
# Summer of Science 2025

CS02
Advanced Algorithms for CP

## Midterm Report

**Prepared by:** Saksham Khandelwal - 24B0965
**Instructor:** Nirav Bhattad
**Date:** June 17, 2025

# Contents


**1 STL Containers and Algorithms** **1**
- 1.1 Container Selection Guide . . . 1
- 1.2 Common Utility Algorithms and Modifiers . . . 2

**2 Sorting Algorithms** **3**
- 2.1 Introduction . . . 3
- 2.2 Selection Sort . . . 3
- 2.3 Bubble Sort . . . 3
- 2.4 Insertion Sort . . . 3
- 2.5 Merge Sort . . . 4
- 2.6 Quick Sort . . . 5
- 2.7 Counting Sort . . . 5
- 2.8 STL Sorting Algorithms . . . 6

**3 Binary Search** **7**
- 3.1 Rotated Sorted Array . . . 7
- 3.2 STL Binary Search Functions . . . 8
  - 3.2.1 Custom Data Structures . . . 8

**4 Recursion and Backtracking** **8**
- 4.1 Sudoku Solver . . . 8
- 4.2 Complexity Analysis . . . 9

**5 Greedy Algorithms** **10**

**6 Dynamic Programming** **10**
- 6.1 Matrix Chain Multiplication . . . 10

**7 Two Pointers Technique** **11**
- 7.1 Trapping Rain Water . . . 11

**8 Sliding Window Technique** **11**
- 8.1 Fixed Size Window . . . 12
- 8.2 Variable Size Window . . . 12

**9 Graph Algorithms** **13**
- 9.1 Graph Traversals . . . 13
  - 9.1.1 Breadth-First Search (BFS) . . . 13
  - 9.1.2 Depth-First Search (DFS) . . . 14
- 9.2 Topological Sorting . . . 14
- 9.3 Shortest Distance and Paths . . . 17
  - 9.3.1 Dijkstra's Algorithm . . . 17
  - 9.3.2 Bellman-Ford Algorithm . . . 18
  - 9.3.3 Floyd-Warshall Algorithm . . . 18
- 9.4 Prim's Algorithm . . . 20
- 9.5 Disjoint Set . . . 21
- 9.6 Kruskal's Algorithm . . . 23
- 9.7 Kosaraju's Algorithm . . . 23

# 1 STL Containers and Algorithms

- **vector** – Dynamic array; fast random access, slow insert/erase at middle.
- **deque** – Double-ended queue; fast insert/delete at both ends, random access supported.
- **list** – Doubly linked list; fast insert/erase anywhere, no random access.
- **stack** – LIFO container adapter; uses deque/list/vector underneath.
- **queue** – FIFO container adapter; uses deque/list underneath.
- **priority_queue** – Max-heap by default; highest element always on top.
- **set** – Balanced BST (Red-Black Tree); sorted, unique keys, O(log n) ops.
- **map** – Key-value pairs in sorted order; unique keys, implemented as BST.
- **unordered_set** – Hash table; faster average-case lookup, no order.
- **unordered_map** – Hash map; faster average-case than map, no order.
- **multiset** – Like set but allows duplicate keys.
- **multimap** – Like map but allows duplicate keys.

## 1.1 Container Selection Guide

| **Use Case** | **Recommended Container** | **Reasons** |
|---|---|---|
| Random access | vector, deque | O(1) access |
| Frequent insertion at ends | deque | O(1) push_front/back |
| Frequent insertion in middle | list | O(1) insertion |
| Sorted data | set, map | Automatic sorting |
| Duplicate keys | multiset, multimap | Allows duplicates |
| LIFO access | stack | Simple interface |
| FIFO access | queue | Simple interface |
| Priority access | priority_queue | Efficient heap ops |
| Key-value pairs | unordered_map | O(1) average access |
| Unique elements | unordered_set | O(1) average access |

Table 1: Container Selection Guide

## 1.2 Common Utility Algorithms and Modifiers

Useful algorithms and iterator-based operations available in the STL for modifying and querying container data.

