**October 12**

**Recursion Problems**

# Problem 1: Generate Parentheses

**Problem Statement:**

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

**Link to problem:**

https://leetcode.com/problems/generate-parentheses/description/

---

**Example 1:**

**Input:** n = 3

**Output:** ["((()))","(()())","(())()","()(())","()()()"]

**Example 2:**

**Input:** n = 1

**Output:** ["()"]

---

**Solution:**

```
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        generateCombinations(result, "", 0, 0, n);
        return result;
    }

    // Recursive helper function to generate combinations
    private void generateCombinations(List<String> result, String current, int open, int close, int max) {
        // Base case: if the current string has reached the maximum length
        if (current.length() == max * 2) {
            result.add(current);
            return;
        }

        // Recursion case: add an opening bracket if there are any left
        if (open < max) {
            generateCombinations(result, current + "(", open + 1, close, max);
        }
```

```
        // Recursion case: add a closing bracket if it doesn't exceed the number of open brackets
        if (close < open) {
            generateCombinations(result, current + ")", open, close + 1, max);
        }
    }
}
```

---

**Explanation:**

- **Initial Call:**
  - We start with an empty string and call the recursive function generateCombinations(result, "", 0, 0, n) where open = 0 and close = 0 represent the number of opening and closing parentheses placed so far.
- **Base Case:**
  - When the current string has reached the length of 2 * n (i.e., when we have placed all n opening and n closing parentheses), we add the string to the result list.
- **Recursive Cases:**
  - If the number of opening parentheses used is less than n, we can add another opening parenthesis (.
  - If the number of closing parentheses used is less than the number of opening parentheses, we can add a closing parenthesis ).

This ensures that only valid combinations are generated.

---

**Time Complexity:**

**O(4^n / √n):** This is a known complexity for generating all valid combinations of parentheses, as the Catalan number bounds the number of valid strings.

**Space Complexity:**

**O(n):** The recursion depth can go up to n, and the space used to store the result list is proportional to the number of combinations.

---

# Problem 2: Permutations

**Problem Statement:**

Given an array of distinct integers, return all the possible permutations. You can return the answer in any order.

**Link to problem:**

https://leetcode.com/problems/permutations/description/

---

**Example 1:**

**Input:** nums = [1,2,3]
**Output:** [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

**Example 2:**

**Input:** nums = [0,1]
**Output:** [[0,1],[1,0]]

---

**Solution:**

```
class Solution {
public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), nums);
        return result;
    }

    // Backtracking helper function
    private void backtrack(List<List<Integer>> result, List<Integer> tempList, int[] nums) {
        // Base case: if the temporary list size equals the original list size
        if (tempList.size() == nums.length) {
            result.add(new ArrayList<>(tempList)); // Add the current permutation to the result
        } else {
            for (int i = 0; i < nums.length; i++) {
                // Skip used elements
                if (tempList.contains(nums[i])) continue;
                tempList.add(nums[i]); // Choose
                backtrack(result, tempList, nums); // Recurse
                tempList.remove(tempList.size() - 1); // Un-choose
            }
        }
    }
}
```

---

**Explanation:**

- **Initialization:**
    - We create a List<List<Integer>> result to store all the permutations.

    - We call the backtrack function with the initial parameters: result, an empty tempList, and the input array nums.

- **Base Case:**
    - The base case checks if the size of tempList is equal to the size of nums. If true, it means we've formed a complete permutation.
    - We then add a copy of tempList to result using new ArrayList<>(tempList).

- **Recursive Cases:**
    - We iterate through each element in nums.
    - **Checking for Used Elements:** We skip elements that are already in tempList to avoid duplicates.
    - If the element is not used, we add it to tempList.
    - We recursively call backtrack to continue building the permutation.
    - **Backtracking:** After the recursive call, we remove the last added element from tempList using tempList.remove(tempList.size() - 1) to backtrack and try other possibilities.

---

### Example Walkthrough

Input: nums = [1, 2, 3]

- Starting with an empty tempList, the process proceeds as follows:
    1. Add 1 → tempList = [1]
        - Add 2 → tempList = [1, 2]
            - Add 3 → tempList = [1, 2, 3] (Base case reached; add [1, 2, 3] to result)
            - Backtrack, remove 3, tempList = [1, 2]
        - Add 3 → tempList = [1, 3]
            - Add 2 → tempList = [1, 3, 2] (Base case reached; add [1, 3, 2] to result)
            - Backtrack, remove 2, tempList = [1, 3]
        - Backtrack to tempList = [1]
    2. Backtrack to empty tempList, add 2 → tempList = [2]
        - Add 1 → tempList = [2, 1]
            - Add 3 → tempList = [2, 1, 3] (Base case reached; add [2, 1, 3] to result)
        - Add 3 → tempList = [2, 3]
            - Add 1 → tempList = [2, 3, 1] (Base case reached; add [2, 3, 1] to result)
    3. Backtrack to empty tempList, add 3 → tempList = [3]
        - Add 1 → tempList = [3, 1]
            - Add 2 → tempList = [3, 1, 2] (Base case reached; add [3, 1, 2] to result)

    - Add 2 → tempList = [3, 2]
        - Add 1 → tempList = [3, 2, 1] (Base case reached; add [3, 2, 1] to result)

Final Result: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]].

---

**Time Complexity:**

**O(n! * n):** The n! factor comes from the number of permutations, and n comes from the time taken to copy the current permutation into the result list.

**Space Complexity:**

**O(n):** The space used by the temporary list and the recursion stack can go up to O(n).

---