, 5, 3, 2]

Output:

    [1, 2, 1, 3, 5]
```

Test Case 4:

```
Unset
Input:

   n = 6

   arr = [2, 3, 6, 5, 4, 1]

Output:

   [2, 4, 1, 3, 5, 6]
```

Test Case 5:

```
Unset
Input:

   n = 2

   arr = [3, 3]

Output:

   [3, 3]
```

## 14. Rotate Matrix by 90 Degrees

You are given a square matrix of size `n x n`. Your task is to rotate the matrix by 90 degrees in a clockwise direction.

You need to perform the rotation in-place, meaning you should modify the given matrix directly without using any additional space for another matrix.

### Input Format:

- An integer `n`, representing the size of the matrix.
- A 2D array of integers representing the `n x n` matrix.

### Output Format:

- The modified matrix after rotating it by 90 degrees clockwise.

## Constraints:

- 1 ≤ n ≤ 50 (dimension of the matrix)
- 0 ≤ matrix[i][j] ≤ 1000 (value of each element in the matrix)

Sample Input:

```
Unset
n = 3
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
```

Sample Output:

```
Unset
[[7, 4, 1],
 [8, 5, 2],
 [9, 6, 3]]
```

Solutions:

```
C/C++
void rotateMatrixBy90(vector<vector<int>>& matrix) {
    int n = matrix.size();

    // Transpose the matrix by swapping elements
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < row; col++) {
            swap(matrix[row][col], matrix[col][row]);
        }
    }

    // Reverse each row to complete the rotation
    for (int i = 0; i < n; i++) {
        reverse(matrix[i].begin(), matrix[i].end());
    }
}
```

```
```

Hidden Test Case:

Test Case 1:

```
Unset

Input:
n = 3
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Output:
[[7, 4, 1], [8, 5, 2], [9, 6, 3]]
```

Test Case 2:

```
Unset
Input:
n = 4
matrix = [[5, 1, 9, 11], [2, 4, 8, 10], [13, 3, 6, 7], [15, 14, 12, 16]]
Output:
[[15, 13, 2, 5], [14, 3, 4, 1], [12, 6, 8, 9], [16, 7, 10, 11]]
```

Test Case 3:

```
Unset
Input:
n = 2
matrix = [[1, 2], [3, 4]]
Output:
[[3, 1], [4, 2]]
```

Test Case 4:

```
Unset
Input:
```

```
n = 1
matrix = [[1]]
Output:
[[1]]
```

Test Case 5:

```
Unset
Input:
n = 5
matrix = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18,
19, 20], [21, 22, 23, 24, 25]]
Output:
[[21, 16, 11, 6, 1], [22, 17, 12, 7, 2], [23, 18, 13, 8, 3], [24, 19, 14, 9,
4], [25, 20, 15, 10, 5]]
```

## 15.Count Subarrays with Target Sum

You are given an array of integers `arr[]` of length `n` and an integer `k`. Your task is to return the total number of subarrays in `arr` whose sum is equal to `k`. A subarray is defined as a contiguous non-empty sequence of elements within the array.

**Input Format:**

- An integer `n` representing the length of the array `arr`.
- An array `arr` of `n` integers.
- An integer `k` representing the target sum of the subarrays.

**Output Format:**

- An integer representing the total number of subarrays in `arr` whose sum equals `k`.

**Constraints:**

- 1 ≤ `n` ≤ 10^5
- -10^4 ≤ `arr[i]` ≤ 10^4 (for each element in the array)

- -10^9 ≤ k ≤ 10^9

Sample Input:

```
Unset
Input:
n = 5
arr = [1, 1, 1]
k = 2
```

Sample Output:

```
Unset
Output:
2
```

Solutions with prefix sum:

```
C/C++
int countSubarraySum(vector<int>& arr, int k) {
    int currentSum = 0, count = 0;
    unordered_map<int, int> prefixSumFrequency;
    prefixSumFrequency[0] = 1; // To handle cases where the sum is equal to k
directly

    for (int i = 0; i < arr.size(); i++) {
        currentSum += arr[i];

        // Check if there exists a prefix sum such that currentSum - prefixSum
= k
        if (prefixSumFrequency.find(currentSum - k) !=
prefixSumFrequency.end()) {
            count += prefixSumFrequency[currentSum - k];
        }

        // Increment the frequency of the current prefix sum
        prefixSumFrequency[currentSum]++;
    }
    return count;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
n = 4
arr = [2, 4, 6, 8]
k = 10
Output:
0
```

Test Case 2:

```
Unset
Input:
n = 6
arr = [3, 4, 7, -7, 1, 2]
k = 7
Output:
4
```

Test Case 3:

```
Unset
n = 4
arr = [-1, -1, 1, 1]
k = 0
Output:
2
```

Test Case 4:

```Unset
n = 3
arr = [1, 2, 3]
k = 3
Output:
2
```

Test Case 5:

```Unset
n = 5
arr = [1, 2, 3, 6, 9]
k = 30
Output:
0
```

## 16.Find All Unique Triplets with Zero Sum

Given an integer array `nums`, return all unique triplets `[nums[i], nums[j], nums[k]]` where `i`, `j`, and `k` are distinct indices, and the sum of `nums[i] + nums[j] + nums[k]` equals 0.

**Input Format:**

- An integer array `nums` of length `n`.

**Output Format:**

- A list of all unique triplets `[nums[i], nums[j], nums[k]]` where the sum of the three elements is equal to 0. Each triplet should be sorted in ascending order, and no duplicate triplets should be included in the output.

**Constraints:**

- `1 <= n <= 10^4` (length of the array `nums`)
- `-10^5 <= nums[i] <= 10^5` (elements of the array `nums`)

Sample Input:

```
Unset
Input :
nums = [-1, 0, 1, 2, -1, -4]
```

Sample Output:

```
Unset
Output:
[[-1, -1, 2], [-1, 0, 1]]
```

Solutions using two pointer approach:

```
C/C++
vector<vector<int>> findTriplets(vector<int>& nums) {
    vector<vector<int>> result;
    int n = nums.size();

    // Sort the input array to make the process easier
    sort(nums.begin(), nums.end());

    for (int i = 0; i < n - 2; i++) {
        // Skip duplicate elements to avoid repeating the same triplets
        if (i > 0 && nums[i] == nums[i - 1]) {
            continue;
        }

        // Use two-pointer technique to find the pairs that sum up to the
required value
        int left = i + 1, right = n - 1;
        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];

            if (sum == 0) {
                // Found a valid triplet
                result.push_back({nums[i], nums[left], nums[right]});

                // Skip duplicates for left and right
                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right--;

                // Move the pointers after adding the valid triplet
                left++;
                right--;
            }
            else if (sum < 0) {
                // Move the left pointer if sum is smaller
                left++;
            }
            else {
                // Move the right pointer if sum is larger
                right--;
            }
        }
    }

    return result;
}
```

Hidden Test Case:
Test Case 1:

```
Unset
Input:
n = 6
arr = [-4, -2, -2, 0, 2, 4]
Output:
[[-4, 0, 4], [-2, -2, 4], [-2, 0, 2]]
```

Test Case 2:

```
Unset
Input:
n = 4
arr = [1, 2, 3, 4]
Output:
[]
```

Test Case 3:

```
Unset
Input:
n = 5
arr = [0, 0, 0, 0, 0]
Output:
[[0, 0, 0]]
```

Test Case 4:

```
Unset
Input:
n = 4
arr = [1, -1, -1, 0]
Output:
[[-1, -1, 2], [-1, 0, 1]]
```

Test Case 5:

```
Unset
Input:
n = 5
arr = [-2, 0, 1, 1, 2]
Output:
[[-2, 0, 2], [-2, 1, 1]]
```

## 17 Count Subarrays with XOR Equal to B

You are given an array `A[]` of integers and an integer `B`. Your task is to find the total number of subarrays whose bitwise XOR of all elements equals to `B`.

### Input Format:

- An integer `n` representing the length of the array `A[]`.
- An array `A[]` of size `n` consisting of integers.
- An integer `B` representing the target XOR value.

### Output Format:

- An integer representing the total number of subarrays whose XOR is equal to B.

## Constraints:

- 1≤n≤1051 \leq n \leq 10^51≤n≤105 (Length of the array)
- −109≤A[i]≤109-10^9 \leq A[i] \leq 10^9−109≤A[i]≤109 (Values of the elements in the array)
- −109≤B≤109-10^9 \leq B \leq 10^9−109≤B≤109 (Target XOR value)

Sample Input:

```
Unset
Input: n = 5
A = [4, 2, 2, 6, 4]
B = 6
```

Sample Output:

```
Unset
Output: 4
```

Solutions using Prefix XOR:

```
C/C++
int countSubarraysWithXor(vector<int>& A, int B) {
    unordered_map<int, int> xorMap; // To store XOR frequencies
    int count = 0, currentXor = 0;

    for (int i = 0; i < A.size(); i++) {
        currentXor ^= A[i];

        // If current XOR is equal to B, increment the count
        if (currentXor == B) {
            count++;
        }
        // If there exists a prefix with XOR equal to currentXor^B, increment
count
        int requiredXor = currentXor ^ B;
        if (xorMap.find(requiredXor) != xorMap.end()) {
            count += xorMap[requiredXor];
        }

        // Increment the frequency of the current XOR
        xorMap[currentXor]++;
    }

    return count;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
n = 4
A = [5, 6, 7, 8]
B = 5
Output:
2
```

Test Case 2:

```
Unset
Input:
n = 6
A = [1, 2, 3, 4, 5, 6]
B = 7
Output:
1
```

Test Case 3:

```
Unset
Input:
n = 3
A = [1, 1, 1]
B = 0
Output:
2
```

Test Case 4:

```
Unset
Input:
n = 4
A = [1, 2, 3, 4]
B = 4
Output:
2
```

Test Case 5:

```
Unset
Input:
n = 5
A = [1, 2, 2, 2, 1]
B = 1
Output:
6
```

## 18. Merge Overlapping Intervals

You are given an array of intervals, where each interval is represented as a pair `[start, end]`. Your task is to merge all overlapping intervals and return an array of non-overlapping intervals that cover all the intervals in the input.

### Input Format:

- An integer `n` representing the number of intervals.

- A 2D array `intervals[]` of size `n`, where each element is an interval `[start, end]` with `start <= end`.

## Output Format:

- A 2D array representing the merged non-overlapping intervals.

## Constraints:

- 1≤n≤1041 \leq n \leq 10^41≤n≤104 (Number of intervals)
- 0≤start≤end≤1040 \leq \text{start} \leq \text{end} \leq 10^40≤start≤end≤104 (Bounds of each interval)

Sample Input:

```
Unset
Input: n = 4
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
```

Sample Output:

```
Unset
Output: [[1, 6], [8, 10], [15, 18]]
```

Solution with greedy approach:

```
C/C++
vector<vector<int>> mergeIntervals(vector<vector<int>>& intervals) {
    // If there are no intervals, return an empty array
    if (intervals.empty()) return {};

    // Sort intervals by their starting point
    sort(intervals.begin(), intervals.end());

    // This will store the merged intervals
    vector<vector<int>> merged;

    // Loop through each interval
    for (const auto& interval : intervals) {
        // If the merged list is empty or there is no overlap, add the current
interval
        if (merged.empty() || merged.back()[1] < interval[0]) {
            merged.push_back(interval);
        }
        // If the intervals overlap, merge them by updating the end of the last
interval
        else {
            merged.back()[1] = max(merged.back()[1], interval[1]);
        }
    }

    // Return the merged intervals
    return merged;
}
```

Hidden Test Case
Test Case 1:

```
Unset
Input:
n = 4
intervals = [[1, 2], [2, 3], [3, 4], [5, 6]]
Output:
[[1, 4], [5, 6]]
```

Test Case 2:

```
Unset
Input:
n = 1
intervals = [[5, 7]]
Output:
[[5, 7]]
```

Test Case 3:

```
Unset
Input:
n = 3
intervals = [[7, 9], [1, 3], [2, 6]]
Output:
[[1, 6], [7, 9]]
```

Test Case 4:

```
Unset
Input:
n = 2
intervals = [[1, 10], [2, 6]]
Output:
[[1, 10]]
```

Test Case 5:

```
Unset
Input:
n = 5
intervals = [[1, 3], [5, 8], [2, 4], [6, 10], [12, 15]]
Output:
[[1, 4], [5, 10], [12, 15]]
```

## 19.Find the Repeating and Missing Numbers

You are given a read-only array `arr[]` of size `N` with integers ranging from 1 to N. Every integer appears exactly once, except for one number `A` which appears twice, and another number `B` which is missing. Your task is to find the repeating number `A` and the missing number `B`.

### Input Format:

- An integer `N`, representing the size of the array.
- An array `arr[]` of size `N`, consisting of integers from the range `[1, N]`.

### Output Format:

- Return a pair of integers where the first value is the repeating number A and the second value is the missing number B.

## Constraints:

- 2≤N≤1052 \leq N \leq 10^52≤N≤105
- All elements of arr[] are within the range [1, N].

Sample Input:

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

Sample output:

```
Unset
{5,6} Repeating number: 5, Missing number: 6
```

Solutions using Maths:

```
C/C++

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

    // Calculate the expected sum and sum of squares for numbers from 1 to n
    long long expectedSum = (n * (n + 1)) / 2;
    long long expectedSumSquares = (n * (n + 1) * (2 * n + 1)) / 6;

    // Calculate the actual sum and sum of squares from the array
    long long actualSum = 0, actualSumSquares = 0;
    for (int i = 0; i < n; i++) {
        actualSum += arr[i];
        actualSumSquares += (long long)arr[i] * arr[i];
    }

    // Difference between actual and expected sums
    long long diffSum = actualSum - expectedSum; // X - Y
    long long diffSumSquares = actualSumSquares - expectedSumSquares; // X^2 -
Y^2

    // Using the relationship (X^2 - Y^2) = (X - Y)(X + Y)
    long long sumXY = diffSumSquares / diffSum; // X + Y

    // Solving for X and Y
    long long X = (diffSum + sumXY) / 2;
    long long Y = X - diffSum;

    return {(int)X, (int)Y}; // X is repeating, Y is missing
}
```

Hidden Test Case:
Test Case 1:

```
Unset
Input:
arr = [1, 3, 3, 4, 5, 6]
Output:
{3, 2}
Repeating: 3
Missing: 2
```

Test Case 2:

```
Unset
Input:
arr = [2, 2, 3, 4, 5]
Output:
{2, 1}
Repeating: 2
Missing: 1
```

Test Case 3:

```
Unset
arr = [6, 1, 2, 3, 4, 6]
Output:
{6, 5}
Repeating: 6
Missing: 5
```

Test Case 4:

```
Unset
arr = [1, 1]
Output:
{1, 2}
Repeating: 1
Missing: 2
```

Test Case 5:

```
Unset
arr = [1, 1, 2, 4]
Output:
{1, 3}
Repeating: 1
Missing: 3
```

## 20.Count Inversions in an Array

Given an array of integers containing N elements, the task is to count the number of inversions in the array. An inversion in an array is defined as a pair of indices `(i, j)` such that:

- `i < j`, and
- `arr[i] > arr[j]`.

In simpler terms, the inversion count indicates how far (or close) the array is from being sorted. The number of inversions indicates how many swaps are required to sort the array.

Input Format:

- An integer N denoting the size of the array.
- A single line containing N integers denoting the elements of the array.

Output Format:

- A single integer representing the number of inversions in the array.

Constraints:

- 1≤N≤10^5
- 1≤arr[i]≤10^9

Sample Input:

```
Unset
N = 5
arr = [2, 4, 1, 3, 5]
```

Sample Output:

```
Unset
3
```

Solutions using merge sort:

```
C/C++

// Helper function to merge two halves and count inversions
long long merge_and_count(vector<int>& arr, vector<int>& temp, int left, int
mid, int right) {
    long long inv_count = 0;
    int i = left;    // Starting index for left subarray
    int j = mid + 1; // Starting index for right subarray
    int k = left;    // Starting index to be sorted

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
            inv_count += (mid - i + 1); // Count inversions
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (i = left; i <= right; i++) arr[i] = temp[i];

    return inv_count;
}

// Recursive function to count inversions
long long merge_sort_and_count(vector<int>& arr, vector<int>& temp, int left,
int right) {
    long long inv_count = 0;
    if (left < right) {
        int mid = left + (right - left) / 2;
        inv_count += merge_sort_and_count(arr, temp, left, mid);
        inv_count += merge_sort_and_count(arr, temp, mid + 1, right);
        inv_count += merge_and_count(arr, temp, left, mid, right);
    }
    return inv_count;
}

// Function to count inversions
long long countInversions(vector<int>& arr, int n) {
    vector<int> temp(n);
    return merge_sort_and_count(arr, temp, 0, n - 1);
}
```

Hidden Test Case:
Test Case 1:

```
Unset
Input:
N = 4
```

```
arr = [4, 1, 3, 2]
Output:
5
```

Test Case 2:

```
Unset
Input:
N = 3
arr = [2, 3, 1]
Output:
2
```

Test Case 3:

```
Unset
N = 7
arr = [7, 6, 5, 4, 3, 2, 1]
Output:
21
```

Test Case 4:

```
Unset
Input:
N = 5
arr = [1, 2, 3, 4, 5]
Output:
0
```

Test Case 5:

```
Unset

Input:
N = 8
arr = [1, 3, 2, 8, 6, 4, 7, 5]
Output:
```

```
11
```

## 21.Count of Reverse Pairs

You are given an array `arr[]` consisting of integers. Your task is to find the total number of reverse pairs in the array. A reverse pair is a pair of indices `(i, j)` where:

```
- i < j
- arr[i] > 2 * arr[j]
```

### Input Format:

- An integer `n` which represents the size of the array `arr`.
- An array `arr[]` containing `n` integers.

### Output Format:

- Return a single integer, which is the count of reverse pairs in the array.

### Constraints:

- `1 <= n <= 100,000` (size of the array `arr`)
- `-10^9 <= arr[i] <= 10^9` (each element of the array lies within this range)

Sample Input:

```
Unset
Input:
N = 5
arr = [1, 3, 2, 3, 1]
```

Sample Output:

```
Unset
Output:
2
```

Solutions using Merge sort:

```
C/C++
#include <iostream>
#include <vector>
using namespace std;

// Function to merge two sorted halves of the array
void mergeHalves(vector<int>& arr, int start, int mid, int end) {
    vector<int> merged; // Temporary array for merged elements
    int leftIdx = start; // Starting index for left half
    int rightIdx = mid + 1; // Starting index for right half

    // Merge the two halves into the temporary array in sorted order
    while (leftIdx <= mid && rightIdx <= end) {
        if (arr[leftIdx] <= arr[rightIdx]) {
            merged.push_back(arr[leftIdx]);
            leftIdx++;
        } else {
            merged.push_back(arr[rightIdx]);
            rightIdx++;
        }
    }

    // Add remaining elements from the left half
    while (leftIdx <= mid) {
        merged.push_back(arr[leftIdx]);
        leftIdx++;
    }

    // Add remaining elements from the right half
    while (rightIdx <= end) {
        merged.push_back(arr[rightIdx]);
        rightIdx++;
    }

    // Copy the merged elements back to the original array
    for (int i = start; i <= end; i++) {
        arr[i] = merged[i - start];
    }
}

// Function to count reverse pairs in the current section of the array
int countReversePairs(vector<int>& arr, int start, int mid, int end) {
    int rightIdx = mid + 1;
    int count = 0;

    // Count pairs (arr[i], arr[j]) such that arr[i] > 2 * arr[j]
    for (int i = start; i <= mid; i++) {
        while (rightIdx <= end && arr[i] > 2 * arr[rightIdx]) {
            rightIdx++;
        }
        count += (rightIdx - (mid + 1));
    }
    return count;
}

// Merge sort function that returns the count of reverse pairs
int performMergeSort(vector<int>& arr, int start, int end) {
    int count = 0;
    if (start >= end) return count;

    int mid = (start + end) / 2;
    count += performMergeSort(arr, start, mid); // Sort left half
    count += performMergeSort(arr, mid + 1, end); // Sort right half
    count += countReversePairs(arr, start, mid, end); // Count reverse pairs
    mergeHalves(arr, start, mid, end); // Merge sorted halves
    return count;
}

// Function to initiate the process
int countReversePairsInArray(vector<int>& arr, int n) {
    return performMergeSort(arr, 0, n- 1);
}
```

Hidden Test Cases

Test Case 1:

```
Unset
Input:
N = 5
```

```
arr = [2, 4, 3, 5, 1]
Output:
3
```

Test Case 2:

```
Unset
Input:
N = 5
arr = [7, 14, 21, 42, 84]
Output:
0
```

Test Case 3:

```
Unset
Input:
N = 5
arr = [1, 0, 1, 0, 1]
Output:
3
```

Test Case 4:

```
Unset
Input:
N = 5
arr = [4, 3, 2, 1, 0]
Output:
10
```

Test Case 5:

```
Unset
Input:
N = 5
arr = [5, 3, 1, 4, 2]
Output:
```

```
6
```

## 22.Find element which appear >N/3 times in the array(Majority Element)

You are given an array consisting of N integers. Your task is to identify the elements that appear more than N/3 times in the array. If no element meets this condition, return an empty vector.

### Input Format

- The first line contains a single integer N, the size of the array.
- The second line contains N space-separated integers representing the elements of the array.

### Output Format

- Return a vector containing the elements that appear more than N/3 times. The elements in the output vector should be unique.

### Constraints

- 1≤N≤10^5
- −109≤arr[i]≤109

Sample Input:

```
Unset
Input: N = 6
arr = [11, 33, 33, 11, 33, 11]
```

Sample output:

```
Unset
11, 33
```

Solutions:

```
C/C++


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

    // Initialize candidates and counts for Boyer-Moore Voting Algorithm
    int candidate1 = INT_MIN, candidate2 = INT_MIN;
    int count1 = 0, count2 = 0;

    // First pass: Find potential candidates for majority element
    for (int num : arr) {
        if (num == candidate1) {
            count1++;
        } else if (num == candidate2) {
            count2++;
        } else if (count1 == 0) {
            candidate1 = num;
            count1 = 1;
        } else if (count2 == 0) {
            candidate2 = num;
            count2 = 1;
        } else {
            count1--;
            count2--;
        }
    }

    // Second pass: Verify if candidates occur more than N/3 times
    count1 = 0;
    count2 = 0;
    for (int num : arr) {
        if (num == candidate1) count1++;
        else if (num == candidate2) count2++;
    }

    vector<int> result;
    if (count1 > n / 3) result.push_back(candidate1);
    if (count2 > n / 3) result.push_back(candidate2);

    return result;
}
```

```
```

Hidden Test Cases:

Test Case 1:

```
Unset
Input:
N = 4
arr = [1, 1, 1, 2]
Output:
1
```

Test Case 2:

```
Unset
Input:
N = 5
arr = [5, 5, 5, 3, 3]
Output:
5
```

Test Case 3:

```
Unset
Input:
N = 8
arr = [7, 7, 8, 8, 8, 7, 2, 3]
Output:
7, 8
```

Test Case 4:

```
Unset
Input:
N = 6
arr = [4, 4, 4, 4, 4, 2]
Output:
4
```

Test Case 5:

```
Unset
Input:
N = 7
arr = [9, 10, 9, 10, 9, 10, 2]
Output:
9, 10
```

## 23.Fractional Knapsack

You are provided with N items, each having a specific weight and value. The goal is to maximize the total value of items you can carry in a knapsack that has a maximum capacity of weight W.

Important: You can either take an item completely or take a fraction of it (i.e., you can split the item).

Input Format:

- First line contains two integers, N and W — the number of items and the maximum weight capacity of the knapsack.
- Next N lines contain two integers each: the weight and value of each item.

Output Format:

- Output the maximum value that can be carried in the knapsack.

Constraints:

- 1≤N≤10^5
- 1≤W≤10^9
- 1≤weighti,valuei≤ 10^9

Sample Input:

```
Unset
3 50
10 60
20 100
30 120
```

Sample output:

```
Unset
240.00
```

Solutions using Greedy approach:

```
C/C++
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// A structure to represent items with their weights and values
struct Item {
    int weight;
    int value;
};

// Comparator function to sort items according to value/weight ratio
bool comparator(const Item &a, const Item &b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(int W, vector<Item> &items) {
    // Sort items by value-to-weight ratio in descending order
    sort(items.begin(), items.end(), comparator);

    double totalValue = 0.0;  // Store the total value of the knapsack
    int currentWeight = 0;    // Store the current weight in the knapsack
    // Iterate over all items
    for (const auto &item : items) {
        // If the item can be fully included
        if (currentWeight + item.weight <= W) {
            currentWeight += item.weight;
            totalValue += item.value;
        } else {
            // If the item can't be fully included, include the fraction that
fits
            int remainingWeight = W - currentWeight;
            totalValue += (double)item.value * ((double)remainingWeight /
item.weight);
            break;  // As the knapsack is now full, we break the loop
        }
    }

    return totalValue;
}

int main() {
    int N, W;  // Number of items and knapsack capacity
    cin >> N >> W;

    vector<Item> items(N);

    // Input each item's weight and value
    for (int i = 0; i < N; i++) {
        cin >> items[i].weight >> items[i].value;
    }

    // Calculate and output the maximum value we can achieve
    double maxValue = fractionalKnapsack(W, items);
    cout << fixed << setprecision(2) << maxValue << endl;

    return 0;
}
```

Hidden Test Case:

Test Case 1:

```
Unset

Input:
N = 5, W = 30
Items = [(5, 10), (10, 40), (20, 60), (5, 15), (5, 25)]

Output:
90.00
```

Test Case 2:

```
Unset

Input:
N = 4, W = 70
Items = [(15, 40), (10, 30), (5, 10), (50, 100)]

Output:
170.00
```

Test Case 3:

```
Unset

Input:
N = 6, W = 100
Items = [(40, 280), (10, 60), (20, 120), (30, 150), (50, 200), (5, 45)]

Output:
585.00
```

Test Case 4:

```
Unset

Input:
N = 3, W = 80
Items = [(30, 100), (40, 150), (50, 200)]
```

```
Output:
250.00
```

Test Case 5:

```Unset
Input:
N = 5, W = 35
Items = [(5, 30), (15, 50), (25, 70), (20, 80), (10, 40)]

Output:
110.00
```

## 24. Children and Candy

You have `n` children standing in a line, and each child has an associated rating given in an integer array `ratings`. You need to distribute candies to the children while adhering to the following rules:

1. Every child must receive at least one candy.
2. Children who have a higher rating than their immediate neighbors must receive more candies than those neighbors.

Your task is to determine the minimum number of candies required to satisfy these conditions.

### Input Format

- An integer `n` (1 ≤ n ≤ 2 * 10^4), representing the number of children.
- An array `ratings` of size `n`, where `ratings[i]` (1 ≤ ratings[i] ≤ 2 * 10^4) represents the rating of the `i-th` child.

### Output Format

- Return an integer representing the minimum number of candies required.

### Constraints

- Each child receives at least one candy.
- Children with a higher rating than their immediate neighbors receive more candies.

Sample Input:

```
Unset
Input:
n = 3
ratings = [1, 0, 2]
```

Sample Output:

```
Unset
Output:
5
```

Solutions using Greedy Approach:

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

int candy(vector<int>& ratings) {
    int n = ratings.size();
    if (n == 0) return 0;

    vector<int> candies(n, 1);

    // Forward pass: Ensure children with higher ratings than their left
neighbor get more candies
    for (int i = 1; i < n; i++) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        }
    }
    // Backward pass: Ensure children with higher ratings than their right
neighbor get more candies
    for (int i = n - 2; i >= 0; i--) {
        if (ratings[i] > ratings[i + 1]) {
            candies[i] = max(candies[i], candies[i + 1] + 1);
        }
    }

    // Sum up the candies
    return accumulate(candies.begin(), candies.end(), 0);
}

int main() {
    vector<int> ratings = {1, 0, 2};
    cout << candy(ratings) << endl; // Output: 5
    return 0;
}
```

Hidden Test Cases:

Test Case 1:

```
Unset
n = 5
ratings = [1, 2, 2]
Output:
7
```

Test Case 2:

```
Unset
Input:
n = 4
ratings = [1, 3, 2, 2]
Output:
7
```

Test Case 3:

```
Unset
Input:
n = 6
ratings = [1, 0, 2, 1, 2, 0]
Output:
10
```

Test Case 4:

```
Unset
Input:
n = 1
ratings = [1]
Output:
1
```

Test Case 5:

```
Unset
Input:
n = 8
ratings = [1, 2, 3, 4, 3, 2, 1, 2]
Output:
21
```

## 25. Jump Game

You are given an array of non-negative integers where each element represents the maximum number of steps you can jump forward from that position. Your goal is to determine if it is possible to reach the last index of the array starting from the first index.

### Input Format

- An integer `n`, representing the number of elements in the array.
- An array `nums` of size `n`, where `nums[i]` indicates the maximum jump length from the index `i`.