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

vector<int> v = {5, 3, 1, 4, 2};

// Copy & Fill
vector<int> dest(5);
copy(v.begin(), v.end(), dest.begin());      // O(n)
fill(dest.begin(), dest.end(), 0);           // O(n)

// Remove
auto new_end = remove(v.begin(), v.end(), 3); // O(n) - moves the
    element to end and return its iterator

v.erase(new_end, v.end());                    // O(n)

// Unique
sort(v.begin(), v.end());
auto last = unique(v.begin(), v.end());       // O(n) - moves the
    consecutive duplicates to end and return its iterator

v.erase(last, v.end());                       // O(n)

// Reverse, Rotate, Partition
reverse(v.begin(), v.end());                  // O(n)
rotate(v.begin(), v.begin() + 2, v.end());    // O(n)
auto pivot = partition(v.begin(), v.end(),
    [](int x){ return x % 2 == 0; });         // O(n) - moves the
        elements that doeesnt satisfy to end and return iterator of
        first such element in new container


// Count & Accumulate
int count_3 = count(v.begin(), v.end(), 3);  // O(n)
int sum = accumulate(v.begin(), v.end(), 0);                    // O(n)
int product = accumulate(v.begin(), v.end(), 1, multiplies<int>());

// Min/Max Elements
auto min_it = min_element(v.begin(), v.end()); // O(n)
auto max_it = max_element(v.begin(), v.end()); // O(n)

// Permutations
next_permutation(v.begin(), v.end());        // O(n)
prev_permutation(v.begin(), v.end());        // O(n)
```

Listing 1: Common STL Modifiers and Utilities

# 2 Sorting Algorithms

## 2.1 Introduction

Sorting algorithms rearrange elements in a specific order (ascending/descending). **Note**: A sorting algorithm is said to be stable if it maintains the relative order of equivalent elements

## 2.2 Selection Sort

Repeatedly selects smallest element from unsorted portion and swaps it to beginning.

**Complexity**:

- Time: $O(n^2)$ all cases
- Space: O(1)
- Not stable

## 2.3 Bubble Sort

Repeatedly swaps adjacent elements if in wrong order, bubbling largest element to end.

**Complexity**:

- Best: O(n) (already sorted)
- Worst/Avg: $O(n^2)$
- Space: O(1)
- Stable

## 2.4 Insertion Sort

Builds sorted array one element at a time by inserting each element into correct position.

**Complexity**:

- Best: O(n) (already sorted)
- Worst/Avg: $O(n^2)$
- Space: O(1)
- Stable and adaptive

## 2.5 Merge Sort

Divide and merge algorithm that recursively splits array, sorts subarrays, and merges them.

```
void merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp;
    int left = low;
    int right = mid + 1;

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.push_back(arr[left]);
            left++;
        }
        else {
            temp.push_back(arr[right]);
            right++;
        }
    }
    while (left<=mid) {
        temp.push_back(arr[left]);
        left++;
    }
    while (right<=high) {
        temp.push_back(arr[right]);
        right++;
    }

    for (int i = low; i<=high; i++) {
        arr[i] = temp[i - low];
    }
}
void mergeSort(vector<int> &arr, int low, int high) {
    if (low == high) return;
    int mid = low+high / 2;
    mergeSort(arr, low, mid);
    mergeSort(arr, mid+1, high);
    merge(arr, low, mid, high);
}

mergeSort(arr, 0, n-1);
```

Listing 2: Merge Sort

**Complexity**:

- Time: O(n log n) all cases
- Space: O(n)
- Stable

## 2.6 Quick Sort

Divide and conquer algorithm that partitions array around pivot and recursively sorts partitions.

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[low], i = low, j = high;
    while (i<j) {
        while(arr[i] <= pivot && i<=high-1) i++;
        while(arr[j] > pivot && j>=low+1) j--;
        if(i<j) swap(arr[i], arr[j]);
    }
    swap(arr[low], arr[j]);
    return j;
}

void quickSort(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);
    }
}
```

Listing 3: Quick Sort (Hoare Partition)

**Complexity**:

- Best/Avg: O(n log n)
- Worst: $O(n^2)$ (poor pivot selection)
- Space: O(log n) stack space
- Not stable

## 2.7 Counting Sort

Non comparison sort for integers in limited range by counts occurrences.

```
void countingSort(vector<int>& arr) {
    if (arr.empty()) return;
    int min_val = *min_element(arr.begin(), arr.end());
    int max_val = *max_element(arr.begin(), arr.end());

    vector<int> count(max_val - min_val + 1), output(arr.size());
    for (int num : arr) count[num - min_val]++;
    for (int i = 1; i < max_val - min_val + 1; i++)
        count[i] += count[i-1];
    for (int i = arr.size()-1; i >= 0; i--) {
        output[--count[arr[i]-min_val]] = arr[i];
    }
    arr = output;
}
```

Listing 4: Counting Sort

**Complexity**:

- Time: O(n + k) where k is range
- Space: O(n + k)
- Stable

## 2.8 STL Sorting Algorithms

C++ provides versatile sorting functions in `<algorithm>` header.

