ARRAYS

### 1. Two Sum

\*\*Problem:\*\* Given an array of integers `nums` and an integer `target`, return the indices of the two numbers such that they add up to `target`.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <unordered\_map>

#include <vector>

std::vector<int> twoSum(std::vector<int>& nums, int target) {

std::unordered\_map<int, int> numMap; //num:index

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

int complement = target - nums[i];

if (numMap.find(complement) != numMap.end()) {

return {numMap[complement], i};

}

numMap[nums[i]] = i;

}

return {};

}

int main() {

std::vector<int> nums = {2, 7, 11, 15};

int target = 9;

std::vector<int> result = twoSum(nums, target);

std::cout << "Indices: " << result[0] << ", " << result[1] << std::endl;

return 0;

}

```

### 2. Best Time to Buy and Sell Stock

\*\*Problem:\*\* Given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day, find the maximum profit you can achieve. You may only complete one transaction (i.e., buy one and sell one share of the stock).

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <vector>

#include <algorithm>

int maxProfit(std::vector<int>& prices) {

int minPrice = INT\_MAX;

int maxProfit = 0;

for (int price : prices) {

minPrice = std::min(minPrice, price);

maxProfit = std::max(maxProfit, price - minPrice);

}

return maxProfit;

}

int main() {

std::vector<int> prices = {7, 1, 5, 3, 6, 4};

std::cout << "Max Profit: " << maxProfit(prices) << std::endl;

return 0;

}

```

### 3. Maximum Subarray recheck

\*\*Problem:\*\* Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

\*\*Solution:\*\* (Kadane’s Algorithm)

```cpp

#include <iostream>

#include <vector>

#include <algorithm>

int maxSubArray(std::vector<int>& nums) {

int maxSum = nums[0];

int currentSum = nums[0];

for (int i = 1; i < nums.size(); i++) {

currentSum = std::max(nums[i], currentSum + nums[i]);

maxSum = std::max(maxSum, currentSum);

}

return maxSum;

}

int main() {

std::vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

std::cout << "Maximum Subarray Sum: " << maxSubArray(nums) << std::endl;

return 0;

}

```

### 4. Container With Most Water

\*\*Problem:\*\* Given `n` non-negative integers `a1, a2, ..., an`, where each represents a point at coordinate `(i, ai)`. `n` vertical lines are drawn such that the two endpoints of the line `i` are at `(i, ai)` and `(i, 0)`. Find two lines, which, together with the x-axis, forms a container that holds the most water.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <vector>

#include <algorithm>

int maxArea(std::vector<int>& height) {

int maxArea = 0;

int left = 0, right = height.size() - 1;

while (left < right) {

int width = right - left;

int h = std::min(height[left], height[right]);

maxArea = std::max(maxArea, width \* h);

if (height[left] < height[right])

left++;

else

right--;

}

return maxArea;

}

int main() {

std::vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};

std::cout << "Max Area: " << maxArea(height) << std::endl;

return 0;

}

```

### 5. Rotate Array

\*\*Problem:\*\* Given an array, rotate the array to the right by `k` steps, where `k` is non-negative.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <vector>

#include <algorithm>

void rotate(std::vector<int>& nums, int k) {

k = k % nums.size();

////(  k might be larger than the size of the array nums. To handle this, we take k modulo nums.size().

 This operation ensures that rotating by k steps has the same effect as rotating by k % nums.size() steps.)

std::reverse(nums.begin(), nums.end()); //reverse entire array

std::reverse(nums.begin(), nums.begin() + k); //reverse first 3 elements

std::reverse(nums.begin() + k, nums.end()); //reverse remaining elements

}

int main() {

std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7};

int k = 3;

rotate(nums, k);

std::cout << "Rotated Array: ";

for (int num : nums)

std::cout << num << " ";

std::cout << std::endl;

return 0;

}

STRINGS

### 1. Reverse String

\*\*Problem:\*\* Write a function that reverses a string. The input string is given as an array of characters `s`.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <vector>

#include <algorithm>

void reverseString(std::vector<char>& s) {

std::reverse(s.begin(), s.end());

}

int main() {

std::vector<char> s = {'h', 'e', 'l', 'l', 'o'};

reverseString(s);

std::cout << "Reversed String: ";

for (char c : s)

std::cout << c;

std::cout << std::endl;

return 0;

}

