DOMAIN WINTER CAMP

DAY - 9

Student Name: Navneet Sharma **UID:** 22BCS15680

Branch: BE-CSE **Section:** 22BCS_FL_IOT-603-B

BackTracking

Very Easy:

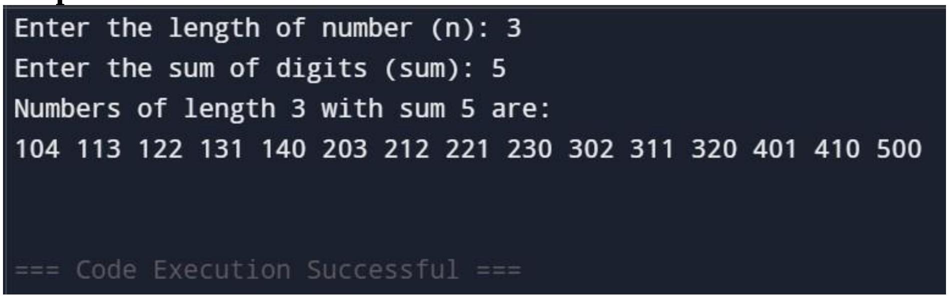
1. Generate Numbers with a Given Sum

Generate all numbers of length n whose digits sum up to a target value sum, The digits of the number will be between 0 and 9, and we will generate combinations of digits such that their sum equals the target.

Solution:

```
#include <iostream>
#include <vector>
#include <string> using
namespace std;

class Solution { public:
    void findNumbers(int n, int sum, string current, vector<string>& result) { if (n
        == 0 && sum == 0) {
            result.push_back(current); return;
        }

        // If there are no more digits to place or the sum is negative, return if (n
        == 0 || sum < 0) {
            return;
        }

        // Starting digit should be between 0 to 9
        for (int i = (current.empty() ? 1 : 0); i <= 9; ++i) {  // For the first digit, skip 0 if n >
1
            findNumbers(n - 1, sum - i, current + to_string(i), result);
        }
    }

    vector<string> generateNumbers(int n, int sum) { vector<string>
        result;
        findNumbers(n, sum, "", result); return
        result;
    }
};

int main() {
    int n, sum;
    cout << "Enter the length of number (n): ";
    cin >> n;
    cout << "Enter the sum of digits (sum): ";
    cin >> sum;

    Solution solution;
    vector<string> result = solution.generateNumbers(n, sum);

    cout << "Numbers of length " << n << " with sum " << sum << " are:\n";
    for (const string& number : result) {
        cout << number << " ";
    }
    cout << endl;

    return 0;
}
```

Output:



```
Enter the length of number (n): 3
Enter the sum of digits (sum): 5
Numbers of length 3 with sum 5 are:
104 113 122 131 140 203 212 221 230 302 311 320 401 410 500


=== Code Execution Successful ===
```

Easy:

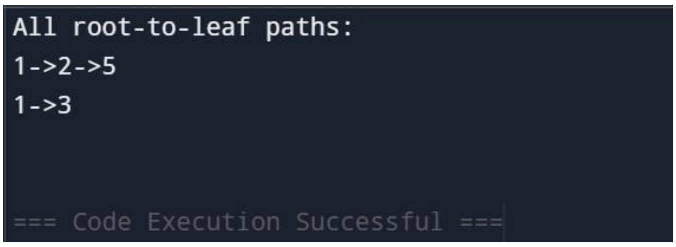
2. Binary Tree Paths

Given the root of a binary tree, return all root-to-leaf paths in any order.
A leaf is a node with no children.