**Basic Sorting**

```
#include <algorithm>
vector<int> v = {5, 3, 1, 4, 2};

// Ascending sort
sort(v.begin(), v.end());

// Descending sort
sort(v.rbegin(), v.rend());
// or
sort(v.begin(), v.end(), greater<int>());
```

Listing 5: STL sort

**Partial Sorting**

```
// Only sort first k elements
partial_sort(v.begin(), v.begin()+3, v.end());

// Copy sorted elements to new location
vector<int> result(3);
partial_sort_copy(v.begin(), v.end(), result.begin(), result.end());

// Stable Sorting
// Maintain relative order of equal elements
stable_sort(v.begin(), v.end());
```

Listing 6: Partial Sort

**Custom Sorting**

```
// Sort structs
struct Person {
    string name;
    int age;
};
vector<Person> people = {{"Alice", 25}, {"Bob", 20}};
sort(people.begin(), people.end(), [](const Person& a, const Person& b)
    {
    return a.age < b.age;
});
```

Listing 7: Custom Comparator

**STL sort characteristics**:

- Hybrid algorithm (Introsort: Quicksort + Heapsort + Insertion sort)
- Average: O(n log n)
- Worst: O(n log n)
- Space: O(1)
- Not stable by default (use `stable_sort` for stability)

# 3 Binary Search

Binary search is an efficient O(log n) algorithm for finding elements in sorted arrays by repeatedly dividing the search interval in half.

## 3.1 Rotated Sorted Array

Given a sorted array rotated at an unknown pivot (e.g., [4,5,6,7,0,1,2]), find the index of a target value or return -1 if not found.

```
int search(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;

        // Atleast one half would be sorted
        if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid])
                right = mid - 1;  // Target in left half
            else
                left = mid + 1;   // Target in right half
        }
        else {
            if (nums[mid] < target && target <= nums[right])
                left = mid + 1;   // Target in right half
            else
                right = mid - 1;  // Target in left half
        }
    }
    return -1;
}
```

Listing 8: Binary Search Application

## 3.2 STL Binary Search Functions

C++ provides built-in binary search operations in `<algorithm>`.
It works with any sorted container.

```
vector<int> v = {1,3,5,7,9};
bool exists = binary_search(v.begin(), v.end(), 5); // Returns true

auto lb = lower_bound(v.begin(), v.end(), 5);
// First position where 5 can be inserted (first >=5)

auto ub = upper_bound(v.begin(), v.end(), 5);
// First position where 5 would be after last 5 (first >5)

auto [first, last] = equal_range(v.begin(), v.end(), 5);
// first = lower_bound, last = upper_bound
```

Listing 9: STL Binary Search

### 3.2.1 Custom Data Structures

```
vector<pair<int, string>> data = {{1,"A"}, {2,"B"}, {2,"C"}};
auto it = lower_bound(data.begin(), data.end(),
    make_pair(2, ""),
    [](const auto& a, const auto& b) {
        return a.first < b.first;
    }
);
// Finds first pair with key >=2 (iterator to {2,"B"})
```

Listing 10: Custom Data Lower Bound

# 4 Recursion and Backtracking

**Recursion**: Technique where a function calls itself to solve smaller instances of the same problem.

**Backtracking**: Systematic trial-and-error approach where:

- Try a candidate solution
- If it leads to solution, return success
- If not, undo (backtrack) and try alternatives

## 4.1 Sudoku Solver

**Concept**: For each empty cell, try valid numbers and check for row, column and box, backtrack when recursive path fails and try next valid number.