```

### 2. Valid Palindrome

\*\*Problem:\*\* Given a string `s`, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <cctype>

#include <string>

bool isPalindrome(std::string s) {

int left = 0, right = s.length() - 1;

while (left < right) {

while (left < right && !isalnum(s[left])) left++;

while (left < right && !isalnum(s[right])) right--;

if (tolower(s[left]) != tolower(s[right])) return false;

left++;

right--;

}

return true;

}

int main() {

std::string s = "A man, a plan, a canal: Panama";

std::cout << "Is Valid Palindrome: " << (isPalindrome(s) ? "True" : "False") << std::endl;

return 0;

}

```

### 3. Longest Substring Without Repeating Characters

\*\*Problem:\*\* Given a string `s`, find the length of the longest substring without repeating characters.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <unordered\_map>

#include <string>

#include <algorithm>

int lengthOfLongestSubstring(std::string s) {

std::unordered\_map<char, int> charMap;

int maxLength = 0, start = 0;

for (int end = 0; end < s.length(); end++) {

if (charMap.find(s[end]) != charMap.end()) {

start = std::max(start, charMap[s[end]] + 1);

}

charMap[s[end]] = end;

maxLength = std::max(maxLength, end - start + 1);

}

return maxLength;

}

int main() {

std::string s = "abcabcbb";

std::cout << "Length of Longest Substring Without Repeating Characters: " << lengthOfLongestSubstring(s) << std::endl;

return 0;

}

```

### 4. Group Anagrams

\*\*Problem:\*\* Given an array of strings, group the anagrams together. You can return the answer in any order.

\*\*Solution:\*\*

```cpp

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_map>

#include <algorithm>

vector<vector<string>> groupAnagrams(vector<string>& strs) {

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

for (string s : strs)

{

string temp = s;

sort(temp.begin(), temp.end());

mp[temp].push\_back(s);

}

vector<vector<string>> ans;

for (auto it : mp)

{

ans.push\_back(it.second);

}

return ans;

}

int main() {

std::vector<std::string> strs = {"eat", "tea", "tan", "ate", "nat", "bat"};

std::vector<std::vector<std::string>> groupedAnagrams = groupAnagrams(strs);

std::cout << "Grouped Anagrams: " << std::endl;

for (auto& group : groupedAnagrams) {

for (std::string& s : group) {

std::cout << s << " ";

}

std::cout << std::endl;

}

return 0;

}

```

### 5. Longest Palindromic Substring

\*\*Problem:\*\* Given a string `s`, return the longest palindromic substring in `s`.

\*\*Solution:\*\* (Expand Around Center)

```cpp

#include <iostream>

#include <string>

std::string expandAroundCenter(std::string& s, int left, int right) {

while (left >= 0 && right < s.length() && s[left] == s[right]) {

left--;

right++;

}

return s.substr(left + 1, right - left - 1);

}

std::string longestPalindrome(std::string s) {

if (s.empty()) return "";

std::string longest = s.substr(0, 1);

for (int i = 0; i < s.length(); i++) {

std::string oddPal = expandAroundCenter(s, i, i);

std::string evenPal = expandAroundCenter(s, i, i + 1);

if (oddPal.length() > longest.length()) {

longest = oddPal;

}

if (evenPal.length() > longest.length()) {

longest = evenPal;

}

}

return longest;

}

int main() {

std::string s = "babad";

std::cout << "Longest Palindromic Substring: " << longestPalindrome(s) << std::endl;

return 0;

}

LINKED LIST

### 1. Reverse Linked List

\*\*Problem:\*\* Reverse a singly linked list.

\*\*Solution:\*\*

```cpp

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* reverseList(ListNode\* head) {

ListNode\* prev = nullptr;

ListNode\* curr = head;

while (curr != nullptr) {

ListNode\* nextTemp = curr->next;

curr->next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

void printList(ListNode\* head) {

while (head) {

std::cout << head->val << " ";

head = head->next;

}

std::cout << std::endl;

