Day 38 / 100 :

Topic - Backtracking, String

1 Problem statement: Combinations (Medium)

Given two integers n and k, return all possible combinations of k numbers chosen from the range [1, n].
You may return the answer in any order.

Example 1:
Input: n = 4, k = 2
Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
Explanation: There are 4 choose 2 = 6 total combinations.
Note that combinations are unordered, i.e., [1,2] and [2,1] are considered to be the same combination.

Example 2:
Input: n = 1, k = 1
Output: [[1]]
Explanation: There is 1 choose 1 = 1 total combination.

Solutions :

Approach 1 - Array + backtracking

intuition:

We can solve this problem using Array + Backtracking.

Approach:

We can easily understand the approach by seeing the code which is easy to understand with comments.

Complexity:

- Time complexity:

Time Complexity : O(k*nCk), here nCk means the binomial coefficient of picking k elements out of n elements. where nCk = C(n,k) = n!/(n−k)!×k!.

- Space complexity:

Space Complexity : O(nCk), as stated above the nCk here refers to the binomial coefficient.

```
class Solution {
private:
    void combine(int n, int k, vector<vector<int>> &output, vector<int> &temp, int
start){
        if(temp.size() == k){
            output.push_back(temp);
            return;
        }
        for(int i=start; i<=n; i++){
            temp.push_back(i);
            combine(n, k, output, temp, i+1);
            temp.pop_back();
        }
    }
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> output;
        vector<int> temp;
        combine(n, k, output, temp, 1);
        return output;
    }
};
```

2 Problem statement: Generate Parenthesis (Medium)

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]

Example 2:
Input: n = 1
Output: ["()"]

Solutions :

Approach 1 - backtracking

Given a number n, we have to generate all valid n pairs of parenthesis. Since we have to generate all the valid combinations, the first solution which comes to our mind is backtracking (recursion). Given the constraint, I would say that is the best solution.

We start with number of open brackets open = 0, and number of close brackets close = 0. Now at any given recursion level, either we can put one open bracket or one close bracket. The constraints would be that open can never be greater than n and that close < open at all times. Below code is self explanatory. Note I used string references to obtain a gain in speed.

```
class Solution {
public:
    void util(vector<string>& res, int open, int close, string& tmp, int n)
    {
        if(tmp.length()==2*n) {res.push_back(tmp); return;}
        if(open<n){
            tmp.push_back('(');
            util(res,open+1,close,tmp,n);
            tmp.pop_back();
        }
        if(close<open){
            tmp.push_back(')');
            util(res,open,close+1,tmp,n);
            tmp.pop_back();
        }
    }
    vector<string> generateParenthesis(int n) {
        int open=0,close=0; // open -> number of open brackets
                                // close -> number of close brackets
        vector<string> res;
        if(n==0) return res;
        string temp="";
        util(res,open,close,temp,n);
        return res;
    }
};
```

Time Complexity - Very important for these kind of questions. Since recursion is a tree, and here there are two recursive calls possible at any level. Heigh of the tree will be 2*n, since we are generating 2*n number of brackets. So, worst-case Time Complexity will be O(2^(2n)).
Space Complexity - O(2*n) (Stack space after using recursion)

Solutions :

Approach 1 - Dynamic Programming

Now, given the constraints in the problem, backtracking is good. But what if, n is large. We can't afford exponential solution. The next thought should be dynamic programming, and if there is a overlapping subproblems nature to it.
Suppose dp[i] contains all the valid parentheses possible of length 2*i. Suppose you got dp[2] which is { (()) , ()() }. Now what will be dp[3]? It can be written as -

(+ dp[0] +) + dp[2] = ()(()) and ()()()
(+ dp[1] +) + dp[1] = (())()
(+ dp[2] +) + dp[0] = ((())) and (()())
So you see, we have an overlapping subproblems structure. It's good to know here that this structure closely follows Catalan Numbers. (You can find the number of valid parentheses using Catalan Numbers).
dp[i] = "(" + dp[j] + ")" + dp[i-j-1] (Recursive relation. Similar to a binary tree generation).

```
vector<string> generateParenthesis(int n) {
        vector<vector<string>> dp(n+1); // cache to store all generated strings
        dp[0] = {""};
        for(int i=1;i<=n;i++){
            for(int j=0;j<i;j++){
                vector<string> left = dp[j];
                vector<string> right = dp[i-j-1];
                for(int k=0;k<left.size();k++){
                    for(int l=0;l<right.size();l++){
                        dp[i].push_back("(" + left[k] + ")" + right[l]);
                    }
                }
            }
        }
        return dp[n];
    }
```

Time Complexity - O(n^4)
Space Complexity - O(n)