**Input**: board (vector of vector of char) where element = num if filled else .

**Output**: Fill the board with the correct solution

```cpp
// Check if num can be placed at (row, col)
bool isValid(vector<vector<char>>& board, int row, int col, char num) {
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == num) return false;
        if (board[i][col] == num) return false;
    }

    int boxRow = 3 * (row/3), boxCol = 3 * (col/3);
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[boxRow+i][boxCol+j] == num)
                return false;
        }
    }
    return true;
}

// Backtracking solver
bool solve(vector<vector<char>>& board) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {

            // Find empty cell
            if (board[row][col] != '.') continue;

            // Try numbers 1-9
            for (char num = '1'; num <= '9'; num++) {
                if (isValid(board, row, col, num)) {
                    board[row][col] = num;

                    // Recursively solve rest
                    if (solve(board)) return true;

                    board[row][col] = '.';  // Backtrack
                }
            }
            return false;  // No valid number
        }
    }
    return true;  // All cells filled
}
```

Listing 11: Sudoku Solver with Backtracking

## 4.2 Complexity Analysis

- **Worst-case time**: O($9^{empty\ cells}$)
- **Average-case**: Better due to early backtracking
- **Space**: O(1) additional space (recursion stack depth = empty cells)

# 5 Greedy Algorithms

Greedy algorithms build a solution step by step, always choosing the local optimal option in the hope that this leads to a globally optimal solution.

### Jump Game (Greedy Leap)

Given an array where each element represents your maximum jump length at that position, determine if you can reach the last index.

**Greedy Insight**: Maintain the furthest index you can reach, and extend it as you iterate

```
\\ Time Complexity - O(n)
bool canJump(vector<int>& nums) {
    short max_forward = nums[0];
    for (short i{1}; i<nums.size(); i++) {
        if (max_forward == 0) return false;
        max_forward = max_forward - 1 - nums[i] > 0 ? max_forward-1 :
           nums[i];
    }
    return true;
}
```

Listing 12: Greedy Leap to End

# 6 Dynamic Programming

Solving problems by combining solutions to subproblems.

## 6.1 Matrix Chain Multiplication

Given a chain of matrices A1, to An denoted by an array of size n+1, find out the minimum number of operations to multiply these n matrices.

```
int matrixMultiplication(vector<int>& arr, int n){

    vector<vector<int>> dp(n,vector<int>(n,-1));
    for (int i = n-1; i > 0; i--) {
        for (int j = i+1; i < n; j++) {
            int minops = INT_MAX;
            for (int k = i; k < j; k++) {
                int currops = arr[i-1]*arr[k]*arr[j] + dp[i][k] + dp[k
                   +1][j];
                if (currops < minops) minops = currops;
            }
        }
        dp[i][j] = minops;
    }

    return dp[1][n-1];
}
```

Listing 13: Matrix Chain Ordering

**Complexity Analysis**

- **Time**: $O(n$ş$)$
- **Space**: $O(n$š$)$

# 7 Two Pointers Technique

In the two pointers method, two pointers are used to iterate through the array values. Both pointers can move to one direction only, which ensures that the algorithm works efficiently.

## 7.1 Trapping Rain Water

```
int trap(vector<int>& height) {
    int maxLeft = height[0];
    int maxRight = height.back();
    int water = 0, left = 1, right = height.size()-2;
    while (left <= right) {
        if (maxLeft > maxRight) {
            if (height[right] > maxRight) maxRight = height[right];
            else water += maxRight - height[right];
            right-=1;
        }
        else {
            if (height[left] > maxLeft) maxLeft = height[left];
            else water += maxLeft - height[left];
            left+=1;
        }
    }
    return water;
}
```

Listing 14: Trapping Rain Water

**Complexity Analysis**

- **Time**: $O(n)$
- **Space**: $O(1)$

# 8 Sliding Window Technique

A sliding window is a constant-size subarray that moves through the array. At each window position, we want to calculate some information about the elements inside the window.

The window here can have fixed or variable size depending on the problem statement and can be moved accordingly.

## 8.1 Fixed Size Window

Max consecutive 1s after flipping at most $k$ zeros:

```
int longestOnes(vector<int>& nums, int k) {
    int left = 0, right = 0;
    int max1 = 0;

    for (; right<nums.size(); right++) {
        if (nums[right] == 0) k--; // Use 1 flip

        while(k<0) {
            if (nums[left] == 0) k++; // Regain flip
            left++;
        }
        if (k==0) max1 = max(max1, right-left+1);
    }
    max1 = max(max1, right-left);
    return max1;
}
```