}

int main() {

ListNode\* head = new ListNode(1);

head->next = new ListNode(2);

head->next->next = new ListNode(3);

head->next->next->next = new ListNode(4);

head->next->next->next->next = new ListNode(5);

std::cout << "Original List: ";

printList(head);

head = reverseList(head);

std::cout << "Reversed List: ";

printList(head);

return 0;

}

```

### 2. Merge Two Sorted Lists

\*\*Problem:\*\* Merge two sorted linked lists and return it as a new sorted list.

\*\*Solution:\*\*

```cpp

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

ListNode\* dummy = new ListNode(0);

ListNode\* current = dummy;

while (l1 != nullptr && l2 != nullptr) {

if (l1->val < l2->val) {

current->next = l1;

l1 = l1->next;

} else {

current->next = l2;

l2 = l2->next;

}

current = current->next;

}

current->next = l1 != nullptr ? l1 : l2;

return dummy->next;

}

void printList(ListNode\* head) {

while (head) {

std::cout << head->val << " ";

head = head->next;

}

std::cout << std::endl;

}

int main() {

ListNode\* l1 = new ListNode(1);

l1->next = new ListNode(2);

l1->next->next = new ListNode(4);

ListNode\* l2 = new ListNode(1);

l2->next = new ListNode(3);

l2->next->next = new ListNode(4);

ListNode\* mergedList = mergeTwoLists(l1, l2);

std::cout << "Merged List: ";

printList(mergedList);

return 0;

}

```

### 3. Remove Nth Node From End of List

\*\*Problem:\*\* Given the head of a linked list, remove the nth node from the end of the list and return its head.

\*\*Solution:\*\*

```cpp

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

ListNode\* dummy = new ListNode(0);

dummy->next = head;

ListNode\* first = dummy;

ListNode\* second = dummy;

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

first = first->next;

}

while (first != nullptr) {

first = first->next;

second = second->next;

}

second->next = second->next->next;

return dummy->next;

}

void printList(ListNode\* head) {

while (head) {

std::cout << head->val << " ";

head = head->next;

}

std::cout << std::endl;

}

int main() {

ListNode\* head = new ListNode(1);

head->next = new ListNode(2);

head->next->next = new ListNode(3);

head->next->next->next = new ListNode(4);

head->next->next->next->next = new ListNode(5);

std::cout << "Original List: ";

printList(head);

head = removeNthFromEnd(head, 2);

std::cout << "List after removing 2nd node from end: ";

printList(head);

return 0;

}

```

### 4. Linked List Cycle

\*\*Problem:\*\* Given a linked list, determine if it has a cycle in it.

\*\*Solution:\*\* (Using Floyd’s Cycle Detection Algorithm)

```cpp

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

bool hasCycle(ListNode\* head) {

if (head == nullptr || head->next == nullptr) return false;

ListNode\* slow = head;

ListNode\* fast = head->next;

while (slow != fast) {

if (fast == nullptr || fast->next == nullptr) return false;

slow = slow->next;

fast = fast->next->next;

}

return true;

}

int main() {

ListNode\* head = new ListNode(3);

head->next = new ListNode(2);

head->next->next = new ListNode(0);

head->next->next->next = new ListNode(-4);

head->next->next->next->next = head->next; // Creates a cycle

std::cout << "Has Cycle: " << (hasCycle(head) ? "True" : "False") << std::endl;

return 0;

}

```

### 5. Intersection of Two Linked Lists

\*\*Problem:\*\* Write a program to find the node at which the intersection of two singly linked lists begins.

\*\*Solution:\*\*

```cpp

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* getIntersectionNode(ListNode\* headA, ListNode\* headB) {

if (headA == nullptr || headB == nullptr) return nullptr;

ListNode\* a = headA;

ListNode\* b = headB;

while (a != b) {

a = (a == nullptr) ? headB : a->next;

b = (b == nullptr) ? headA : b->next;

}

return a;

}

int main() {

ListNode\* headA = new ListNode(4);

headA->next = new ListNode(1);

ListNode\* intersection = new ListNode(8);

headA->next->next = intersection;

intersection->next = new ListNode(4);

intersection->next->next = new ListNode(5);

ListNode\* headB = new ListNode(5);

headB->next = new ListNode(6);

headB->next->next = new ListNode(1);

headB->next->next->next = intersection;

