DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Ansh Jain **UID: 22BCS15216**
Branch: BE-CSE **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

Day 9 : 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.

Example 1:
Input: n = 2 and sum = 5
Output: 14 23 32 41 50

Example 2:
Input: n = 3 and sum = 5
Output: 104 113 122 131 140 203 212 221 230 302 311 320 401 410 500

Constraints:
1 <= n <= 9: The number of digits must be between 1 and 9.
1 <= sum <= 100: The sum of the digits must be between 1 and 100.
The first digit cannot be zero if n > 1.

CODE:
```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

vector<int> bfsTraversal(int n, vector<vector<int>>& adj) {
    vector<int> result;         // To store BFS traversal order
    vector<bool> visited(n, false); // To track visited nodes
    queue<int> q;               // Queue for BFS

    // Start BFS from vertex 0
```

```
    q.push(0);
    visited[0] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        result.push_back(current);

        // Visit all neighbors of the current node
        for (int neighbor : adj[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }

    return result;
}

int main() {
    // Example 1
    int n1 = 5;
    vector<vector<int>> adj1 = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    vector<int> result1 = bfsTraversal(n1, adj1);
    cout << "Example 1 BFS: ";
    for (int node : result1) {
        cout << node << " ";
    }
    cout << endl;

    // Example 2
    int n2 = 5;
    vector<vector<int>> adj2 = {{1, 2}, {0, 2}, {0, 1, 3, 4}, {2}, {2}};
    vector<int> result2 = bfsTraversal(n2, adj2);
    cout << "Example 2 BFS: ";
    for (int node : result2) {
        cout << node << " ";
    }
    cout << endl;

    // Example 3
    int n3 = 5;
    vector<vector<int>> adj3 = {{1}, {0, 2, 3}, {1}, {1, 4}, {3}};
    vector<int> result3 = bfsTraversal(n3, adj3);
    cout << "Example 3 BFS: ";
    for (int node : result3) {
        cout << node << " ";
    }
```

```
    cout << endl;

    return 0;
}
```

OUTPUT:

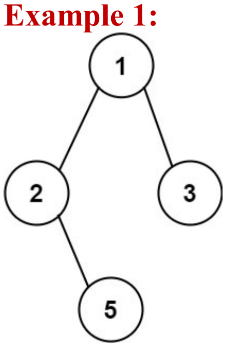
```
Example 1 BFS: 0 2 3 1 4
Example 2 BFS: 0 1 2 3 4
Example 3 BFS: 0 1 2 3 4
```

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.

Example 1:



Input: root = [1,2,3,null,5]Output: ["1->2->5","1->3"]
Example 2:
Input: root = [1]Output: ["1"]

Constraints:
The number of nodes in the tree is in the range [1, 100].
-100 <= Node.val <= 100

Reference: https://leetcode.com/problems/binary-tree-paths/description/?envType=problem-list-v2&envId=backtracking

CODE:

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

// Definition for a binary tree node.
struct TreeNode {
```

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

void dfs(TreeNode* root, string currentPath, vector<string>& result) {
    // If the current node is null, return
    if (root == nullptr) {
        return;
    }

    // Add the current node's value to the path
    currentPath += to_string(root->val);

    // If it's a leaf node, add the path to the result
    if (root->left == nullptr && root->right == nullptr) {
        result.push_back(currentPath);
    } else {
        // Otherwise, continue the path to the left and right children
        currentPath += "->"; // To separate node values
        dfs(root->left, currentPath, result);
        dfs(root->right, currentPath, result);
    }
}

vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> result;
    dfs(root, "", result);
    return result;
}

int main() {
    // Construct the binary tree: [1,2,3,null,5]
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->right = new TreeNode(5);

    // Get all root-to-leaf paths
    vector<string> paths = binaryTreePaths(root);

    // Print the result
    for (const string& path : paths) {
        cout << path << " ";
    }
    cout << endl;

    return 0;
```

```
}
```

OUTPUT:

```
1->2->5 1->3
```

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.

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.

Constraints:

1 <= n <= 20
1 <= k <= n

Reference: https://leetcode.com/problems/combinations/description/?envType=study-plan-v2&envId=top-interview-150

CODE:

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

// Helper function to generate combinations using backtracking
void backtrack(int start, int n, int k, vector<int>& current, vector<vector<int>>& result) {
    // If the current combination has reached size k, add it to the result
    if (current.size() == k) {
        result.push_back(current);
        return;
```

```
    }

    // Explore numbers from start to n
    for (int i = start; i <= n; ++i) {
        current.push_back(i);           // Add number i to the combination
        backtrack(i + 1, n, k, current, result); // Recurse with next number
        current.pop_back();             // Backtrack
    }
}

vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> result;
    vector<int> current;
    backtrack(1, n, k, current, result); // Start from 1 and explore combinations
    return result;
}

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

    // Print the result
    for (const auto& combination : combinations) {
        cout << "[";
        for (int num : combination) {
            cout << num << " ";
        }
        cout << "] ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT:

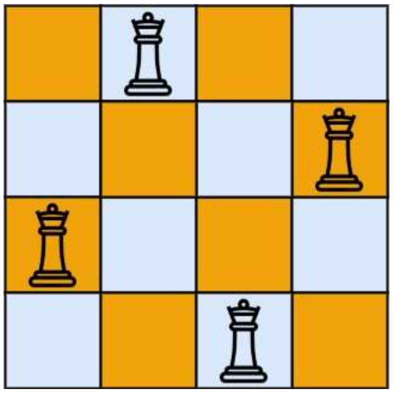
```
[1 2 ] [1 3 ] [1 4 ] [2 3 ] [2 4 ] [3 4 ]
```