Listing 15: Max Consecutive Ones III

**Complexity Analysis**

- **Time**: $O(n)$
- **Space**: $O(1)$

## 8.2 Variable Size Window

Longest subarray with at most 2 types (Fruit Into Baskets):

```
int totalFruit(vector<int>& fruits) {
    unordered_map<int, int> count;
    int left = 0, right = 0;
    int maxfruits = 0;

    for(; right < fruits.size(); right++) {
        count[fruits[right]]++;

        while(count.size() > 2) {
            count[fruits[left]]--;
            if (count[fruits[left]] == 0) count.erase(fruits[left]);
            left++;
        }

        maxfruits = max(maxfruits, right-left+1);
    }
    return maxfruits;
}
```

Listing 16: Fruit Into Baskets

**Complexity Analysis**

- **Time**: $O(n)$
- **Space**: $O(1)$

# 9 Graph Algorithms

## 9.1 Graph Traversals

Following code return the traversal vector and only work for connected graphs.
Use inside loop checking visited for general graphs and process data during traversal according to the problem.

### 9.1.1 Breadth-First Search (BFS)

Level-order traversal for unweighted graphs:
Breadth first search is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

```
vector<int> bfs(int n, vector<int> adj[], int snode) {
    int vis[n] = {0};
    vis[snode] = 1;
    queue<int> q;
    q.push(snode);
    vector<int> bfs;

    while(!q.empty()) {
        int node = q.front();
        q.pop();
        bfs.push_back(node);

        for(auto it : adj[node]) {
            if(!vis[it]) {
                vis[it] = 1;
                q.push(it);
            }
        }
    }
    return bfs;
}
```

Listing 17: BFS

**Complexity Analysis**

- **Time**: $O(n + E)$
- **Space**: $O(n)$

### 9.1.2 Depth-First Search (DFS)

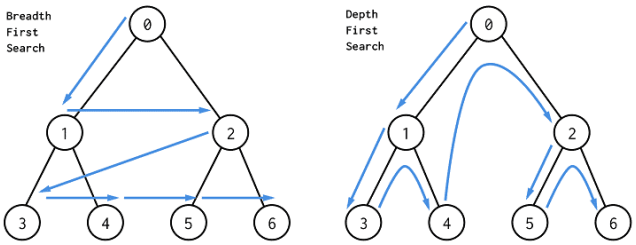
Recursive traversal for graphs:
In this method, we traverse by starting from a node, going in one direction as far as we can, and then we return and do the same on the previous nodes.

```
void dfs(int node, vector<int> adj[], int vis[], vector<int> &ls) {
    vis[node] = 1;
    ls.push_back(node);

    for (auto it : adj[node]) {
        if (!vis[it]) dfs(it, adj, vis, ls);
    }
}

vector<int> dfsgraph(int n, vector<int> adj[], int snode) {
    int vis[] = {0};
    vector<int> ls;
    dfs(snode, adj, vis, ls);

    return ls;
}
```

Listing 18: DFS

#### Complexity Analysis

- **Time**: $O(n + E)$
- **Space**: $O(n)$




BFS vs DFS

## 9.2 Topological Sorting

Ordering for directed acyclic graphs (DAGs):
A topological sort is an ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering.

```
void dfs (int node, int vis[], stack<int> &st, vector<int> adj[]) {
    vis[node] = 1;
    for (auto it : adj[node]) {
        if (!vis[it]) dfs(it, vis, st, adj);
    }
    st.push(node);
}
```

```
vector<int> topoSort(int n, vector<int> adj[])
{
    int vis[n] = {0};
    stack<int> st;

    for (int i = 0; i<n; i++) {
        if (!vis[i]) dfs(i, vis, st, adj);
    }

    vector<int> output;
    while(!st.empty()) {
        output.push_back(st.top());
        st.pop();
    }

    return output;
}
vector<int> topoSort(int n, vector<int> adj[])
{
    int indegree[n] = {0};
    for (int i = 0; i<n; i++) {
        for (auto it : adj[i]) indegree[it]++;
    }

    queue<int> q;
    for (int i = 0; i<n; i++) {
        if (indegree[i] == 0) q.push(i);
    }

    vector<int> output;
    while(!q.empty()) {
        int node = q.front();
        q.pop();
        topo.push_back(node);

        for (auto it : adj[node]) {
            if (indegree[--it] == 0) q.push(it);
        }
    }
    return output;
}
```

Listing 19: Topological Sort using BFS (Kahn Algorithm)

