Day 86/180

Recursion on Subsequence 📑

---

1. Subsets

```cpp
class Solution {
public:
    void subsequence(vector<int>& arr, int index,int
n,vector<vector<int>>& ans, vector<int>& temp)
    {
        if(index==n)
        {
            ans.push_back(temp);
            return;
        }

        // Not included
        subsequence(arr, index+1, n,ans, temp) ;
        // Included
        temp.push_back(arr[index]);
        subsequence (arr, index+1, n, ans, temp);
        temp.pop_back();
    }
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> ans;
        vector<int>temp;
        subsequence(nums,0,nums.size(),ans,temp);
        return ans;
    }
};
```

<u>Code Explanation and Complexity</u>

- The subsequence function is a recursive helper function that generates subsets by considering each element either included or excluded.
- The base case of the recursion is when the entire array has been processed, at which point the current subset (temp) is added to the result (ans).

- The recursive calls are made for two scenarios: one without including the current element and another including the current element.
- The main subsets function initializes the result vector (ans) and a temporary vector (temp) to store subsets during recursion.
- It calls the subsequence function with the initial parameters to start the recursion process.
- The final list of subsets is returned.

Time Complexity:
The time complexity is O(2^n), where n is the length of the input array. This is because for each element, there are two possibilities (include or exclude), and there are a total of 2^n possible subsets.

Space Complexity:
The space complexity is O(n) due to the recursive call stack. Additionally, the space required for the temporary vector temp is also O(n) since, in the worst case, it can store all elements of the input array.

2. Generate Parentheses

```
class Solution {
public:
    void parenth(int n, int left, int right, vector<string>&ans, string
&temp)
    {
        if(left+right==2*n)
        {
            ans.push_back(temp);
            return;
        }
        // Left parenth
        if(left<n)
        {
            temp.push_back('(');
            parenth(n, left+1, right, ans, temp);
            temp.pop_back();
        }
        // Right parenth
        if(right<left)
        {
            temp.push_back(')');
            parenth(n, left, right+1, ans, temp);
            temp.pop_back();
        }
    }
```

```
    vector<string> generateParenthesis(int n) {
        vector<string>ans;
        string temp;
        parenth(n, 0,0, ans, temp);
        return ans;
    }
};
```

Code Explanation and Complexity

- The parenth function is a recursive helper function that generates combinations of well-formed parentheses.
- The base case of the recursion is when the total number of parentheses (left + right) reaches 2n, at which point the current combination (temp) is added to the result (ans).
    - Two recursive calls are made:
    - One for adding a left parenthesis if there are remaining left parentheses (left < n).
    - Another for adding a right parenthesis if it matches with a left parenthesis (right < left).
- Backtracking is performed by popping the last character from the temporary string to explore other possibilities.
- The main generateParenthesis function initializes the result vector (ans) and a temporary string (temp) to store combinations during recursion.
- It calls the parenth function with the initial parameters to start the recursion process.
- The final list of well-formed parentheses combinations is returned.

Time Complexity:

- The code uses recursion to generate all possible combinations of parentheses. For each position, there are two choices: either to add a left parenthesis or a right parenthesis. Therefore, the number of recursive calls is 2^n, where n is the input parameter. Each recursive call takes constant time, so the overall time complexity is O(2^n).

Space Complexity:

- The space complexity is O(n) due to the recursive call stack. Additionally, the space required for the temporary string temp is also O(n) since, in the worst case, it can store all characters of the generated parentheses combinations.

3. Given an array of size n, print all the sums possible from its subsequence.

```
class Solution {
   public:
    void print(vector<int> arr, int index, int n, int sum, vector<int>&
ans) {
        if (index == n) {
            ans.push_back(sum);
            return;
        }
        print(arr, index + 1, n, sum, ans);
        print(arr, index + 1, n, sum + arr[index], ans);
    }
    vector<int> subsetSums(vector<int> arr, int N) {
        vector<int> ans;
        int index = 0, n = arr.size(), sum = 0;
        print(arr, 0, n, 0, ans);
        return ans;
    }
};
```

Code Explanation and Complexity

The code defines a function print and another function subsetSums.

1. print function:
    - This is a recursive helper function that generates all possible subset sums.
    - It takes the input array arr, the current index index, the total number of elements n, the current sum sum, and a vector ‘ans’ to store the subset sums.
    - It recursively explores two possibilities for each element:
        - Exclude the current element by calling print with the same index and sum.
        - Include the current element by calling print with the incremented index and the updated sum.
    - The base case for the recursion is when the index reaches the size of the array (index == n), and at this point, the current sum is added to the ans vector.

2. subsetSums function:
    - This is the main function that initializes the process by calling the print method with the initial parameters.
    - It returns the vector ans containing all subset sums.

- The subsetSums method initializes the necessary variables (index, n, sum) and then calls the print function with the initial values.

Time Complexity:

The time complexity is O(2^N), where N is the number of elements in the input array. This is because for each element, there are two recursive calls (include/exclude), leading to an exponential time complexity.

Space Complexity:

The space complexity is O(n) due to the recursive call stack.

_____ END _____