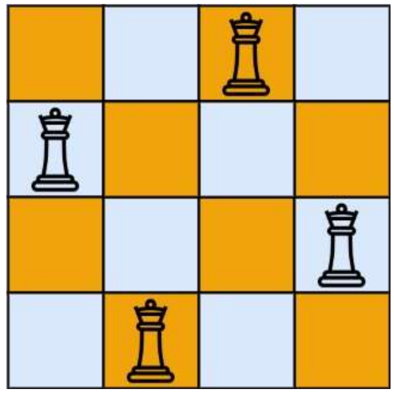
Hard:

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.

Example 1:





Input: n = 4
Output: 2
Explanation: There are two distinct solutions to the 4-queens puzzle as shown.

Example 2:
Input: n = 1
Output: 1

Constraints:
1 <= n <= 9

Reference: https://leetcode.com/problems/n-queens-ii/description/?envType=study-plan-v2&envId=top-interview-150

CODE:

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

class NQueens {
public:
    int totalSolutions = 0;

    void solveNQueens(int n) {
        vector<int> board(n, -1); // board[i] represents the column of the queen in row i
        vector<bool> cols(n, false); // To track if a column is under attack
        vector<bool> diag1(2 * n - 1, false); // To track if a major diagonal is under attack
        vector<bool> diag2(2 * n - 1, false); // To track if a minor diagonal is under attack
        backtrack(n, 0, board, cols, diag1, diag2);
    }

    void backtrack(int n, int row, vector<int>& board, vector<bool>& cols, vector<bool>& diag1,
vector<bool>& diag2) {
        // If all queens are placed
        if (row == n) {
            totalSolutions++; // Found a solution
            return;
```

```
        }

        // Try placing a queen in each column of the current row
        for (int col = 0; col < n; ++col) {
            int d1 = row - col + (n - 1); // Major diagonal index
            int d2 = row + col; // Minor diagonal index

            // Check if the column or diagonals are under attack
            if (cols[col] || diag1[d1] || diag2[d2]) continue;

            // Place the queen
            board[row] = col;
            cols[col] = true;
            diag1[d1] = true;
            diag2[d2] = true;

            // Recur for the next row
            backtrack(n, row + 1, board, cols, diag1, diag2);

            // Backtrack, remove the queen
            board[row] = -1;
            cols[col] = false;
            diag1[d1] = false;
            diag2[d2] = false;
        }
    }

    int getTotalSolutions() {
        return totalSolutions;
    }
};

int main() {
    NQueens solver;
    int n;
    cout << "Enter n: ";
    cin >> n;

    solver.solveNQueens(n);
    cout << "Total distinct solutions: " << solver.getTotalSolutions() << endl;

    return 0;
}
```

OUTPUT:

```
Enter n: 1
Total distinct solutions: 1
```

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].

Example 1:
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"

Example 2:
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: []
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:
1 <= beginWord.length <= 5
endWord.length == beginWord.length
1 <= wordList.length <= 500
wordList[i].length == beginWord.length
beginWord, endWord, and wordList[i] consist of lowercase English letters.
beginWord != endWord
All the words in wordList are unique.
The sum of all shortest transformation sequences does not exceed 105.

Reference:https://leetcode.com/problems/word-ladder-ii/description/?envType=problem-list-v2&envId=backtracking

CODE:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
```

```
#include <unordered_map>
#include <string>
#include <algorithm>  // Include this header for reverse function

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;

        // Early exit if endWord is not in the wordList
        if (wordSet.find(endWord) == wordSet.end()) {
            return result;
        }

        // BFS to find the shortest transformation sequences
        unordered_map<string, vector<string>> parentMap; // to store parent words for
backtracking
        unordered_set<string> visited;
        queue<string> q;
        q.push(beginWord);
        visited.insert(beginWord);

        bool found = false;
        bool levelFound = false;

        while (!q.empty() && !found) {
            int size = q.size();
            visited.clear();

            // Explore current level
            for (int i = 0; i < size; i++) {
                string current = q.front();
                q.pop();

                // Try all possible transformations
                for (int j = 0; j < current.length(); j++) {
                    char originalChar = current[j];
                    for (char c = 'a'; c <= 'z'; c++) {
```

```
                    if (c == originalChar) continue;
                    current[j] = c;

                    // If the new word is in the wordSet, we continue
                    if (wordSet.find(current) != wordSet.end()) {
                        parentMap[current].push_back(q.front());
                        if (current == endWord) {
                            found = true;
                        }
                        if (!visited.count(current)) {
                            visited.insert(current);
                            q.push(current);
                        }
                    }
                }
                current[j] = originalChar;
            }
        }
    }

    // Backtrack to find all paths from endWord to beginWord
    vector<string> path = {endWord};
    backtrack(beginWord, endWord, parentMap, path, result);

    return result;
}

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

    for (const string& parent : parentMap[endWord]) {
        path.push_back(parent);
        backtrack(beginWord, parent, parentMap, 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& path : result) {
        for (const auto& word : path) {
            cout << word << " ";
        }
        cout << endl;
    }

    return 0;
}
```

OUTPUT:

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