Day 92/180 Recursion permutation 2

1:Permutation 2: Solution:-

```cpp
class Solution {
public:
    // Recursive function to generate unique permutations
    void get(int i, vector<int> nums, vector<vector<int>>& ans) {
        // If we reach the end of the array, add the current permutation to the
result
        if (i == nums.size() - 1) {
            ans.push_back(nums);
            return;
        }

        // Iterate over the array starting from the current index
        for (int j = i; j < nums.size(); j++) {
            // Skip duplicates to avoid redundant permutations
            if (i != j && nums[i] == nums[j]) continue;

            // Swap elements to generate permutations
            swap(nums[i], nums[j]);

            // Recursively generate permutations for the next index
            get(i + 1, nums, ans);
        }
    }
    // Function to generate unique permutations for a given vector of numbers
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        // Sort the input vector to handle duplicates efficiently
        sort(nums.begin(), nums.end());

        // Vector to store the generated permutations
        vector<vector<int>> ans;
        // Start the recursive permutation generation process
        get(0, nums, ans);
        // Return the final result
        return ans;
    }
};
```

2:Ways to Sum N:

```
class Solution {
public:
    // Variable to store the modulo value
    int mod = 1e9 + 7;
    // Recursive function to count the total number of ways to sum up to 'N'
    int solve(int m, int N, int arr[], vector<int>& dp) {
        // Base case: If the sum is 0, there is one way (empty subset)
        if (N == 0) return 1;
        // Base case: If the sum is negative, no way to form subsets
        if (N < 0) return 0;

        // Memoization: If the result for the current sum is already calculated,
return it
        if (dp[N] != -1) return dp[N];

        // Initialize the result for the current sum
        dp[N] = 0;
        // Iterate through the array to consider each element for the subset
        for (int i = 0; i < m; i++) {
            // Update the result by considering subsets with the current element
            dp[N] = (dp[N] % mod + solve(m, N - arr[i], arr, dp) % mod) % mod;
        }

        // Return the result for the current sum, taking modulo into account
        return dp[N] % mod;
    }
    // Function to count the total number of ways to sum up to 'N'
    int countWays(int arr[], int m, int N) {
        // Vector to store intermediate results using memoization
        vector<int> dp(N + 1, -1);

        // Start the recursive process to count ways
        return solve(m, N, arr, dp);
    }
};
```

3: Combination 2

```
class Solution {
public:
    // Recursive function to find combinations that sum up to the target
    void solve(int i, int target, vector<int>& a, vector<int>& cur,
vector<vector<int>>& ans) {
        // Base case: If the target becomes negative, stop exploring this path
        if (target < 0) return;

        // Base case: If the target becomes zero, a valid combination is found
        if (target == 0) {
            ans.push_back(cur);
            return;
        }

        // Iterate through the array to consider each element for the combination
        for (int j = i; j < a.size(); j++) {
            // Skip duplicates to avoid redundant combinations
            if (i != j && a[j - 1] == a[j]) continue;

            // Add the current element to the combination
            cur.push_back(a[j]);

            // Recursively explore combinations with the current element
            solve(j + 1, target - a[j], a, cur, ans);

            // Backtrack: Remove the last added element for the next iteration
            cur.pop_back();
        }
    }

    // Main function to find combination sums
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        // Sort the candidates to handle duplicates and ease skip conditions
        sort(candidates.begin(), candidates.end());

        // Vector to store the final combinations
```

```
        vector<vector<int>> ans;

        // Vector to store the current combination during recursive calls
        vector<int> cur;

        // Start the recursive process to find combinations
        solve(0, target, candidates, cur, ans);

        // Return the final combinations
        return ans;
    }
};
```

4:Elimination game

```
class Solution {
public:
    // Function to find the last remaining element in the sequence
    int lastRemaining(int n) {
        // Base case: If there is only one element, it is the last remaining
        if (n == 1) {
            return 1;
        }

        //firstly n==1 the output is 1
        //then using lifo 9 is number [2*(5.5-4.5)]=2
        //for 8 :- [2*(5-4)]=2
        //for 2 :- [2*(2-1)]=2
        //for 3 :- [2*(2.5-1.5)]=2
        //for 4 :- [2*(3-2)]=2
        return 2 * (1 + n / 2 - lastRemaining(n / 2));
    }
};
```