ListNode\* intersectNode = getIntersectionNode(headA, headB);

if (intersectNode) {

std::cout << "Intersection at node with value: " << intersectNode->val << std::endl;

} else {

std::cout << "No Intersection" << std::endl;

}

return 0;

}

Searching and Sorting

**1. Merge Sort**

**Problem:** Implement the merge sort algorithm.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

void merge(std::vector<int>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

std::vector<int> L(n1);

std::vector<int> R(n2);

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(std::vector<int>& arr, int left, int right) {

if (left >= right)

return;

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

int main() {

std::vector<int> arr = {12, 11, 13, 5, 6, 7};

mergeSort(arr, 0, arr.size() - 1);

std::cout << "Sorted array: ";

for (int i : arr) {

std::cout << i << " ";

}

std::cout << std::endl;

return 0;

}

**2. Quick Sort**

**Problem:** Implement the quick sort algorithm.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

int partition(std::vector<int>& arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

std::swap(arr[i], arr[j]);

}

}

std::swap(arr[i + 1], arr[high]);

return i + 1;

}

void quickSort(std::vector<int>& arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

std::vector<int> arr = {10, 7, 8, 9, 1, 5};

quickSort(arr, 0, arr.size() - 1);

std::cout << "Sorted array: ";

for (int i : arr) {

std::cout << i << " ";

}

std::cout << std::endl;

return 0;

}

**3. Binary Search**

**Problem:** Implement the binary search algorithm.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

int binarySearch(std::vector<int>& arr, int left, int right, int x) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == x)

return mid;

if (arr[mid] < x)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

int main() {

std::vector<int> arr = {2, 3, 4, 10, 40};

int x = 10;

int result = binarySearch(arr, 0, arr.size() - 1, x);

if (result != -1)

std::cout << "Element found at index " << result << std::endl;

else

std::cout << "Element not found" << std::endl;

return 0;

}

**4. Search in Rotated Sorted Array**

**Problem:** Given a sorted array that has been rotated, search for a target value.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

int search(std::vector<int>& nums, int target) {

int left = 0, right = nums.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target)

return mid;

if (nums[left] <= nums[mid]) {

if (target >= nums[left] && target < nums[mid])

right = mid - 1;

else

left = mid + 1;

} else {

if (target > nums[mid] && target <= nums[right])

left = mid + 1;

else

right = mid - 1;

}

}

return -1;

}

int main() {

std::vector<int> nums = {4, 5, 6, 7, 0, 1, 2};

int target = 0;

int result = search(nums, target);

std::cout << "Target found at index: " << result << std::endl;

return 0;

}

**5. First Bad Version**

**Problem:** Given an array of versions, where the first bad version is identified, find the first bad version.

**Solution:**

cpp

Copy code

#include <iostream>

// Assume this is provided by the system

bool isBadVersion(int version) {

return version >= 4; // Example, version 4 is the first bad version

}

int firstBadVersion(int n) {

int left = 1, right = n;

while (left < right) {

int mid = left + (right - left) / 2;

if (isBadVersion(mid))

right = mid;

else

left = mid + 1;

}

return left;

}

int main() {

int n = 5; // Example: 5 versions

int firstBad = firstBadVersion(n);

std::cout << "First Bad Version: " << firstBad << std::endl;

return 0;

}

Tree

**1. Maximum Depth of Binary Tree**

**Problem:** Given the root of a binary tree, return its maximum depth.

**Solution:**

cpp

Copy code

#include <iostream>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

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

};

int maxDepth(TreeNode\* root) {

if (root == nullptr) return 0;

int leftDepth = maxDepth(root->left);

int rightDepth = maxDepth(root->right);

return std::max(leftDepth, rightDepth) + 1;

}

int main() {

TreeNode\* root = new TreeNode(3);

root->left = new TreeNode(9);

root->right = new TreeNode(20);

root->right->left = new TreeNode(15);

root->right->right = new TreeNode(7);

std::cout << "Maximum Depth of Binary Tree: " << maxDepth(root) << std::endl;

return 0;

