

## Assignment 2: Algorithms Using Backtracking

### Question 1

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

### Algorithm

```
Function generateParenthesis (n):  
    results = []  
    Function backtrack (opened, close, res):  
        if opened == 0 and close == 0:  
            results.append(res)  
            return  
        if opened != 0:  
            backtrack (opened+1, close, res+'(')  
        if close > opened:  
            backtrack (opened, close-1, res+')'  
    backtrack (n, n, '')  
    return results.
```

### Code

```
class Solution:  
    def generateParenthesis(self, n: int) -> List[str]:  
  
        def backtrack(opened: int, close: int, res='') -> None:  
            if opened == 0 and close == 0:  
                results.append(res)  
                return
```

```
    if opened != 0:
        backtrack(opened - 1, close, res + '(')

    if close > opened:
        backtrack(opened, close - 1, res + ')')

results = []
backtrack(n, n, '')
return results
```

## Time complexity

For this problem, the valid parenthesis combinations are given by the Catalan Number Series.

It's in the form:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Which can be approximated by the Stirlings equation.

$$C_n \sim \frac{4^n}{n^{3/2} \sqrt{\pi}}$$

In addition to this, we copy the string to the result with is of  $O(n)$  time complexity. Which makes it,

$$\text{Time} = O(n * C_n)$$

Substituting with above equation and taking higher order growths, the total time complexity is,

$$\text{Time} = O\left(\frac{4^n}{\sqrt{n}}\right)$$

## Screenshot of Output

The screenshot displays a coding platform interface for the problem "22. Generate Parentheses". The problem description states: "Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses." Example 1 shows input n=3 and output ["((()))", "(()())", "()()()", "(()())", "()()()()"]. Example 2 shows input n=1 and output ["()"]. Constraints include 1 <= n <= 8. The solution code is written in Python3 and uses a backtracking approach. The test case result is "Accepted" with a runtime of 0 ms. The input is n=3 and the output is ["((()))", "(()())", "()()()", "(()())", "()()()"].

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
    def backtrack(opened: int, close: int, res='') -> None:
        if opened == 0 and close == 0:
            results.append(res)
            return
        if opened != 0:
            backtrack(opened - 1, close, res + '(')
        if close > opened:
            backtrack(opened, close - 1, res + ')')
```

## Question 2

**Aim:** Given a directed acyclic graph (DAG) of n nodes labeled from 0 to n - 1, find all possible paths from node 0 to node n - 1 and return them in any order.

The graph is given as follows: graph[i] is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node graph[i][j]).

## Algorithm

Function AllPathsSourceTarget(G)

n = G.rows

target = n-1

results = []

path = [0]

DFS-VISIT-PATHS(G, 0, target, path, results)

return results

Function DFS-VISIT-PATHS(G, u, target, path, results)

if u == target

    results.append(copy(path))

    return

for each v in G.adj[u]

    path.append(v)

    DFS-VISIT-PATHS(G, v, target, path, results)

    path.pop()

## Code

```
class Solution:
    def allPathsSourceTarget(self, graph):
        target = len(graph) - 1
        results = []

        def dfs(node, path):
            if node == target:
                results.append(list(path))
                return

            for neighbor in graph[node]:
                path.append(neighbor)
                dfs(neighbor, path)
                path.pop()

        dfs(0, [0])
        return results
```

## Time complexity

Let's take  $V$  as number of vertices,  $E$  as number of edges, and  $N_p$  as total paths from source to target.

In the worst case when path exists through every  $i < j$ , no of paths from node 0 to  $n - 1$  is:

$$N_p = 2^{V-2}$$

Each path involved visiting  $V$  nodes, and copying current path into result list when target is reached, which takes  $O(V)$ .

Hence the total time complexity is

$$T(V) = \sum_{p=1}^{N_p} (\text{nodes in path } p)$$

$$T(V) = O(V \cdot 2^V)$$

## Screenshot of Output

The screenshot displays a coding interface for the problem "797. All Paths From Source to Target". The problem description states: "Given a directed acyclic graph (DAG) of  $n$  nodes labeled from  $0$  to  $n - 1$ , find all possible paths from node  $0$  to node  $n - 1$  and return them in any order. The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node  $i$  (i.e., there is a directed edge from node  $i$  to node `graph[i][j]`).

**Example 1:**

```
graph LR; 0((0)) --> 1((1)); 0 --> 2((2)); 1 --> 3((3)); 2 --> 3
```

**Input:** `graph = [[1,2],[3],[3],[1]]`  
**Output:** `[[0,1,3],[0,2,3]]`  
**Explanation:** There are two paths:  $0 \rightarrow 1 \rightarrow 3$  and  $0 \rightarrow 2 \rightarrow 3$ .

**Example 2:**

```
graph LR; 0((0)) --> 1((1)); 0 --> 2((2)); 1 --> 3((3)); 2 --> 3
```

The Python code solution is as follows:

```
class Solution:
    def allPathsSourceTarget(self, graph):
        target = len(graph) - 1
        results = []

        def dfs(node, path):
            if node == target:
                results.append(list(path))
                return

            for neighbor in graph[node]:
                path.append(neighbor)
                dfs(neighbor, path)
                path.pop()

        dfs(0, [0])

        return results
```

The test results show "Accepted" with a runtime of 0 ms. Two test cases are passed:

- Case 1: `graph = [[1,2],[3],[3],[1]]`, Output: `[[0,1,3],[0,2,3]]`
- Case 2: (Details not fully visible)

A user profile menu for "Varghese K James" is visible on the right side of the interface, showing options like "My Lists", "Notebook", "Progress", "Points", "Try New Features", "Orders", "My Playgrounds", "Settings", "Appearance", and "Sign Out".

## **Learning Outcomes**

- Learned about using backtracking algorithms to solve questions involving permutations and combination
- Learned about bespoke optimisations done to solve competitive programming questions.