Solution:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    void backtrack(TreeNode* node, string path, vector<string>& result) {
        if (node == nullptr) {
            return;
        }

        path += to_string(node->val);

        if (!node->left && !node->right) {
            result.push_back(path);
        } else {
            path += "->";
            backtrack(node->left, path, result);
            backtrack(node->right, path, result);
        }
    }

    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> result;
        if (root != nullptr) {
            backtrack(root, "", result);
        }
        return result;
    }
};

int main() {

    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->right = new TreeNode(5);

    Solution solution;
    vector<string> paths = solution.binaryTreePaths(root);

    cout << "All root-to-leaf paths: \n";
    for (const string& path : paths) {
        cout << path << endl;
    }

    return 0;
}
```

Output:



```
All root-to-leaf paths:
1->2->5
1->3


=== Code Execution Successful ===
```

Medium:

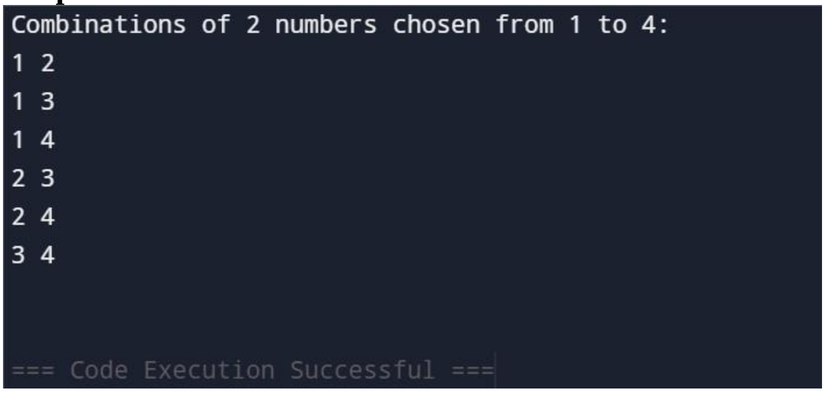
3. Combinations

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.

Solution:

```
#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    void backtrack(int n, int k, int start, vector<int>& path, vector<vector<int>>& result) {
        // If the combination is complete, add it to the result
        if (path.size() == k) {
            result.push_back(path);
            return;
        }

        // Explore numbers from start to n
        for (int i = start; i <= n; i++) {
            path.push_back(i);  // Include i in the combination
            backtrack(n, k, i + 1, path, result);  // Move to the next number
            path.pop_back();  // Backtrack and remove i from the combination
        }
    }

    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> result;
        vector<int> path;
        backtrack(n, k, 1, path, result);  // Start the backtracking from 1
        return result;
    }
};

int main() {
    Solution solution;
    int n = 4, k = 2;
    vector<vector<int>> combinations = solution.combine(n, k);

    cout << "Combinations of " << k << " numbers chosen from 1 to " << n << ": \n";
    for (const auto& combination : combinations) {
        for (int num : combination) {
            cout << num << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:



```
Combinations of 2 numbers chosen from 1 to 4:
1 2
1 3
1 4
2 3
2 4
3 4


=== Code Execution Successful ===
```

Hard:

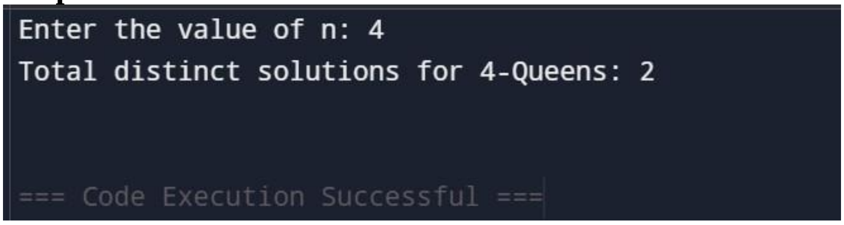
4. N-Queens II

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.
Given an integer n, return the number of distinct solutions to the n-queens puzzle.