### Output Format

- Return a boolean value: `true` if you can reach the last index, otherwise return `false`.

## Constraints

- 0 ≤ nums[i] ≤ 10^5.
- 1 ≤ n ≤ 10^4

Sample Input:

```
Unset
Input:
nums = [2, 3, 1, 1, 4]
```

Sample Output:

```
Unset
True
```

Solution using Greedy Approach:

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

bool canJump(vector<int>& nums) {
    int n = nums.size();
    int farthest = 0;

    for (int i = 0; i < n; i++) {
        if (i > farthest) {
            return false; // If we reach a point that is not reachable
        }
        farthest = max(farthest, i + nums[i]); // Update the farthest reachable
index
        if (farthest >= n - 1) {
            return true; // If we can reach or exceed the last index
        }
    }
    return false; // If we finish the loop without reaching the last index
}
```

```
int main() {
    vector<int> nums = {2, 3, 1, 1, 4};
    cout << (canJump(nums) ? "True" : "False") << endl; // Output: True
    return 0;
}
```

Hidden Test Cases:
Test Case 1:

```
Unset
Input:
nums = [3, 2, 1, 0, 4]
Output:
False
```

Test Case 2:

```
Unset
Input:
nums = [0]
Output:
True
```

Test Case 3:

```
Unset
Input:
nums = [2, 0, 0, 0, 0]
Output:
True
```

Test Case 4:

```
Unset
Input:
```

```
nums = [4, 0, 0, 0, 0]
Output:
True
```

Test Case 5:

```
Unset
Input:
nums = [5, 4, 3, 2, 1, 0, 0]
Output:
True
```

## 26. Minimum Jumps to Reach the End

You are given a 0-indexed array `nums` of integers of length `n`. You start at `nums[0]`, and each element in `nums[i]` represents the maximum number of forward steps you can jump from that position.

The goal is to reach the last element (`nums[n-1]`) in the minimum number of jumps.

You need to return the minimum number of jumps required to reach `nums[n-1]`. It is guaranteed that it is always possible to reach the last index.

### Input Format:

- An integer array `nums` of length `n`, where each element represents the maximum number of forward steps you can jump from that position.

### Output Format:

- An integer representing the minimum number of jumps required to reach the last index.

### Constraints:

- `1 <= n <= 10^4`
- `0 <= nums[i] <= 1000`

- The test cases are designed such that it is always possible to reach the last index.

Sample Input:

```
Unset
Input:
N = 5
nums = [2, 3, 1, 1, 4]
```

Sample Output:

```
Unset
Output:
2
```

Solution using greedy approach:

```
C/C++
int minJumpsToEnd(vector<int>& nums) {
    int jumps = 1, n = nums.size();
    if (n == 1)  // If array size is 1, no jumps are needed
        return 0;

    int farthest = nums[0];  // Maximum reach at any point
    int stepsLeft = nums[0];  // Steps remaining within the current jump

    if (farthest >= n - 1)  // If we can reach the end with the first jump
        return jumps;

    for (int i = 1; i < n; i++) {
        farthest = max(farthest, i + nums[i]);  // Update the farthest we can
reach

        if (farthest >= n - 1) {  // If we can reach the end from the current
position
            jumps++;
            return jumps;
        }
        stepsLeft--;  // Decrease steps left within the current jump
        if (stepsLeft == 0) {  // If we need to jump again
            jumps++;
            stepsLeft = farthest - i;  // Reset steps to the farthest reachable
index
        }
    }

    return jumps;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
N = 6
nums = [1, 2, 0, 1, 1, 1]
Output:
4
```

Test Case 2:

```
Unset
Input:
N = 4
nums = [1, 3, 2, 1]
Output:
2
```

Test Case 3:

```
Unset
Input:
N = 6
```

```
nums = [2, 3, 1, 1, 1, 4]
Output:
3
```

Test Case 4:

```
Unset
Input:
N = 5
nums = [1, 1, 1, 1, 1]
Output:
4
```

Test Case 5:

```
Unset
Input:
N = 5
nums = [1, 1, 1, 1, 2]
Output:
4
```

## 27.Maximize Meetings in a Single Room

You are given `n` meetings, each defined by a start time and an end time. The start time of the i-th meeting is `start[i]`, and the end time is `end[i]`. Only one meeting can be held at a time in a single meeting room. Your task is to determine the maximum number of meetings that can be scheduled in the room such that no two meetings overlap. It is also required that the start time of any chosen meeting cannot coincide with the end time of another meeting.

Input Format:

- An integer `n` representing the number of meetings.
- Two arrays, `start` and `end`, each of size `n`, where `start[i]` and `end[i]` represent the start and end times of the i-th meeting.

Output Format:

- Return a single integer representing the maximum number of meetings that can be scheduled without any overlap.

Constraints:

- 1 ≤ n ≤ 10^5
- 0 ≤ `start[i]`, `end[i]` ≤ 10^9

Sample Input:

```
Unset
n = 3
start = [2, 2, 2]
end = [3, 3, 3]
```

Sample Output:

```
Unset
1
```

Solutions using greedy approach:

```
C/C++


// Function to return the maximum number of non-overlapping meetings
int maxMeetings(vector<int>& start, vector<int>& end, int n) {
    vector<pair<int, int>> meetings(n);

    // Store the start and end times as pairs in a vector
    for (int i = 0; i < n; i++) {
        meetings[i] = {end[i], start[i]};  // Store end time first to sort by
it
    }

    // Sort meetings by end time
    sort(meetings.begin(), meetings.end());
    int count = 1;  // We can always select the first meeting
    int last_end_time = meetings[0].first;  // End time of the first selected
meeting

    // Iterate over remaining meetings
    for (int i = 1; i < n; i++) {
        if (meetings[i].second > last_end_time) {  // Check if the next meeting
can be selected
            count++;
            last_end_time = meetings[i].first;  // Update the end time of the
last selected meeting
        }
    }

    return count;  // Return the maximum number of meetings
}
```

Hidden Test Cases:

Test Case 1:

```
Unset
n = 6
start = [1, 3, 0, 5, 8, 5]
end = [2, 4, 6, 7, 9, 9]
Output:
4
```

Test Case 2:

```
Unset
Input:
n = 3
start = [1, 2, 3]
end = [2, 3, 4]
```

```
Output:
3
```

Test Case 3:

```
Unset
n = 4
start = [1, 2, 3, 4]
end = [10, 3, 5, 6]
Output:
2
```

Test Case 4:

```
Unset
Input:
n = 5
start = [0, 10, 15, 18, 20]
end = [10, 15, 20, 25, 30]
Output:
5
```

Test Case 5:

```
Unset
Input:
n = 1
start = [5]
end = [10]
Output:
1
```

## 28. Lemonade Change

You are running a lemonade stand where each lemonade costs $5. Customers come in order, and each customer pays with a $5, $10, or $20 bill. You start with no money, so you must provide change for each customer from the money you have collected so far. The goal is to determine if you can give the correct change to each customer in the queue.

You need to return `true` if you can provide change for every customer; otherwise, return `false`.

## Input Format:

- An integer array `bills`, where `bills[i]` represents the bill that the i-th customer pays.

## Output Format:

- Return a boolean value `true` if it's possible to provide change to every customer, otherwise return `false`.

## Constraints:

- 1 ≤ `bills.length` ≤ 10^5
- `bills[i]` is either 5, 10, or 20.

Sample Input:

