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.

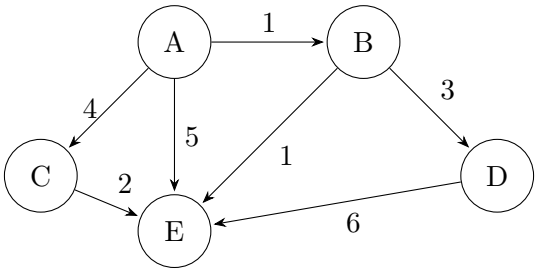
How It Works:

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.

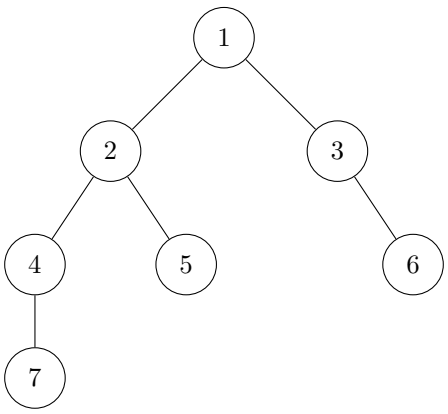
How It Works:

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)

Endterm Report: SoS Advanced Algorithms in CP

Y Harsha Vardhan
Mentor: Rishi Kalra

End of Week 8

1 Covered Topics with Use-Cases

Weeks 5-8 progress as per Plan of Action:

Week 5: Fenwick & Segment Trees

- **Point/Range queries**: Efficiently handle point updates and range queries like sum, min, or max.
- **Lazy Propagation**: Optimize range updates without traversing all affected nodes.
- **Persistent ST**: Track multiple versions of data to answer historical range queries efficiently.

Weeks 6: FFT/NTT

- **Polynomial Multiplication**: Multiply large polynomials or integers in O(nlogn) time.
- **FFT/NTT applications**: Perform fast convolutions for problems involving patterns, subsets, or frequency combinations.

Week 7: String Algorithms

- **Suffix Array (Kasai's)**: Lexicographically sort suffixes for fast substring and pattern matching queries.
- **Ukkonen's Algorithm (Suffix Tree)**: Build a suffix tree online in linear time for powerful substring operations and LZ-style matching.

Week 8: Boss Algorithms (Self-Chosen)

- **Heavy-Light Decomposition**: Reduce path queries on trees to range queries using segment trees.
- **Centroid Decomp**: Divide a tree to solve path-based problems using divide-and-conquer strategies.
- **Li Chao Tree**: Maintain and query a dynamic set of linear functions to solve convex DP problems efficiently.

2 Solved Problems

In my GitHub Repo, I have added an additional list of the problems that I have solved after the Midterm 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

3 Detailed Explanations of Boss Algorithms:

Heavy-Light Decomposition (HLD)

Intuition:

Tree path queries (e.g., sum, max between two nodes) are hard to answer efficiently because trees don't support direct indexing like arrays.
The idea is to decompose the tree into "heavy" paths (those with large subtrees) and "light" edges to flatten it into manageable chunks.

How It Works:

1. `DFS Phase`: Compute subtree sizes and designate one child as the "heavy" child (the one with the largest subtree).
2. `Decomposition Phase`:
   - Start new paths when moving through a light edge; continue the current path through heavy edges.
   - Each node is assigned a position in a segment tree, so the tree becomes linear in terms of these positions.
3. `Query/Update Phase`: For queries between two nodes, climb up the tree in chunks (from current node to top of current heavy path), reducing the query range to segments.

Use-Cases:

- Efficient path queries in trees (sum, min, max, xor)
- Online or dynamic tree operations
- Used in combination with segment trees or BITS

Code Implementation:

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

const int MAXN = 100010;
```

```
vector<int> graph[MAXN];  // Adjacency list for tree
int parent[MAXN];         // Parent of each node
int depth[MAXN];          // Depth from root
int size[MAXN];           // Subtree size
int heavy[MAXN];          // Heavy child (-1 if none)
int head[MAXN];           // Head of chain
int pos[MAXN];            // Position in segment tree
int cur_pos = 0;          // Current position counter