}

**2. Validate Binary Search Tree**

**Problem:** Determine if a given binary tree is a valid binary search tree (BST).

**Solution:**

cpp

Copy code

#include <iostream>

#include <limits>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

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

};

bool validate(TreeNode\* node, long long low, long long high) {

if (node == nullptr) return true;

if (node->val <= low || node->val >= high) return false;

return validate(node->left, low, node->val) && validate(node->right, node->val, high);

}

bool isValidBST(TreeNode\* root) {

return validate(root, std::numeric\_limits<long long>::min(), std::numeric\_limits<long long>::max());

}

int main() {

TreeNode\* root = new TreeNode(2);

root->left = new TreeNode(1);

root->right = new TreeNode(3);

std::cout << "Is Valid BST: " << (isValidBST(root) ? "True" : "False") << std::endl;

return 0;

}

**3. Symmetric Tree**

**Problem:** Check whether a given binary tree is symmetric around its center.

**Solution:**

cpp

Copy code

#include <iostream>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

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

};

bool isMirror(TreeNode\* left, TreeNode\* right) {

if (left == nullptr && right == nullptr) return true;

if (left == nullptr || right == nullptr) return false;

return (left->val == right->val) && isMirror(left->left, right->right) && isMirror(left->right, right->left);

}

bool isSymmetric(TreeNode\* root) {

if (root == nullptr) return true;

return isMirror(root->left, root->right);

}

int main() {

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(2);

root->left->left = new TreeNode(3);

root->left->right = new TreeNode(4);

root->right->left = new TreeNode(4);

root->right->right = new TreeNode(3);

std::cout << "Is Symmetric: " << (isSymmetric(root) ? "True" : "False") << std::endl;

return 0;

}

**4. Binary Tree Level Order Traversal**

**Problem:** Given the root of a binary tree, return the level order traversal of its nodes' values.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <queue>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

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

};

std::vector<std::vector<int>> levelOrder(TreeNode\* root) {

std::vector<std::vector<int>> result;

if (root == nullptr) return result;

std::queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

int levelSize = q.size();

std::vector<int> currentLevel;

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

TreeNode\* node = q.front();

q.pop();

currentLevel.push\_back(node->val);

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

result.push\_back(currentLevel);

}

return result;

}

int main() {

TreeNode\* root = new TreeNode(3);

root->left = new TreeNode(9);

root->right = new TreeNode(20);

root->right->left = new TreeNode(15);

root->right->right = new TreeNode(7);

std::vector<std::vector<int>> levels = levelOrder(root);

std::cout << "Level Order Traversal: " << std::endl;

for (const auto& level : levels) {

for (int val : level) {

std::cout << val << " ";

}

std::cout << std::endl;

}

return 0;

}

**5. Lowest Common Ancestor of a Binary Tree**

**Problem:** Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

**Solution:**

cpp

Copy code

#include <iostream>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

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

};

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if (root == nullptr || root == p || root == q) return root;

TreeNode\* left = lowestCommonAncestor(root->left, p, q);

TreeNode\* right = lowestCommonAncestor(root->right, p, q);

if (left != nullptr && right != nullptr) return root;

return left != nullptr ? left : right;

}

int main() {

TreeNode\* root = new TreeNode(3);

root->left = new TreeNode(5);

root->right = new TreeNode(1);

root->left->left = new TreeNode(6);

root->left->right = new TreeNode(2);

root->right->left = new TreeNode(0);

root->right->right = new TreeNode(8);

root->left->right->left = new TreeNode(7);

root->left->right->right = new TreeNode(4);

TreeNode\* p = root->left; // Node with value 5

TreeNode\* q = root->left->right->right; // Node with value 4

TreeNode\* lca = lowestCommonAncestor(root, p, q);

std::cout << "Lowest Common Ancestor: " << lca->val << std::endl;

return 0;

}

Graphs

**1. Number of Islands**

**Problem:** Given a 2D grid map of 1s (land) and 0s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

void dfs(std::vector<std::vector<char>>& grid, int i, int j) {

if (i < 0 || j < 0 || i >= grid.size() || j >= grid[0].size() || grid[i][j] == '0') {

return;

}

grid[i][j] = '0'; // Mark as visited

dfs(grid, i + 1, j);

dfs(grid, i - 1, j);

dfs(grid, i, j + 1);

dfs(grid, i, j - 1);