```
Unset
Input:
bills = [5, 5, 5, 10, 5, 10, 10, 20, 20]
```

Sample Output:

```
Unset
false
```

Solutions:

```
C/C++
#include <vector>
using namespace std;
```

```
bool lemonadeChange(vector<int>& bills) {
    int countFive = 0, countTen = 0;  // Track the number of $5 and $10 bills

    for (int bill : bills) {
        if (bill == 5) {
            countFive++;  // Increase count of $5 bills
        }
        else if (bill == 10) {
            if (countFive == 0) return false;  // Need one $5 bill for change
            countFive--;  // Use one $5 bill for change
            countTen++;  // Collect the $10 bill
        }
        else {  // bill == 20
            if (countTen > 0 && countFive > 0) {  // Prefer giving one $10 and
one $5 as change
                countTen--;
                countFive--;
            }
            else if (countFive >= 3) {  // Else, give three $5 bills as change
                countFive -= 3;
            }
            else {
                return false;  // Cannot give correct change
            }
        }
    }

    return true;  // Successfully gave correct change to all customers
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
bills = [5, 5, 5, 10, 20]
Output:
true
```

Test Case 2:

```
Unset
Input:
bills = [5, 5, 10, 10, 20]
Output:
false
```

Test Case 3:

```
Unset
Input:
bills = [5, 5, 5, 5, 10, 5, 10, 20]
Output:
true
```

Test Case 4:

```
Unset
Input:
bills = [5, 10, 5, 20, 5, 10, 5, 5, 20]
Output:
true
```

Test Case 5:

```
Unset
Input:
bills = [5, 10, 20]
Output:
false
```

## 29. Job Scheduling to Maximize Profit

Given a list of n jobs where each job has a deadline and a profit associated with it, we need to find a way to schedule jobs such that the number of jobs completed is maximized, and the total profit is also maximized.

Each job takes exactly 1 unit of time, and a job can only be completed if it is finished by its deadline. We can only work on one job at a time, so we need to carefully select the jobs in order to maximize the total profit.

## Input Format:

- `n`: The number of jobs.
- `jobs[]`: A list of `n` jobs, where each job contains three values:
    - `jobId`: Unique identifier of the job.
    - `deadline`: The time by which the job should be completed.
    - `profit`: The profit gained by completing the job on or before its deadline.

## Output Format:

- The first value should be the maximum number of jobs that can be done.
- The second value should be the maximum profit earned by completing the jobs.

## Constraints:

- `1 <= n <= 1000`
- Each job takes exactly 1 unit of time.
- `1 <= Deadline[i] <= 1000`
- `1 <= Profit[i] <= 10^4`

Sample Input:

```
Unset
n = 5
jobs = {{1, 2, 100}, {2, 1, 19}, {3, 2, 27}, {4, 1, 25}, {5, 3, 15}}
```

Sample Output:

```
Unset
3
130
```

Solutions using greedy :

```
C/C++
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Structure to represent a job
struct Job {
    int id;        // Job ID
    int deadline;  // Deadline of the job
    int profit;    // Profit associated with the job
};

// Comparator function to sort jobs according to profit in descending order
bool compare(Job a, Job b) {
    return a.profit > b.profit;
}

pair<int, int> jobScheduling(vector<Job>& jobs, int n) {
    // Sort the jobs by descending profit
    sort(jobs.begin(), jobs.end(), compare);

    int maxDeadline = 0;
    for (auto job : jobs) {
        maxDeadline = max(maxDeadline, job.deadline);  // Find the maximum
deadline
    }

    // Create an array to track which time slots are occupied
    vector<int> slots(maxDeadline + 1, -1);  // -1 means slot is free
    int jobsDone = 0;  // Number of jobs completed
    int totalProfit = 0;  // Total profit earned

    // Process each job in the sorted order
    for (int i = 0; i < n; i++) {
        // Try to find a free slot for this job (starting from its deadline)
        for (int j = min(maxDeadline, jobs[i].deadline); j > 0; j--) {
            if (slots[j] == -1) {  // If the slot is free
                slots[j] = jobs[i].id;  // Assign this job to the slot
                jobsDone++;  // Increment the count of jobs done
                totalProfit += jobs[i].profit;  // Add profit of the job
                break;  // Job scheduled, break out of the loop
            }
        }
    }

    return {jobsDone, totalProfit};  // Return the result
}

int main() {
    int n = 5;
    vector<Job> jobs = {{1, 2, 100}, {2, 1, 19}, {3, 2, 27}, {4, 1, 25}, {5, 3,
15}};

    pair<int, int> result = jobScheduling(jobs, n);
    cout << "Number of jobs done: " << result.first << endl;
    cout << "Maximum profit: " << result.second << endl;

    return 0;
}
```

Hidden Test Case:

Test Case 1:

```
Unset

Input:
n = 4
jobs = {{1, 4, 50}, {2, 1, 30}, {3, 2, 20}, {4, 3, 60}}
```

```
Output:
3
130
```

Test Case 2:

```Unset
Input:
n = 6
jobs = {{1, 1, 20}, {2, 2, 30}, {3, 1, 25}, {4, 3, 15}, {5, 2, 10}, {6, 1, 35}}

Output:
3
85
```

Test Case 3:

```Unset
Input:
n = 5
jobs = {{1, 3, 80}, {2, 2, 40}, {3, 1, 70}, {4, 3, 50}, {5, 1,
60}}

Output:
3
180
```

Test Case 4:

```Unset
Input:
n = 4
jobs = {{1, 2, 50}, {2, 2, 10}, {3, 1, 30}, {4, 3, 20}}
```

```
Output:
3
100
```

Test Case 5:

```
Unset

Input:
n = 6
jobs = {{1, 3, 55}, {2, 1, 25}, {3, 2, 75}, {4, 2, 20}, {5, 3, 45}, {6, 4, 35}}

Output:
4
210
```

## 30.Average Waiting Time using Shortest Job First (SJF)

You are given a list of processes, each with a specific burst time, representing the amount of time required for execution. Your task is to calculate the average waiting time of all the processes when they are scheduled using the **Shortest Job First (SJF)** scheduling algorithm.

In **Shortest Job First (SJF)**, the process with the shortest burst time is executed first. If two processes have the same burst time, any one of them can be chosen.

You need to determine the average waiting time for all the processes and return the largest integer less than or equal to this average waiting time.

**Assumptions:**

- All processes are available for execution at time 0.

### Input:

- An integer `n` denoting the number of processes.
- A list `bt` of size `n` where `bt[i]` represents the burst time of the `i-th` process.

## Output:

- Return the integer part of the average waiting time, rounded down.

## Constraints:

- `1 ≤ n ≤ 100`
- `1 ≤ bt[i] ≤ 1000`

Sample Input:

```
Unset
Input:
n = 5
bt = [6, 8, 7, 3, 4]
```

Sample Output:

```
Unset
7
```

Solutions:

```
C/C++
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int averageWaitingTime(vector<int>& bt) {
    int n = bt.size();

    // Sort the burst times for SJF scheduling
    sort(bt.begin(), bt.end());

    vector<int> waitingTime(n, 0);
    waitingTime[0] = 0;  // First process has no waiting time

    int totalWaitingTime = 0;
    // Calculate waiting time for each process
    for (int i = 1; i < n; i++) {
        waitingTime[i] = waitingTime[i - 1] + bt[i - 1];
        totalWaitingTime += waitingTime[i];
    }

