Day 47/180 Solving Hard Problems

1:Finding Missing and Repeating Elements:

```
vector<int> findTwoElement(vector<int> arr, int n) {
    long long sum = 0, sqSum = 0;

    // Calculate the sum of numbers and the sum of squares of numbers
    for (long long i = 1; i <= n; i++) {
        long long cur = arr[i - 1];
        sum += i - cur;        // Calculate the difference between expected and
actual sum
        sqSum += i * i - cur * cur;  // Calculate the difference between the expected
and actual sum of squares
    }

    // Calculate the sum of both elements and the sum of the elements' squares
    long long sumb = sqSum / sum;

    // Calculate the two elements
    long long a = (sum + sumb) / 2;
    long long b = (sumb - sum) / 2;

    // Ensure that 'a' is the smaller element and 'b' is the larger element
    if (sum > 0) {
        if (a > b) swap(a, b);
    } else {
        if (a < b) swap(a, b);
    }

    return {a, b}; // Return the found elements
}
Time complexity - O(N), Space Complexity - O(1)
```

2: Count Frequency of Element:

Solution 1:

```
void frequencyCount(vector<int>& arr, int N, int P) {
    // Create an unordered map to store the frequency of each number
    unordered_map<int, int> ump;

    // Step 1: Calculate the frequency of each number in the array
    for (int i = 0; i < N; i++) {
        ump[arr[i]]++;
    }

    // Clear the original 'arr' to prepare for updating with frequency information
    arr.clear();

    // Step 2: Rebuild the 'arr' with frequency information
    for (int i = 1; i <= N; i++) {
        arr.push_back(ump[i]);
    }
}

Time Complexity - O(N), Space Complexity - O(N)
```

Solution 2: Space Complexity - O(1)

```
void frequencyCount(vector<int>& arr, int n, int p) {
    // Step 1: Normalize elements that are greater than 'n'
    for (int i = 0; i < n; i++) {
        if (arr[i] > n) {
            arr[i] = 0; // Normalize the element to be within the range [0, n]
        }
    }

    // Step 2: Update the frequency information using element values
    for (int i = 0; i < n; i++) {
        // Calculate 'num' as the updated index for the element based on its
value
        int num = arr[i] % (p + 1) - 1;

        if (num >= 0) {
            // Increment the frequency of the element at index 'num'
            arr[num] += (p + 1);
        }
    }

    // Step 3: Revert the frequency information to the original element values
    for (int i = 0; i < n; i++) {
        // Divide the element value by (p + 1) to obtain the original element
value
        arr[i] = arr[i] / (p + 1);
    }
}
Time Complexity - O(N), Space Complexity - O(1)
```

3: Majority ELement: Solution

```
int majorityElement(vector<int>& nums) {
    int ans = -1; // Initialize the answer to an invalid value (it will be
updated)
    int vote = 0; // Initialize the vote count to zero

    // Iterate through the elements of the 'nums' array
    for (auto x : nums) {
        if (vote == 0) {
            // If the vote count is zero, assign the current element as the 'ans'
            ans = x;
            vote++;
        } else {
            // If the vote count is not zero, check if the current element is the
same as 'ans'
            if (x == ans) {
                // If it is the same, increment the vote count
                vote++;
            } else {
                // If it is different, decrement the vote count
                vote--;
            }
        }
    }
    return ans; // Return the majority element (the one with the most votes)
}
```

```
Explanation of Correctness:
The algorithm works on the basis of the assumption that the majority element occurs
more than n/2 times in the array. This assumption guarantees that even if the count
is reset to 0 by other elements, the majority element will eventually regain the
lead.

Let's consider two cases:

If the majority element has more than n/2 occurrences:

The algorithm will ensure that the count remains positive for the majority element
throughout the traversal, guaranteeing that it will be selected as the final
candidate.
If the majority element has exactly n/2 occurrences:

In this case, there will be an equal number of occurrences for the majority element
and the remaining elements combined.
However, the majority element will still be selected as the final candidate because
it will always have a lead over any other element.
In both cases, the algorithm will correctly identify the majority element.

The time complexity of the Moore's Voting Algorithm is O(n) since it traverses the
array once.
```

4: Smallest missing positive integer (Use Modulous)

```
int firstMissingPositive(vector<int>& A) {
    int n = A.size(); // Get the number of elements in the array

    // Step 1: Rearrange the array to place each positive integer in its correct
position
    for (int i = 0; i < n; ++i) {
        while (A[i] > 0 && A[i] <= n && A[A[i] - 1] != A[i]) {
            // While the current element is a positive integer within the range [1,
n]
            // and it's not in its correct position, swap it with the element at its
correct position.
            swap(A[i], A[A[i] - 1]);
        }
    }

    // Step 2: Find the first missing positive integer
    for (int i = 0; i < n; ++i) {
        if (A[i] != i + 1) {
            // If the current element doesn't match its expected value (i + 1),
            // return the first missing positive integer, which is (i + 1).
            return i + 1;
        }
    }

    // If all positive integers from 1 to n are present, return the next positive
integer, which is (n + 1).
    return n + 1;
}
Time Complexity - O(N), Space Complexity - O(1)
```