Midterm Report: SoS Advanced Algorithms in CP

Y Harsha Vardhan
Mentor: Rishi Kalra

End of Week 4

1 Covered Topics with Use-Cases

Weeks 1-4 progress as per Plan of Action:

Week 1: Core Refresher

- **Sorting (Merge/Quick)**: Large dataset ordering (e.g., leaderboard rankings)
- **Binary Search**: Efficient element location in sorted arrays
- **Greedy Algorithms**: Scheduling optimization problems
- **Basic DP (Knapsack)**: Resource allocation with constraints

Weeks 2-3: Graphs & Trees

- **Dijkstra**: Shortest-path in navigation systems
- **Binary Lifting (LCA)**: Genealogy tree ancestry queries
- **SCCs (Tarjan)**: Social network community detection
- **Tree DP**: Subtree aggregation in organizational hierarchies

Week 4: Light Advanced Algorithms

- **Mo's Algorithm**: Real-time analytics on subarrays
- **KMP**: DNA sequence pattern matching

2 Solved Problems

In my GitHub Repo, I have added a list of all the problems that I have solved in Codeforces.
GitHub Repo Link: https://github.com/Y-Harsha-Vardhan/CP-SoS

For the problems that I felt are better and required more thinking, I have also presented my solutions as well as my approach towards solving those problems.
List of All Solved Problems: Solved Codeforces Problems

Following are the implementations of the above mentioned Algorithms:

Merge Sort:

Features: Divide-and-Conquer, Stable, O(nlogn)
Merge Sort recursively splits the array into halves, sorts them, and merges the sorted halves.
Implementation:

```
#include<bits/stdc++.h>
using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid-left+1;
    int n2 = right-mid;
    vector<int> L(n1), R(n2);

    for(int i=0; i<n1; i++) L[i] = arr[left+1];
    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(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);
}

void mergeSort(vector<int>& arr) {mergeSort(arr, 0, arr.size()-1);}
```

Binary Search:

There are two implementations, one is *iterative* and the other is *recursive*
Features: Average Time Complexity is O(logn), Space Complexity is O(1) for *iterative* and it is O(logn) for *recursive*.
Iterative Implementation:

```
#include<bits/stdc++.h>
using namespace std;

int binarySearch (const vector<int>& arr, int target){
    int left = 0;
    int 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;
}
```

Recursive Implementation:

```
#include<bits/stdc++.h>
using namespace std;

int binarySearch(const vector<int>& arr, int target, int left, int right){
    if (left > right) return -1;
    int mid = left + (right - left)/2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target){
    return binarySearch(arr, target, mid+1, right);}
```

```
    else {
    return binarySearch(arr, target, left, mid-1);}
}

int binarySearch(const vector<int>& arr, int target) {
    return binarySearch(arr, target, 0, arr.size() - 1);
}
```

Dijkstra Algorithm:

Problem Statement:
Given a graph and a source node, find the shortest path from the source to all other nodes.
Features:
O((V+E)logV) is the *Time Complexity* with a priority queue (where V = nodes, E = edges).
O(V) is the *Space Complexity*.
Implementation:

```
#include<bits/stdc++.h>
using namespace std;
typedef vector<vector<pair<int, int>>> Graph;

vector<int> dijkstra(const Graph& graph, int src) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        int current_dist = pq.top().first;
        pq.pop();

        if (current_dist > dist[u]) continue;

        for (const auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
```

```
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

Binary Lifting for LCA:

Problem Statement:
Given a tree (acyclic undirected graph) and multiple queries of the form (u, v), find the lowest common ancestor of nodes u and v.
Features:
Time Complexity:
- *Preprocessing:* O(NlogN)
- *LCA Query:* O(logN) per query.
Space Complexity: O(NlogN)

Implementation:

```
#include<bits/stdc++.h>
using namespace std;

class BinaryLiftingLCA {
    int n, log_max;
    vector<vector<int>> up;
    vector<int> depth;
public:
    BinaryLiftingLCA(const vector<vector<int>>& tree, int root) {
        n = tree.size();
        log_max = log2(n) + 1;
        up.assign(n, vector<int>(log_max, -1));
        depth.resize(n);
        dfs(tree, root, -1);
    }

    void dfs(const vector<vector<int>>& tree, int u, int parent) {
```

```
        up[u][0] = parent;
        for (int k=1; k<log_max; k++) {
            if (up[u][k-1] != -1) {
                up[u][k] = up[ up[u][k-1] ][k-1];}
        }
        for (int v : tree[u]) {
            if (v != parent) {
                depth[v] = depth[u] + 1;
                dfs(tree, v, u);
            }
        }
    }

    int lift(int u, int steps) {
        for (int k=0; k<log_max; k++) {
            if (steps& (1 << k)) {
                u = up[u][k];
                if (u == -1) break;
            }
        }
        return u;
    }