}

int numIslands(std::vector<std::vector<char>>& grid) {

if (grid.empty()) return 0;

int count = 0;

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

for (int j = 0; j < grid[0].size(); j++) {

if (grid[i][j] == '1') {

count++;

dfs(grid, i, j);

}

}

}

return count;

}

int main() {

std::vector<std::vector<char>> grid = {

{'1', '1', '0', '0', '0'},

{'1', '1', '0', '0', '0'},

{'0', '0', '1', '0', '0'},

{'0', '0', '0', '1', '1'}

};

std::cout << "Number of Islands: " << numIslands(grid) << std::endl;

return 0;

}

**2. Course Schedule**

**Problem:** There are n courses you have to take, labeled from 0 to n-1. Some courses have prerequisites. Determine if it is possible to finish all courses.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <queue>

bool canFinish(int numCourses, std::vector<std::vector<int>>& prerequisites) {

std::vector<int> inDegree(numCourses, 0);

std::vector<std::vector<int>> adj(numCourses);

for (auto& pre : prerequisites) {

adj[pre[1]].push\_back(pre[0]);

inDegree[pre[0]]++;

}

std::queue<int> q;

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

if (inDegree[i] == 0) q.push(i);

}

int count = 0;

while (!q.empty()) {

int course = q.front();

q.pop();

count++;

for (int next : adj[course]) {

if (--inDegree[next] == 0) q.push(next);

}

}

return count == numCourses;

}

int main() {

int numCourses = 2;

std::vector<std::vector<int>> prerequisites = {{1, 0}};

std::cout << "Can Finish All Courses: " << (canFinish(numCourses, prerequisites) ? "Yes" : "No") << std::endl;

return 0;

}

**3. Word Ladder**

**Problem:** Given two words (beginWord and endWord), and a dictionary's word list, find the shortest transformation sequence from beginWord to endWord using only words from the dictionary.

**Solution:**

cpp

Copy code

#include <iostream>

#include <unordered\_set>

#include <queue>

int ladderLength(std::string beginWord, std::string endWord, std::unordered\_set<std::string>& wordList) {

std::queue<std::pair<std::string, int>> q;

q.push({beginWord, 1});

while (!q.empty()) {

std::string word = q.front().first;

int length = q.front().second;

q.pop();

if (word == endWord) return length;

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

char originalChar = word[i];

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

if (word[i] == c) continue;

word[i] = c;

if (wordList.find(word) != wordList.end()) {

q.push({word, length + 1});

wordList.erase(word);

}

}

word[i] = originalChar;

}

}

return 0;

}

int main() {

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

std::string beginWord = "hit";

std::string endWord = "cog";

std::cout << "Word Ladder Length: " << ladderLength(beginWord, endWord, wordList) << std::endl;

return 0;

}

**4. Clone Graph**

**Problem:** Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph.

**Solution:**

cpp

Copy code

#include <iostream>

#include <unordered\_map>

#include <vector>

class Node {

public:

int val;

std::vector<Node\*> neighbors;

Node() : val(0), neighbors() {}

Node(int \_val) : val(\_val), neighbors() {}

Node(int \_val, std::vector<Node\*> \_neighbors) : val(\_val), neighbors(\_neighbors) {}

};

Node\* cloneGraph(Node\* node) {

if (node == nullptr) return nullptr;

std::unordered\_map<Node\*, Node\*> map;

return dfs(node, map);

}

Node\* dfs(Node\* node, std::unordered\_map<Node\*, Node\*>& map) {

if (map.find(node) != map.end()) return map[node];

Node\* clone = new Node(node->val);

map[node] = clone;

for (Node\* neighbor : node->neighbors) {

clone->neighbors.push\_back(dfs(neighbor, map));

}

return clone;

}

int main() {

Node\* node1 = new Node(1);

Node\* node2 = new Node(2);

Node\* node3 = new Node(3);

Node\* node4 = new Node(4);

node1->neighbors = {node2, node4};

node2->neighbors = {node1, node3};

node3->neighbors = {node2, node4};

node4->neighbors = {node1, node3};