**Complexity Analysis**

- **Time**: $O(n + E)$
- **Space**: $O(n)$

## Word Ladder - II

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.

```
class Solution {
    unordered_map<string, int> wordstep;
    string begin;
    int wsize;
    vector<vector<string>> output;
private:
    void dfs(string word, vector<string> &curr) {
        if (word == begin) {
            output.emplace_back(curr.rbegin(), curr.rend());
            return;
        }

        int steps = wordstep[word];
        for (int i = 0; i<wsize; i++) {
            char original = word[i];
            for (char ch = 'a'; ch <= 'z'; ch++) {
                word[i] = ch;
                if (wordstep.find(word) != wordstep.end() && wordstep[word] + 1 == steps
                    ) {
                    curr.push_back(word);
                    dfs(word, curr);
                    curr.pop_back();
                }
            }
            word[i] = original;
        }
    }

public:
    vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>&
         wordList) {
        begin = beginWord;
        wsize = beginWord.size();

        unordered_set<string> wordset(wordList.begin(), wordList.end());
        queue<string> q;

        q.push(beginWord);
        wordstep[beginWord] = 1;
        wordset.erase(beginWord);

        while(!q.empty()) {
            string word = q.front();
            int steps = wordstep[word];
            q.pop();

            if (word == endWord) break;

            for (int i = 0; i<wsize; i++) {
                char original = word[i];
                for (char ch = 'a'; ch <= 'z'; ch++) {
                    word[i] = ch;
                    if (wordset.count(word)) {
                        q.push(word);
                        wordset.erase(word);
                        wordstep[word] = steps + 1;
                    }
                }
                word[i] = original;
            }
        }

        if (wordstep.find(endWord) != wordstep.end()) {
            vector<string> curr;
            curr.push_back(endWord);
            dfs(endWord, curr);
        }
        return output;
    }
};
```

Listing 20: Word Ladder Transformation

## 9.3 Shortest Distance and Paths

Finding a shortest path between two nodes of a graph is an important problem. In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find a shortest path. But for weighted graph, we use:

### 9.3.1 Dijkstra's Algorithm

To find shortest path for non-negative weights from single source:
It uses a priority queue to always expand the closest unvisited node, updating distances greedily based on current known shortest paths.

```
vector<int> dijkstra(int n, vector<pair<int, int>> graph[], int src) {

    vector<bool> processed(n, false);
    vector<int> dist(n, INT_MAX);

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair
       <int, int>>> pq;

    dist[src] = 0;
    pq.push({0, src});

    while(!pq.empty()) {
        int dis = pq.top().first;
        int node = pq.top().second;
        pq.pop();

        if (processed[node]) continue;
        processed[node] = true;

        for (auto it : graph[node]) {
            int adjNode = it.first;
            int adjWeight = it.second;

            if (dis + adjWeight < dist[adjNode]) {
                dist[adjNode] = dis + adjWeight;
                pq.push({dist[adjNode], adjNode});
            }
        }
    }
    return dist;
}
```

Listing 21: Dijkstra's Algorithm

Complexity Analysis:

- **Time**: $O((n + E)logn)$
- **Space**: $O(n)$

### 9.3.2 Bellman-Ford Algorithm

To find shortest path with negative weights from single source:
The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

***edges***: vector(node1, node2, weight) of vectors which represents the **directed** graph and can be in any order.

```
vector<int> bellman_ford(int n, vector<vector<int>>& edges, int S) {
    vector<int> dist(n, 1e8);
    dist[S] = 0;
    for (int i = 0; i < n - 1; i++) {
        for (auto it : edges) {
            int u = it[0];
            int v = it[1];
            int wt = it[2];
            if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                dist[v] = dist[u] + wt;
            }
        }
    }
    // To check negative cycle - return singleton -1 if present as algo
        completes in n-1 iteration otherwise
    for (auto it : edges) {
        int u = it[0];
        int v = it[1];
        int wt = it[2];
        if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
            return {-1};
        }
    }
    return dist;
}
```

Listing 22: Bellman-Ford

Complexity Analysis:

- **Time**: $O(n * E)$
- **Space**: $O(n)$

### 9.3.3 Floyd-Warshall Algorithm

Multi source shortest paths - Find out all pairs shortest distance:
It incrementally updates the shortest distances between every pair of nodes by checking if passing through an intermediate node offers a shorter path.