    // Calculate average waiting time
    int avgWaitingTime = totalWaitingTime / n;

    return avgWaitingTime;
}

int main() {
    vector<int> burstTimes = {10, 5, 8, 7};
    cout << averageWaitingTime(burstTimes) << endl;
    return 0;
}
```

Hidden Test Case:
Test Case 1:

```
Unset

Input:
n = 3
bt = [2, 3, 4]

Output:
2
```

Test Case 2:

```
Unset

Input:
n = 6
bt = [1, 2, 3, 4, 5, 6]
```

```
Output:
7
```

Test Case 3:

```Unset
Input:
n = 4
bt = [5, 1, 3, 9]

Output:
4
```

Test Case 4:

```Unset
Input:
n = 3
bt = [5, 9, 3]

Output:
3
```

Test Case 5:

```Unset
Input:
n = 4
bt = [10, 20, 30, 40]

Output:
45
```

## 31. Preorder of Binary Tree

You are given the root node of a binary tree. Write a recursive function that performs a **preorder traversal** of the binary tree and returns an array containing the node values in the order they are visited.

## Input:

- A binary tree's root node.

## Output:

- **Preorder Traversal**: An array with the values of nodes in preorder.

## Constraints:

- The number of nodes in the binary tree is between 1 and 1000.
- Each node's value is unique, ranging from [-1000, 1000].
- The binary tree is well-formed and rooted.

Sample Input:

```
Unset
        1
       / \
      2   3
     / \
    4   5
```

Sample Output:

```
Unset
[1, 2, 4, 5, 3]
```

Solutions using Recursion:

```
C/C++

// Definition of a tree node
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Function to perform preorder traversal
void preorderTraversal(TreeNode* root, vector<int>& result) {
    if (root == NULL) return;          // Base case: if node is null, return
    result.push_back(root->val);        // Visit the root
    preorderTraversal(root->left, result);  // Traverse the left subtree
    preorderTraversal(root->right, result); // Traverse the right subtree
}

// Helper function to initiate preorder traversal
vector<int> preorder(TreeNode* root) {
    vector<int> result;
    preorderTraversal(root, result);
    return result;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
      10
        \
         20
        /
       15

Output:
[10, 20, 15]
```

Test Case 2:

```
Unset
Input:
      1
     / \
    2   3

Output: [1, 2, 3]
```

Test Case 3:

```
Unset
Input:
      5
     / \
    2   8
   / \
  1   3

Output: [5, 2, 1, 3, 8]
```

Test Case 4:

```
Unset
Input:
      7
     / \
    6   10
   /    /
  5    9

Output: [7, 6, 5, 10, 9]
```

Test Case 5:

```
Unset
Input:
      4
       \
        6
       /
      5

Output: [4, 6, 5]
```

## 32. Postorder of Binary Tree

You are given the root node of a binary tree. Write a recursive function that performs a **postorder traversal** of the binary tree and returns an array containing the node values in the order they are visited.

### Input:

- A binary tree's root node.

### Output:

- **Postorder Traversal**: An array with the values of nodes in postorder.

### Constraints:

- The number of nodes in the binary tree is between 1 and 1000.
- Each node’s value is unique, ranging from [-1000, 1000].
- The binary tree is well-formed and rooted.

Sample Input:

```
Unset
        1
       / \
      2   3
     / \
    4   5
```

Sample Output:

```
Unset
[4, 5, 2, 3, 1]
```

Solution using Recursion:

```cpp
C/C++

// Definition of a tree node
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Function to perform postorder traversal
void postorderTraversal(TreeNode* root, vector<int>& result) {
    if (root == NULL) return;           // Base case: if node is null, return
    postorderTraversal(root->left, result);  // Traverse the left subtree
    postorderTraversal(root->right, result); // Traverse the right subtree
    result.push_back(root->val);         // Visit the root after subtrees
}

// Helper function to initiate postorder traversal
vector<int> postorder(TreeNode* root) {
    vector<int> result;
    postorderTraversal(root, result);
    return result;
}
```

Hidden Test Cases:
Test Case 1:

```
Unset
Input:
      10
```

```
        \
         20
        /
       15

Output:
[15, 20, 10]
```

Test Case 2:

```
Unset
Input:
      1
     / \
    2   3

Output: [2, 3, 1]
```

Test Case 3:

```
Unset
Input:
      5
     / \
    2   8
   / \
  1   3

Output: [1, 3, 2, 8, 5]
```

Test Case 4:

```
Unset
Input:
      7
     / \
    6   10
   /    /
  5    9

Output: [5, 6, 9, 10, 7]
```

Test Case 5:

```
Unset
Input:
      4
       \
        6
       /
      5

Output: [5, 6, 4]
```

## 33. Inorder of Binary Tree:

You are given the root node of a binary tree. Write a recursive function that performs a **inorder traversal** of the binary tree and returns an array containing the node values in the order they are visited.

### Input:

- A binary tree's root node.

### Output:

- **Inorder Traversal**: An array with the values of nodes in inorder.

### Constraints:

- The number of nodes in the binary tree is between 1 and 1000.
- Each node's value is unique, ranging from [-1000, 1000].

- The binary tree is well-formed and rooted.

Sample Input:

```
Unset
        1
       / \
      2   3
     / \
    4   5
```

Sample output:

```
Unset
[4, 2, 5, 1, 3]
```

Solution using Recursion:

```
C/C++

// Definition of a tree node
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Function to perform inorder traversal
void inorderTraversal(TreeNode* root, vector<int>& result) {
    if (root == NULL) return;           // Base case: if node is null, return
    inorderTraversal(root->left, result);  // Traverse the left subtree
    result.push_back(root->val);         // Visit the root after left subtree
    inorderTraversal(root->right, result); // Traverse the right subtree
}

// Helper function to initiate inorder traversal
vector<int> inorder(TreeNode* root) {
    vector<int> result;
    inorderTraversal(root, result);
    return result;
}
```

Hidden Test Case:
Test Case 1:

```Unset
Input:
      10
        \
         20
        /
       15

Output:
[10, 15, 20]
```

Test Case 2:

```Unset
Input:
      1
     / \
    2   3

Output: [2, 1, 3]
```

Test Case 3:

```
Unset
Input:
      5
     / \
    2   8
   / \
  1   3

Output: [1, 2, 3, 5, 8]
```

Test Case 4:

```
Unset
Input:
      7
     / \
    6   10
   /    /
  5    9

Output: [5, 6, 7, 9, 10]
```

Test Case 5:

```
Unset
Input:
      4
       \
        6
       /
      5

Output: [4, 5, 6]
```

## 34. Level Order Traversal of a Binary Tree

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

**Input Format:**

- The input consists of a binary tree and the root is given.

**Output Format:**

- Return an array of arrays, where each inner array represents the values of nodes at that level.

**Constraints:**

- The number of nodes in the tree will be in the range [0, 10^4].
- The value of each node is an integer within the range of [-10^5, 10^5].

Sample Input:

```
Unset
Input:
      1
     / \
    2   3
```

Sample Output:

```
Unset

Output: [[1], [2, 3]]
```

Solution using BFS and Queue:

```
C/C++

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Function for level order traversal
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* currentNode = q.front();
            q.pop();
            currentLevel.push_back(currentNode->val);

            if (currentNode->left) q.push(currentNode->left);
            if (currentNode->right) q.push(currentNode->right);
        }

        result.push_back(currentLevel);
    }

    return result;
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
      5
     / \
    2   8
   / \
  1   3

