Overview

May 21, 2025

1 Vectors in C++

1.1 Include the vector library

```
#include <vector>
```

a) Basic Declaration and Insertion

```
vector<int> v; // []
v.push_back(1); // [1]
v.push_back(2); // [1, 2]
v.push_back(3); // [1, 2, 3]
v.pop_back(); // [1, 2]
```

b) Vector Initialization

```
vector<int> v1(5, 10);  // [10, 10, 10, 10, 10]
vector<int> v2 = {1, 2, 3, 4}; // [1, 2, 3, 4]
```

c) Iterating a Vector

```
vector<int> v2;
for (int x : v2) {
    cout << x << " ";
}
```

d) Sorting a Vector

```
sort(v2.begin(), v2.end());
```

e) 2D Vector (Matrix)

```
int rows = 3, cols = 4;
vector<vector<int>> mat(rows, vector<int>(cols, 0));
```

2 Input

Input a List

```
// 5
// 1 2 3 4 5
#include <vector>

vector<int> lst;
int n, val;
cin >> n;
for (int i = 0; i < n; i++) {
    cin >> val;
    lst.push_back(val);
}
```

Input a Graph (Adjacency Matrix)

```
// Input: number of vertices
// 3
// 0 1 0
// 1 0 1
// 0 1 0

int n;
cin >> n;

vector<vector<int>> adj(n, vector<int>(n));

// Read the adjacency matrix
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cin >> adj[i][j];
    }
}
```

Input from File

```cpp
#include <fstream>

ifstream fin("input.txt");
int x;
fin >> x;
```

3 Backtracking

Backtracking is a general algorithmic technique that involves searching through all possible configurations to solve a problem. It incrementally builds candidates and abandons a candidate as soon as it determines it cannot lead to a valid solution.

Example: Generate all permutations of a given array.

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

void backtrack(vector<int>& nums, int start) {
    if (start == nums.size()) {
        for (int n : nums) cout << n << " ";
        cout << endl;
        return;
    }
    for (int i = start; i < nums.size(); ++i) {
        swap(nums[start], nums[i]);
        backtrack(nums, start + 1);
        swap(nums[start], nums[i]); // backtrack
    }
}

int main() {
    vector<int> nums = {1, 2, 3};
    backtrack(nums, 0);
    return 0;
}
```

4 Binary Search and Extended Binary Search

Binary Search is a fast search algorithm with $O(\log n)$ complexity. It works on a sorted array by repeatedly dividing the search interval in half.

Example: Standard Binary Search

```cpp
int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1; // not found
}
```

Extension: Find the Largest Number Less Than a Given Value

```
int findLargestLessThan(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    int ans = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] < target) {
            ans = arr[mid];
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return ans;
}
```

Note: This function assumes the array is sorted in ascending order.

5 DFS to Traverse a Tree

Depth-First Search (DFS) is a recursive algorithm that explores as far down a branch as possible before backtracking. It's often used for tree traversal (preorder, inorder, postorder).

Example: Inorder Traversal of a Binary Tree

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

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

void inorder(TreeNode* root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->val << " ";
```

```
    inorder(root->right);
}

int main() {
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    inorder(root); // Output: 1 3 2
    return 0;
}
```

Other DFS Variants:

- Preorder: visit node → left → right
- Postorder: left → right → node

6 BFS to Traverse a Graph

Breadth-First Search (BFS) explores the graph level by level. It's commonly used to find the shortest path in unweighted graphs.

Example: BFS on an Undirected Graph using Adjacency List

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

void bfs(int start, vector<vector<int>>& adj, int n) {
    vector<bool> visited(n, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
```

```
}

int main() {
    int n = 5; // number of nodes (0 to 4)
    vector<vector<int>> adj(n);

    // Example graph edges
    adj[0] = {1, 2};
    adj[1] = {0, 3};
    adj[2] = {0, 4};
    adj[3] = {1};
    adj[4] = {2};

    bfs(0, adj, n); // Output: 0 1 2 3 4
    return 0;
}
```

Note: Always mark nodes as visited to avoid cycles and infinite loops.

7 Exercises

Below is a curated list of problems to strengthen your understanding of key Data Structures and Algorithms concepts. The problems are grouped by topic, with links to the problem statements on LeetCode and CSES.

Sliding Window and Binary Search

- **Minimum Size Subarray Sum**
  https://leetcode.com/problems/minimum-size-subarray-sum/
- **Search in Rotated Sorted Array**
  https://leetcode.com/problems/search-in-rotated-sorted-array/
- **Koko Eating Bananas**
  https://leetcode.com/problems/koko-eating-bananas/
- **Longest Increasing Subsequence**
  https://leetcode.com/problems/longest-increasing-subsequence/
- **Binary Tree Level Order Traversal**
  https://leetcode.com/problems/binary-tree-level-order-traversal/
- **Validate Binary Search Tree**
  https://leetcode.com/problems/validate-binary-search-tree/
- **Lowest Common Ancestor of a Binary Search Tree**
  https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/

- **Diameter of Binary Tree**
  https://leetcode.com/problems/diameter-of-binary-tree/
- **Subsets**
  https://leetcode.com/problems/subsets/
- **Combination Sum**
  https://leetcode.com/problems/combination-sum/
- **Permutations**
  https://leetcode.com/problems/permutations/
- **Number of Islands**
  https://leetcode.com/problems/number-of-islands/
- **Max Area of Island**
  https://leetcode.com/problems/max-area-of-island/
- **Course Schedule**
  https://leetcode.com/problems/course-schedule/
- **Course Schedule II**
  https://leetcode.com/problems/course-schedule-ii/
- **Subordinates**
  https://cses.fi/problemset/task/1674
- **Two Sets**
  https://cses.fi/problemset/task/1092
- **Permutations**
  https://cses.fi/problemset/task/1070

Keep Practicing After the Course!

After finishing this course, **do not stop here**! Continue to sharpen your algorithmic thinking by solving programming problems on reputable online platforms. Below are some highly recommended resources:

- **LeetCode** (`https://leetcode.com/problemset/`): The most popular platform nowadays, closely aligned with FAANG interview problems. Problems are categorized by topics (Array, Tree, Graph, etc.) and difficulty levels (Easy, Medium, Hard).
- **NeetCode Roadmap** (`https://neetcode.io/roadmap`): A curated collection of LeetCode problems organized by topics, perfect for beginners to follow a structured learning path.
- **Codeforces** (`https://codeforces.com/`): Focused on competitive programming contests. Problems here are often algorithmically challenging and great for advanced practice.
- **CSES Problem Set** (`https://cses.fi/problemset/`): Covers a wide range of topics with a well-balanced progression from easy to hard. The problem set is smaller but very high quality.
- **VNOI Wiki** (`https://wiki.vnoi.info/`): Comprehensive documentation and tutorials on various algorithmic topics, great as a reference.

Keep practicing regularly to build strong problem-solving skills that will help you excel in interviews and competitions!