***graph***: adjacency matrix which represents the **directed** graph and if no edge present between u and v nodes then graph[i][j] = -1 else graph[i][j] = weight of edge.

```
vector<vector<int>> shortest_distance(vector<vector<int>>&graph) {
    vector<vector<int>> matrix = graph;
    int n = graph.size();

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == -1) matrix[i][j] = 1e9;
            if (i == j) matrix[i][j] = 0;
        }
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][k] < 1e9 && matrix[k][j] < 1e9) {
                    matrix[i][j] = min(matrix[i][j], matrix[i][k] +
                       matrix[k][j]);
                }
            }
        }
    }

    // For Not connected nodes, the matrix would give distance of 1e9

    // To check for negative cycles
    for (int i = 0; i < n; i++) {
        if (matrix[i][i] < 0) return {{-1}};
    }

    return matrix;
}
```

Listing 23: Floyd-Warshall

Complexity Analysis:

- **Time**: $O(n^3)$
- **Space**: $O(n^2)$

## Note: Spanning Trees

A **spanning tree** of a connected, undirected graph is a subgraph that includes all the vertices of the original graph with the minimum possible number of edges such that the subgraph is a tree. This means it is both connected and acyclic. If the original graph has $n$ vertices, every spanning tree will have exactly $n - 1$ edges.

A **minimum spanning tree (MST)** is a spanning tree where the sum of the weights of the included edges is minimized. There can be multiple MSTs for a given graph, but they will all have the same total weight.

Popular algorithms to compute MSTs include **Prim's algorithm** and **Kruskal's algorithm**, each using a greedy approach to build the minimum cost tree step-by-step.

## 9.4 Prim's Algorithm

Prim's algorithm is a greedy method for finding a minimum spanning tree by simply selecting the minimum weight edge that adds a new node to the tree.

```
pair<int, vector<pair<int, int>>> spanningTree(int n, vector<pair<int,
    int>> adj[]) {

    // Min-heap storing: {edgeWeight, {currentNode, parentNode}}
    priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int
        , int>>>, greater<pair<int, pair<int, int>>>> pq;

    vector<int> vis(n, 0);
    int sum = 0;
    vector<pair<int, int>> mst;

    pq.push({0,{0,-1}});

    while(!pq.empty()) {
        auto it = pq.top();
        pq.pop();
        int wt = it.first;
        int node = it.second.first;
        int parent = it.second.second;

        if (vis[node] == 1) continue;
        vis[node] = 1;
        sum += wt;
        if (parent != -1) mst.push_back({parent, node});

        for (auto it : adj[node]) {
            int adjNode = it.first;
            int edW = it.second;

            if (!vis[adjNode]) {
                pq.push({edW, {adjNode, node}});
            }
        }
    }

    return {sum, mst};
}
```

Listing 24: Prim's Algorithm

Complexity Analysis:

- **Time**: $O((E + n) * logE)$
- **Space**: $O(E + n)$

| Kruskal's algorithm | Prim's algorithm |
|---|---|
| Kruskal's algorithm uses a greedy strategy of choosing the minimum weight edge and adding it to the MST | Prim's algorithm uses a greedy strategy of choosing the minimum weight edge that connects a vertex in the MST to a vertex outside the MST. |
| Uses a Disjoint Set Union (DSU) data structure to keep track of the MST | Prim's algorithm uses a priority queue to keep track of the minimum weight edge. |
| Kruskal's algorithm has a time complexity of O(E log E) where E is the number of edges in the graph. | Prim's algorithm has a time complexity of O(E log V) using a binary heap or Fibonacci heap, where V is the number of vertices in the graph. |
| Guaranteed to find the MST in a connected, weighted graph, | Guaranteed to find the MST in a connected, weighted graph, but the output might be different depending on the choice of starting node for Prim's algorithm. |
| Kruskal's algorithm performs better on sparse graphs with fewer edges | Prim's algorithm performs better on dense graphs with more edges. |

Kruskal vs Prim Algorithm to find MST

## 9.5 Disjoint Set

Disjoint Set, also known as Union-Find, is a data structure that maintains a collection of disjoint sets and supports efficient operations to:

- **Find**: Determine the ultimate parent of a set containing a given element.
- **Union**: Merge two distinct sets into one.

To optimize performance, we apply:

- **Path Compression**: Flattens the tree structure during `find`, pointing each node directly to the root.
- **Union by Rank/Size**: Ensures the smaller or shallower tree is attached to the larger or deeper one, keeping the tree shallow.