Solution:

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    int totalNQueens(int n) {
        int result = 0;
        unordered_set<int> cols, diag1, diag2;
        backtrack(0, n, cols, diag1, diag2, result);
        return result;
    }

    void backtrack(int row, int n, unordered_set<int>& cols, unordered_set<int>&
diag1, unordered_set<int>& diag2, int& result) {
        if (row == n) {
            result++;
            return;
        }

        for (int col = 0; col < n; col++) {
            if (cols.count(col) || diag1.count(row - col) || diag2.count(row + col)) {
                continue;
            }

            cols.insert(col);
            diag1.insert(row - col);
            diag2.insert(row + col);

            backtrack(row + 1, n, cols, diag1, diag2, result);

            cols.erase(col);
            diag1.erase(row - col);
            diag2.erase(row + col);
        }
    }
};

int main() {
    Solution solution;
    int n;
    cout << "Enter the value of n: ";
    cin >> n;
    int result = solution.totalNQueens(n);
    cout << "Total distinct solutions for " << n << "-Queens: " << result << endl;
    return 0;
}
```

Output:



```
Enter the value of n: 4
Total distinct solutions for 4-Queens: 2


=== Code Execution Successful ===
```

Very Hard:

5. Word Ladder II

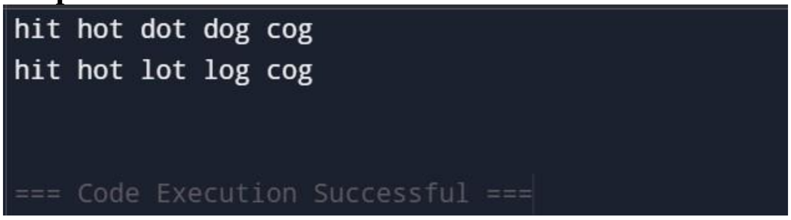
A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:
Every adjacent pair of words differs by a single letter.
Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.

sk == endWord
Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

Solution:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
#include <unordered_map>
using namespace std;

class Solution {
public:
    vector<vector<string>> findLadders(string beginWord, string endWord,
vector<string>& wordList) {
        unordered_set<string> wordSet(wordList.begin(), wordList.end());
        vector<vector<string>> result;
        if (wordSet.find(endWord) == wordSet.end()) {
            return result;
        }

        unordered_map<string, vector<string>> adjList;
        bfs(beginWord, endWord, wordSet, adjList);
        vector<string> path = {beginWord};
        backtrack(beginWord, endWord, adjList, path, result);
        return result;
    }

    void bfs(string& beginWord, string& endWord, unordered_set<string>& wordSet,
unordered_map<string, vector<string>>& adjList) {
        queue<string> q;
        unordered_map<string, int> dist;
        dist[beginWord] = 0;
        q.push(beginWord);

        while (!q.empty()) {
            string current = q.front();
            q.pop();
            int currentDist = dist[current];

            for (int i = 0; i < current.size(); ++i) {
                string temp = current;
                for (char c = 'a'; c <= 'z'; ++c) {
                    temp[i] = c;
                    if (wordSet.find(temp) != wordSet.end()) {
                        if (dist.find(temp) == dist.end()) {
                            dist[temp] = currentDist + 1;
                            q.push(temp);
                        }
                        if (dist[temp] == currentDist + 1) {
                            adjList[current].push_back(temp);
                        }
                    }
                }
            }
        }
    }

    void backtrack(const string& beginWord, const string& endWord,
unordered_map<string, vector<string>>& adjList, vector<string>& path,
vector<vector<string>>& result) {
        if (beginWord == endWord) {
            result.push_back(path);
            return;
        }

        for (const string& neighbor : adjList[beginWord]) {
            path.push_back(neighbor);
            backtrack(neighbor, endWord, adjList, path, result);
            path.pop_back();
        }
    }
};

int main() {
    Solution solution;
    string beginWord = "hit";
    string endWord = "cog";
    vector<string> wordList = {"hot", "dot", "dog", "lot", "log", "cog"};
    vector<vector<string>> result = solution.findLadders(beginWord, endWord,
wordList);

    for (const auto& sequence : result) {
        for (const auto& word : sequence) {
            cout << word << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:



```
hit hot dot dog cog
hit hot lot log cog



=== Code Execution Successful ===
```