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

#include <iostream>

#include <vector>

using namespace std;

void findCombinations(int target, int start, vector<int>& current, vector<vector<int>>& result) {

if (target == 0) {

result.push\_back(current);

return;

}

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

current.push\_back(i);

findCombinations(target - i, i, current, result);

current.pop\_back();

}

}

vector<vector<int>> generateNumbersWithGivenSum(int target) {

vector<vector<int>> result;

vector<int> current;

findCombinations(target, 1, current, result);

return result;

}

int main() {

int target = 5; // Example target

vector<vector<int>> result = generateNumbersWithGivenSum(target);

for (const auto& combination : result) {

for (int num : combination) {

cout << num << " ";

}

cout << endl;

}

return 0;

}

1. **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(NULL), right(NULL) {}

};

void findPaths(TreeNode\* root, string current, vector<string>& result) {

if (!root) return;

current += to\_string(root->val);

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

result.push\_back(current);

} else {

current += "->";

findPaths(root->left, current, result);

findPaths(root->right, current, result);

}

}

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

vector<string> result;

findPaths(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);

vector<string> paths = binaryTreePaths(root);

for (const string& path : paths) {

cout << path << endl;

}

return 0;

}

1. **Combinations of k Numbers from the Range [1, n]**

#include <iostream>

#include <vector>

using namespace std;

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

if (current.size() == k) {

result.push\_back(current);

return;

}

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

current.push\_back(i);

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

current.pop\_back();

}

}

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

vector<vector<int>> result;

vector<int> current;

combineHelper(n, k, 1, 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;

}

1. **N-Queens II**

#include <iostream>

#include <vector>

using namespace std;

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

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

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

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(row, col, queens)) {

queens[row] = col;

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

}

}

}

int totalNQueens(int n) {

int count = 0;

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

solveNQueens(n, 0, queens, count);

return count;

}

int main() {

int n = 4; // Example input

cout << "Total solutions: " << totalNQueens(n) << endl;

return 0;

}

1. **Word Ladder II**

#include <iostream>

#include <vector>

#include <unordered\_set>

#include <queue>

#include <string>

using namespace std;

void findLaddersHelper(const string& beginWord, const string& endWord, unordered\_set<string>& wordList,

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);

findLaddersHelper(neighbor, endWord, wordList, adjList, path, result);

path.pop\_back();

}

}

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

vector<vector<string>> result;

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

queue<string> q;

q.push(beginWord);

wordList.insert(endWord);

bool found = false;

unordered\_set<string> visited;

visited.insert(beginWord);

while (!q.empty() && !found) {

unordered\_set<string> localVisited;

int size = q.size();

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

string word = q.front();

q.pop();

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

string temp = word;

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

temp[j] = c;

if (wordList.find(temp) != wordList.end() && visited.find(temp) == visited.end()) {

adjList[word].push\_back(temp);

localVisited.insert(temp);

if (temp == endWord) {

found = true;

}

q.push(temp);

}

}

}

}

for (const string& word : localVisited) {

visited.insert(word);

}

}

if (found) {

vector<string> path = {beginWord};

findLaddersHelper(beginWord, endWord, wordList, adjList, path, result);

}

return result;

}

int main() {

string beginWord = "hit", endWord = "cog";

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

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

for (const auto& sequence : result) {

for (const string& word : sequence) {

cout << word << " ";

}

cout << endl;

}

return 0;

}