**Time Complexity**: For both Find and Union operation: $O(\alpha(n))$

Here, $\alpha(n)$ is the inverse Ackermann function, which grows very slowly and is nearly constant for practical values of $n$.

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
```

Listing 25: Disjoint Set by Rank

## 9.6 Kruskal's Algorithm

This algorithm is used to find the minimum spanning by sorting all edges by weight and adding them one by one simulatenously tracking connected components using disjoint set data structure:

```
pair<int, vector<pair<int, int>>> spanningTree(int n, vector<pair<int,
    pair<int, int>>> &edges) {
    sort(edges.begin(), edges.end());
    DisjointSet ds(n);

    int mstWt = 0;
    vector<pair<int, int>> mst;

    for (auto it : edges) {
        int wt = it.first;
        int u = it.second.first;
        int v = it.second.second;

        if (ds.findUPar(u) != ds.findUPar(v)) {
            mstWt += wt;
            mst.push_back({u, v});
            ds.unionBySize(u, v);
        }
    }

    return {mstWt, mst};
}
```

Listing 26: Kruskal's Algorithm

Complexity Analysis:

- **Time**: $O(E * logE)$
- **Space**: $O(n)$

## 9.7 Kosaraju's Algorithm

In a directed graph, A component is called a Strongly Connected Component(SCC) only if for every possible pair of vertices (u, v) inside that component, u is reachable from v and v is reachable from u.
This algorithm is used to find the SCCs in a given directed graph using 2 dfs:

```
void dfs1(vector<vector<int>>& graph, int u, vector<bool>& visited,
    stack<int>& st) {
    visited[u] = true;
    for(int v : graph[u]) {
        if(!visited[v]) {
            dfs1(graph, v, visited, st);
        }
    }
    st.push(u);
}
```

```
void dfs2(vector<vector<int>>& transpose, int u, vector<bool>& visited,
    vector<int>& comp) {
    visited[u] = true;
    comp.push_back(u);
    for(int v : transpose[u]) {
        if(!visited[v]) {
            dfs2(transpose, v, visited, comp);
        }
    }
}

vector<vector<int>> kosaraju(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false);
    stack<int> st;

    // First DFS pass - for topo sort
    for(int i = 0; i < n; i++) {
        if(!visited[i]) {
            dfs1(graph, i, visited, st);
        }
    }

    // Transpose graph - reversing edges
    vector<vector<int>> transpose(n);
    for(int u = 0; u < n; u++) {
        for(int v : graph[u]) {
            transpose[v].push_back(u);
        }
    }

    fill(visited.begin(), visited.end(), false);
    vector<vector<int>> scc;

    // Second DFS pass - scc computation
    while(!st.empty()) {
        int u = st.top(); st.pop();
        if(!visited[u]) {
            vector<int> comp;
            dfs2(transpose, u, visited, comp);
            scc.push_back(comp);
        }
    }
    return scc;
}
```

Listing 27: Kosaraju's Algorithm

Complexity Analysis:

- **Time**: $O(n + E)$
- **Space**: $O(n + E)$

# List of Listings


1 Common STL Modifiers and Utilities . . . 2
2 Merge Sort . . . 4
3 Quick Sort (Hoare Partition) . . . 5
4 Counting Sort . . . 5
5 STL sort . . . 6
6 Partial Sort . . . 6
7 Custom Comparator . . . 6

---

8 Binary Search Application . . . 7
9 STL Binary Search . . . 8
10 Custom Data Lower Bound . . . 8
11 Sudoku Solver with Backtracking . . . 9
12 Greedy Leap to End . . . 10
13 Matrix Chain Ordering . . . 10

---

14 Trapping Rain Water . . . 11
15 Max Consecutive Ones III . . . 12
16 Fruit Into Baskets . . . 12

---

17 BFS . . . 13
18 DFS . . . 14
19 Topological Sort using BFS (Kahn Algorithm) . . . 15
20 Word Ladder Transformation . . . 16
21 Dijkstra's Algorithm . . . 17
22 Bellman-Ford . . . 18
23 Floyd-Warshall . . . 19
24 Prim's Algorithm . . . 20
25 Disjoint Set by Rank . . . 22
26 Kruskal's Algorithm . . . 23
27 Kosaraju's Algorithm . . . 24