Day 89/180

Permutations of Arrays 

---

1. Permutations

```
class Solution {
public:
    void Permutations(int ind, vector<int> &nums, vector<vector<int>>
&ans){
        if(ind == nums.size()){
            ans.push_back(nums);
            return;
        }
        for(int i=ind ; i<nums.size() ; i++){
            swap(nums[ind],nums[i]);
            Permutations(ind+1, nums, ans);
            swap(nums[ind],nums[i]);
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> ans;
        Permutations(0, nums, ans);
        return ans;
    }
};
```

Code Explanation and Complexity

- The permutation function is defined to generate permutations, which takes as arguments a reference to the vector nums, an integer i representing the current index to consider for permutations, and an integer n representing the last index of the vector nums.
- If i equals n, the base case of the recursion is reached, meaning that a permutation is complete, and it is added to the result.
- If i is not equal to n, a loop from i to n is used to generate permutations by swapping elements.
- Inside the loop, the function first swaps the element at the current index i with the element at index j.
- After swapping, it recursively calls the permutation function with i + 1 to continue creating permutations with the next elements.

- After the recursive call returns, it swaps the elements back to revert the nums vector to its previous state before the next iteration of the loop.
- This ensures all possible permutations are considered, as each element gets the chance to be at index i.
- The permute function is the initial call that kicks off the permutation process by passing the nums vector, starting index 0, and the last index of the nums vector (nums.size() - 1).
- The permute function then returns the result vector containing all the permutations once they're generated.

Time Complexity:
The time complexity of this algorithm is O(N!*N), where N is the number of elements in the input array. This is because there are N! possible permutations. Within each recursive call, there is a constant amount of work done (swapping elements), so the work done in each call is not dependent on the size of the array N

Space Complexity:
The space complexity is O(N). This is due to the recursive stack space used by the function. Additionally, the result vector stores all permutations, but it is not counted in the space complexity because it is part of the output.

2. Permutations of a given string

```
class Solution {
   public:
    vector<string> result;
    set<string> resultSet;
    void permutation(string S, int i, int n) {
        if (i == n) {
            resultSet.insert(S);
            return;
        }

        for (int j = i; j <= n; j++) {
            swap(S[i], S[j]);
            permutation(S, i + 1, n);
            swap(S[i], S[j]);
        }
    }
    vector<string> find_permutation(string S) {
        permutation(S, 0, S.length() - 1);
        for (auto it = resultSet.begin(); it != resultSet.end(); it++) {
            result.push_back(*it);
```

```
        }
        return result;
    }
};
```

Code Explanation and Complexity

- The code uses a set called resultSet to store unique permutations and eliminate duplicates.
- The find_permutation function initializes the permutation generation process by calling the permutation function with the input string and necessary parameters.
- The permutation function uses a recursive approach to generate permutations and populate the resultSet with unique permutations.
- After the permutation generation is complete, the unique permutations are converted from the set to the result vector for output.

Time Complexity
The time complexity of this algorithm is O(N!*N), where N is the number of elements in the input array. This is because there are N! possible permutations. Within each recursive call, there is a constant amount of work done (swapping elements), so the work done in each call is not dependent on the size of the array N

Space Complexity:
The space complexity is O(N!*N) due to the storage of all unique permutations in the set resultSet. Additionally, the recursive calls' stack space used in the permutation generation contributes to the overall space complexity.

3. Permutations II

```
class Solution {
public:
    vector<vector<int>> result;
    set<vector<int>> resultSet;
    void permutation(vector<int> &nums,int i,int n){
        if(i==n){
            resultSet.insert(nums);
            return ;
        }

        for(int j=i;j<=n;j++){
            swap( nums[i],nums[j]);
```

```
            permutation(nums,i+1,n);
            swap( nums[i],nums[j]);
        }
    }
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        permutation(nums,0,nums.size()-1);
        for (const auto& vec : resultSet) {
            result.push_back(vec);
        }
        return result;
    }
};
```

Code Explanation and Complexity

1. Member Variables:

- vector<vector<int>> result: Used to store the final result, i.e., all unique permutations.
- set<vector<int>> resultSet: A set to store unique permutations.

2. Permutation Function:

- void permutation(vector<int> &nums, int i, int n): A recursive function that generates all permutations of the array nums starting from index i up to index n.
- Base case: If i is equal to n, a permutation is complete, and it is inserted into the resultSet.
- Recursive case: It uses a loop to swap elements at index i with each element from i to n and recursively calls itself with the updated array.

3. Permute Unique Function:

- vector<vector<int>> permuteUnique(vector<int>& nums): Initiates the permutation process by calling the permutation function and converts the unique permutations from resultSet to result.
- The result is then returned.

Time Complexity:
The time complexity is O(N! * N), where N is the number of elements in the input array. This is due to the generation of all unique permutations.

Space Complexity:
The space complexity is O(N! * N) as well.
resultSet stores unique permutations, and in the worst case, it requires O(N! * N) space.
result stores the final result, and in the worst case, it also requires O(N! * N) space.

_____ END _____