**Q1 .Generate Numbers with a Given Sum**

#include <iostream>

#include <vector>

using namespace std;

void generateNumbers(int n, int sum, vector<int>& current, vector<vector<int>>& result) {

if (sum == 0 && current.size() == n) {

result.push\_back(current);

return;

}

if (sum < 0 || current.size() >= n) {

return;

}

for (int i = 1; i <= 9; ++i) {

current.push\_back(i);

generateNumbers(n, sum - i, current, result);

current.pop\_back();

}

}

vector<vector<int>> generateNumbersWithSum(int n, int sum) {

vector<vector<int>> result;

vector<int> current;

generateNumbers(n, sum, current, result);

return result;

}

int main() {

int n = 3, sum = 6;

vector<vector<int>> result = generateNumbersWithSum(n, sum);

for (const auto& combination : result) {

for (int num : combination) {

cout << num << " ";

}

cout << endl;

}

return 0;

}

**Q2. Binary Tree Paths**

#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) {}

};

void findPaths(TreeNode\* node, string path, vector<string>& paths) {

if (!node) return;

path += to\_string(node->val);

if (!node->left && !node->right) {

paths.push\_back(path);

return;

}

path += "->";

findPaths(node->left, path, paths);

findPaths(node->right, path, paths);

}

vector<string> binaryTreePaths(TreeNode\* root) {

vector<string> paths;

findPaths(root, "", paths);

return paths;

}

int main() {

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(3);

root->left->right = new TreeNode(5);

vector<string> paths = binaryTreePaths(root);

for (const string& path : paths) {

cout << path << endl;

}

return 0;

}

**Q3. Combinations leet code problem**

#include <iostream>

#include <vector>

using namespace std;

void combineHelper(int start, int n, int k, vector<int>& current, vector<vector<int>>& result) {

if (k == 0) {

result.push\_back(current);

return;

}

for (int i = start; i <= n; ++i) {

current.push\_back(i);

combineHelper(i + 1, n, k - 1, current, result);

current.pop\_back();

}

}

vector<vector<int>> combine(int n, int k) {

vector<vector<int>> result;

vector<int> current;

combineHelper(1, n, k, current, result);

return result;

}

int main() {

int n = 4, k = 2;

vector<vector<int>> result = combine(n, k);

for (const auto& combination : result) {

for (int num : combination) {

cout << num << " ";

}

cout << endl;

}

return 0;

}

Q**4. N-Queens II**

#include <iostream>

#include <vector>

using namespace std;

bool isSafe(vector<int>& queens, int row, int col) {

for (int i = 0; i < row; ++i) {

if (queens[i] == col ||

abs(queens[i] - col) == abs(i - row)) {

return false;

}

}

return true;

}

void solveNQueens(int n, int row, vector<int>& queens, int& count) {

if (row == n) {

++count;

return;

}

for (int col = 0; col < n; ++col) {

if (isSafe(queens, row, col)) {

queens[row] = col;

solveNQueens(n, row + 1, queens, count);

queens[row] = -1;

}

}

}

int totalNQueens(int n) {

vector<int> queens(n, -1);

int count = 0;

solveNQueens(n, 0, queens, count);

return count;

}

int main() {

int n = 8; // Example for 8-Queens

cout << "Number of solutions for " << n << "-Queens: " << totalNQueens(n) << endl;

return 0;

}

**Q5. Word Ladder II**

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_set>

#include <unordered\_map>

#include <queue>

using namespace std;

vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {

unordered\_set<string> wordSet(wordList.begin(), wordList.end());

if (!wordSet.count(endWord)) return {};

unordered\_map<string, vector<string>> graph;

unordered\_map<string, int> levels;

queue<string> q;

q.push(beginWord);

levels[beginWord] = 0;

while (!q.empty()) {

string word = q.front();

q.pop();

string originalWord = word;

for (int i = 0; i < word.size(); ++i) {

char originalChar = word[i];

for (char c = 'a'; c <= 'z'; ++c) {

word[i] = c;

if (wordSet.count(word) && (levels.find(word) == levels.end() || levels[word] == levels[originalWord] + 1)) {

graph[originalWord].push\_back(word);

if (levels.find(word) == levels.end()) {

levels[word] = levels[originalWord] + 1;

q.push(word);

}

}

}

word[i] = originalChar;

}

}

vector<vector<string>> results;

vector<string> path;

function<void(string)> backtrack = [&](string word) {

path.push\_back(word);

if (word == endWord) {

results.push\_back(path);

} else {

for (const string& neighbor : graph[word]) {

backtrack(neighbor);

}

}

path.pop\_back();

};

if (levels.find(endWord) != levels.end()) {

backtrack(beginWord);

}

return results;

}

int main() {

string beginWord = "hit";

string endWord = "cog";

vector<string> wordList = {"hot", "dot", "dog", "lot", "log", "cog"};

vector<vector<string>> ladders = findLadders(beginWord, endWord, wordList);

for (const auto& ladder : ladders) {

for (const string& word : ladder) {

cout << word << " ";

}

cout << endl;

}

return 0;

}