Node\* clonedGraph = cloneGraph(node1);

std::cout << "Cloned Graph Node 1: " << clonedGraph->val << std::endl;

return 0;

}

**5. Network Delay Time**

**Problem:** You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges in the graph. Calculate the time it will take for all nodes to receive a signal from a given starting node k.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_map>

int networkDelayTime(std::vector<std::vector<int>>& times, int n, int k) {

std::vector<std::vector<std::pair<int, int>>> adj(n + 1);

for (auto& time : times) {

adj[time[0]].push\_back({time[1], time[2]});

}

std::vector<int> dist(n + 1, INT\_MAX);

dist[k] = 0;

using pii = std::pair<int, int>;

std::priority\_queue<pii, std::vector<pii>, std::greater<pii>> pq;

pq.push({0, k});

while (!pq.empty()) {

int d = pq.top().first;

int node = pq.top().second;

pq.pop();

if (d > dist[node]) continue;

for (auto& neighbor : adj[node]) {

int next = neighbor.first;

int time = neighbor.second;

if (dist[node] + time < dist[next]) {

dist[next] = dist[node] + time;

pq.push({dist[next], next});

}

}

}

int maxDist = \*max\_element(dist.begin() + 1, dist.end());

return maxDist == INT\_MAX ? -1 : maxDist;

}

int main() {

std::vector<std::vector<int>> times = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};

int n = 4;

int k = 2;

std::cout << "Network Delay Time: " << networkDelayTime(times, n, k) << std::endl;

return 0;

}

Dynnamic programming

**1. Fibonacci Number**

**Problem:** Find the nth Fibonacci number.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

int fib(int n) {

if (n <= 1) return n;

std::vector<int> dp(n + 1);

dp[0] = 0;

dp[1] = 1;

for (int i = 2; i < n + 1; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

int main() {

int n = 10;

std::cout << "Fibonacci Number at position " << n << " is: " << fib(n) << std::endl;

return 0;

}

**2. Climbing Stairs**

**Problem:** You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

int climbStairs(int n) {

if (n <= 1) return 1;

std::vector<int> dp(n + 1);

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

int main() {

int n = 5;

std::cout << "Number of ways to climb " << n << " stairs: " << climbStairs(n) << std::endl;

return 0;

}

**3. Longest Increasing Subsequence**

**Problem:** Given an integer array nums, return the length of the longest strictly increasing subsequence.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <algorithm>

int lengthOfLIS(std::vector<int>& nums) {

if (nums.empty()) return 0;

std::vector<int> dp(nums.size(), 1);

for (int i = 1; i < nums.size(); i++) {

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

if (nums[i] > nums[j]) {

dp[i] = std::max(dp[i], dp[j] + 1);

}

}

}

return \*std::max\_element(dp.begin(), dp.end());

}

int main() {

std::vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};

std::cout << "Length of Longest Increasing Subsequence: " << lengthOfLIS(nums) << std::endl;

return 0;

}

**4. Maximum Subarray Sum**

**Problem:** Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <algorithm>

int maxSubArray(std::vector<int>& nums) {

if (nums.empty()) return 0;

int maxSum = nums[0];

int currentSum = nums[0];

for (int i = 1; i < nums.size(); i++) {

currentSum = std::max(nums[i], currentSum + nums[i]);

maxSum = std::max(maxSum, currentSum);

}

return maxSum;

}

int main() {

std::vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

std::cout << "Maximum Subarray Sum: " << maxSubArray(nums) << std::endl;

return 0;

}

**5. Coin Change**

**Problem:** You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money. Return the fewest number of coins that you need to make up that amount.

**Solution:**

cpp

Copy code

#include <iostream>

#include <vector>

#include <algorithm>

int coinChange(std::vector<int>& coins, int amount) {

std::vector<int> dp(amount + 1, amount + 1);

dp[0] = 0;

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

for (int coin : coins) {

if (i - coin >= 0) {

dp[i] = std::min(dp[i], dp[i - coin] + 1);

}

}

}

return dp[amount] > amount ? -1 : dp[amount];

}

int main() {

std::vector<int> coins = {1, 2, 5};

int amount = 11;

std::cout << "Fewest number of coins to make " << amount << " is: " << coinChange(coins, amount) << std::endl;

return 0;

}