Output: [[5], [2, 8], [1, 3]]
```

Test Case 2:

```
Unset
Input:
      7
     / \
    6   10
   /    /
  5    9

Output: [[7], [6, 10], [5, 9]]
```

Test Case 3:

```
Unset
Input:
      4
       \
        6
       /
      5

Output: [[4], [6], [5]]
```

Test Case 4:

```
Unset
Input:
       10
      /  \
     8    15
    /    /  \
   7    12  20

Output: [[10], [8, 15], [7, 12, 20]]
```

Test Case 5:

```
Unset
Input:
        3
       / \
      9  20
         / \
        15  7

Output: [[3], [9, 20], [15, 7]]
```

## 35.Height of Binary Tree

Given the root of a Binary Tree, write a code that returns the height of the tree. The height of the tree is defined as the number of nodes on the longest path from the root to a leaf node.

### Input Format:

- The input consists of a binary tree represented by its root node.

### Output Format:

- Return an integer representing the height of the tree.

### Constraints:

- The number of nodes in the tree is in the range `[0, 1000]`.
- The value of each node is a non-negative integer.

Sample Input:

```
Unset
Input:
        1
       / \
      2   3
     /
    4
```

Sample Output:

```
Unset
Output: 3
```

Solution Using Recursion:

```cpp
C/C++
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

int heightOfBinaryTree(TreeNode* root) {
    if (root == nullptr) return 0; // Base case: height of empty tree is 0
    return 1 + max(heightOfBinaryTree(root->left),
heightOfBinaryTree(root->right)); // 1 for the root + max height of left or
right subtree
}
```

Hidden Test Cases:

Test Case 1:

```
Unset
```

```
Input:
        5
       / \
      6   7
           \
            8

Output: 4
```

Test Case 2:

```
Unset
Input:
        10
       /
      20
     / \
    30  40

Output: 3
```

Test Case 3:

```
Unset
Input:
        9
       / \
      6   12
     /      \
    3        15

Output: 3
```

Test Case 4:

```
Unset
Input:
        2
       / \
      1   3

Output: 2
```

Test Case 5:

```
Unset
Input:
        6
       / \
      5   7
     /
    4

Output: 3
```

## 36.Balanced Binary Tree

Given a Binary Tree, write a code that returns true if it is a balanced Binary Tree; otherwise, return false. A Binary Tree is considered balanced if, for all nodes in the tree, the difference between the heights of the left and right subtrees is not more than 1.

### Input Format:

- The input consists of a binary tree represented by its root node.

### Output Format:

- Return a boolean value: `true` if the tree is balanced, otherwise `false`.

### Constraints:

- The number of nodes in the tree is in the range `[0, 1000]`.
- The value of each node is a non-negative integer.

Sample Input:

```
Unset
Input:
        1
       / \
      2   3
     /
    4
```

Sample Output:

```
Unset
Output: false
```

Solutions using Recursion:

```
C/C++
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Helper function to check the height and balance simultaneously
int checkBalance(TreeNode* root) {
    if (root == nullptr) return 0; // Base case: height of empty tree is 0

    int leftHeight = checkBalance(root->left); // Check left subtree
    if (leftHeight == -1) return -1; // Left subtree is not balanced

    int rightHeight = checkBalance(root->right); // Check right subtree
    if (rightHeight == -1) return -1; // Right subtree is not balanced

    // Check if current node is balanced
    if (abs(leftHeight - rightHeight) > 1) return -1;

    // Return height of the current subtree
    return 1 + max(leftHeight, rightHeight);
}

bool isBalanced(TreeNode* root) {
    return checkBalance(root) != -1; // If the return value is -1, the tree is
not balanced
}
```

Hidden Test Case:

Test Case 1:

```
Unset
Input:
        5
       / \
      6   7
           \
            8

Output: false
```

Test Case 2:

```
Unset
Input:
        10
       / \
      20  30
     / \
    40  50

Output: true
```

Test Case 3:

```
Unset
Input:
        9
       / \
      6   12
     / \
    3   8

Output: true
```

Test Case 4:

```
Unset
Input:
        2
       / \
      1   3

Output: true
```

Test Cse 5:

```
Unset
Input:
        6
       / \
      5   7
     /
    4

Output: false
```

## 37.Diameter of Binary Tree

Given the root of a binary tree, return the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root. The length of a path between two nodes is represented by the number of edges between them.

## Input Format:

- The input consists of a binary tree represented by its root node.

## Output Format:

- Return an integer representing the length of the diameter of the binary tree.

## Constraints:

- The number of nodes in the tree is in the range `[0, 1000]`.
- The value of each node is a non-negative integer.

Sample Input:

```
Unset
Input:
        1
       / \
      2   3
     / \
    4   5
```

Sample Output:

```
Unset
Output: 3
```

Solutions:

```
C/C++

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

```
    int diameter = 0; // Initialize diameter

    int height(TreeNode* node) {
        if (node == nullptr) return 0; // Base case: height of empty tree is 0

        int leftHeight = height(node->left); // Height of left subtree
        int rightHeight = height(node->right); // Height of right subtree

        // Update the diameter if the path through the current node is larger
        diameter = max(diameter, leftHeight + rightHeight);

        // Return the height of the tree rooted at the current node
        return 1 + max(leftHeight, rightHeight);
    }

    int diameterOfBinaryTree(TreeNode* root) {
        height(root); // Calculate height and update diameter
        return diameter; // Return the maximum diameter found
    }
```

Hidden Test Case

Test Case 1:

```
Unset
Input:
        5
       / \
      6   7
           \
            8

Output: 3
```

Test Case 2:

```
Unset
Input:
        10
       / \
      20  30
     / \
    40  50

Output: 3
```

Test Case 3:

```
Unset
Input:
        9
       / \
      6   12
     / \
    3   8

Output: 3
```

Test Case 4:

```
Unset
Input:
        2
       / \
      1   3

Output: 2
```

Test Case 5:

```
Unset
```

```
Input:
        1
       /
      2
     /
    3
   /
  4

Output: 3
```

## 38. Lowest Common Ancestor of a Binary Tree [**Flipkart, Accolite, Amazon, Microsoft, OYO Rooms, Snapdeal, MakeMyTrip, Payu, Google, Times Internet, Cisco, PayPal**
**Expedia**
**Twitter**
**American Express**
]

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree. The LCA is defined as the deepest node that is an ancestor of both nodes. A node is considered an ancestor of itself.

### Input Format:

- The input consists of the root node of the binary tree and two node values `p` and `q` for which the LCA needs to be found.

### Output Format:

- Return the node that represents the lowest common ancestor of the two nodes.

### Constraints:

- The number of nodes in the tree is in the range `[1, 1000]`.
- The values of nodes are unique.

Sample Input:

```
Unset

Input:
        3
       / \
      5   1
     / \   \
    6   2   0
       / \
      7   4

p = 5, q = 4
```

Sample Output:

```
Unset
Output: 5
```

Solution:

```
C/C++

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == nullptr || root == p || root == q) {
            return root; // Base case: return root if it is null or equal to p
or q
        }

        TreeNode* left = lowestCommonAncestor(root->left, p, q); // Search in
left subtree
        TreeNode* right = lowestCommonAncestor(root->right, p, q); // Search in
right subtree

        if (left && right) {
            return root; // If both left and right are not null, root is LCA
        }
        return left ? left : right; // Return the non-null value
    }
```