    int lca(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);
        u = lift(u, depth[u] - depth[v]);
        if (u == v) return u;
        for (int k=log_max-1; k>=0; k--) {
            if (up[u][k] != up[v][k]) {
                u = up[u][k];
                v = up[v][k];
            }
        }
        return up[u][0];
    }
}
```

3 Detailed Explanations

Dijkstra's Algorithm

Intuition:

Dijkstra's Algorithm finds the shortest path from a source node to all other nodes in a graph with non-negative edge weights. You can think of it like trying to drive to every city as efficiently as possible, starting from one city, and always picking the next closest place to go.

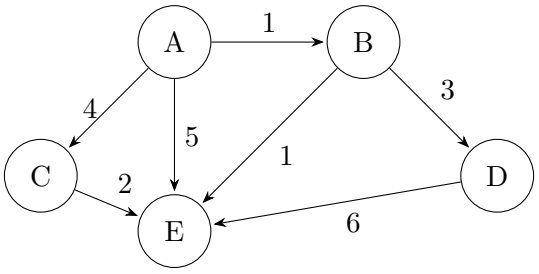
How It Works:

1. Start at the source node. Set the distance to 0.
2. All other nodes get a distance of infinity (we haven't reached them yet).
3. Put the source in a min-heap (priority queue).
4. While the queue is not empty:
    - Pick the node with the smallest distance.
    - For each neighbor, relax the edge: if we found a shorter way to it, update its distance and push it into the queue.

Use-Cases:

- **GP Navigation:** Google Maps uses variants of this to suggest shortest routes.
- **Network Routing:** Data packets are sent via the least-cost path.
- **Game AI:** Finding optimal path on maps.

Example Graph:
Dijkstra's Algorithm from Node A



Source Node: A

We run Dijkstra's Algorithm starting from node A. The table below shows the shortest distances computed from A to all other nodes.

Node	Distance from A
A	0
B	1
C	4
D	4
E	2

Shortest Paths:

- A $\rightarrow$ B (1)
- A $\rightarrow$ B $\rightarrow$ D $(1 + 3 = 4)$
- A $\rightarrow$ C (4)
- A $\rightarrow$ B $\rightarrow$ E $(1 + 1 = 2)$

Binary Lifting

Intuition:

Binary Lifting is a method to quickly jump up the tree by powers of 2. It's used to answer Lowest Common Ancestor (LCA) queries in logarithmic time after preprocessing.
Imagine a tree as a family tree. To find out how two people are related (like their common ancestor), we want to go up efficiently. Instead of climbing one level at a time, we take power-of-two jumps.

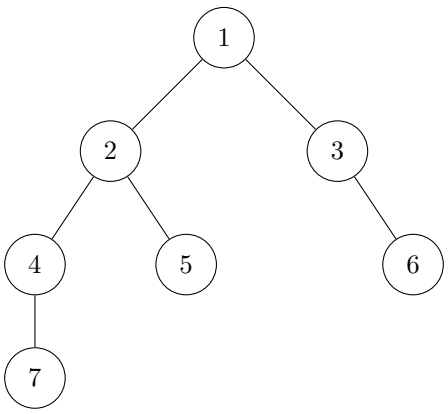
How It Works:

1. Precompute up[node][j] which stores the 2^j-th ancestor of each node.
2. To move k steps up from a node, break k into powers of 2, and jump accordingly.
3. TO find LCA:
    - Equalize depth.
    - Move both nodes up in sync using powers of 2 until they meet.

Use-Cases:

- **LCA Queries:** Frequently used in trees (e.g. in competitive programming).
- **Path Queries:** Jumping in logarithmic time.
- **Range ancestor queries** in biology or taxonomy trees.

Example:
LCA Queries with Visualization



Binary Tree Used for Binary Lifting

Binary Lifting Table (`up[node][j]` gives the 2^j-th ancestor):

Node	up[0]	up[1]	up[2]
1	–	–	–
2	1	–	–
3	1	–	–
4	2	1	–
5	2	1	–
6	3	1	–
7	4	2	1

Query: LCA(7, 5)

- Node 7 is deeper than 5. Lift 7 up until both are at the same level:
  - $7 \rightarrow$ `up[0]` $= 4$
  - $4 \rightarrow$ `up[0]` $= 2$
- Now both are at node 2 (and 5's parent is also 2):
  - $\Rightarrow$ **LCA = 2**

4 Progress Assessment

- **Strengths**: Mastered graph fundamentals (Dijkstra, SCCs), Mo's Algorithm, and KMP
- **Areas for Improvement**: Tree DP implementation speed
- **Rating Progress**: **unrated** $\rightarrow$ **1200+** (Codeforces)