// First DFS: Computing subtree sizes and heavy children
void dfs1(int u, int p) {
    size[u] = 1;
    heavy[u] = -1;
    parent[u] = p;
    depth[u] = (p == -1) ? 0 : depth[p] + 1;

    int max_size = 0;
    for (int v : graph[u]) {
        if (v == p) continue;
        dfs1(v, u);
        size[u] += size[v];
        if (size[v] > max_size) {
            max_size = size[v];
            heavy[u] = v;
        }
    }
}

// Second DFS: Decomposing the tree into chains
void dfs2(int u, int h) {
    head[u] = h;
    pos[u] = cur_pos++;

    // Heavy child first (extends chain)
    if (heavy[u] != -1)
        dfs2(heavy[u], h);

    // Light children start new chains
    for (int v : graph[u]) {
        if (v == parent[u] || v == heavy[u]) continue;
        dfs2(v, v);
    }
}

// Initializing HLD with root node
void init_hld(int root) {
    cur_pos = 0;
    memset(heavy, -1, sizeof(heavy));
    dfs1(root, -1);
    dfs2(root, root);
}
```

```
// Example query function using segment tree
int query_path(int u, int v, vector<int>& seg_tree) {
    int res = 0;  // Needs to be modified based on operation (max/sum/etc)
    while (head[u] != head[v]) {
        if (depth[head[u]] < depth[head[v]])
            swap(u, v);

        // Query segment from head[u] to u
        int l = pos[head[u]], r = pos[u];
        // Combining with segment tree result
        // res = combine(res, seg_tree_query(l, r));
        u = parent[head[u]];  // Moving to next chain
    }

    // Final chain segment
    if (depth[u] > depth[v]) swap(u, v);
    int l = pos[u], r = pos[v];
    // res = combine(res, seg_tree_query(l, r));
    return res;
}
```

Key Features:

- Partitions tree into chains for efficient path queries
- Uses two DFS passes to compute chains, and chains are processed using segment trees
- Time Complexity: `O(nlogn)` for preprocessing, and $O(\log^2 n)$ per query.

Example Graph:

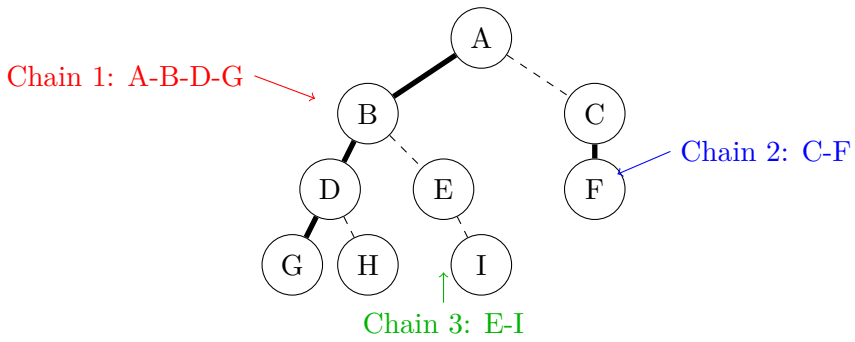


Figure 1: HLD decomposes tree into chains. Heavy edges form chains, light edges connect chains.

Query Path G→F:

1. G→D→B→A (Chain 1)
2. A→C→F (Chain 2)

Centroid Decomposition

Intuition:

Many hard tree problems (e.g., counting paths, dynamic queries) because easier when we break the tree into balanced subtrees. The centroid of a tree is a node whose removal breaks the tree into components of size $\leq$ n/2.

How It Works:

1. `Centroid Finding`: We use DFS to compute sizes and find the centroid of the tree (or subtree).
2. `Decomposition`: Mark the centroid as "processed", then recursively decompose each remaining subtree.
3. `Processing`: At each centroid, solve the problem involving paths through that node (e.g., count valid paths), possibly using prefix sums or frequency tables.

Use-Cases:

- Count number of paths of a certain type (e.g., paths with distance $\leq$ k)
- Tree diameter queries
- Optimizing brute force DFS by reducing overlapping computations

Code Implementation:

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

const int MAXN = 100010;
vector<int> graph[MAXN];        // Original tree
vector<int> centroid_tree[MAXN]; // Centroid decomposition tree
int size[MAXN];                 // Subtree size for centroid
int centroid_parent[MAXN];      // Parent in centroid tree
bool visited[MAXN];             // For tracking processed nodes

// Computing subtree sizes
void dfs_centroid(int u, int p) {
    size[u] = 1;
    for (int v : graph[u]) {
        if (v == p || visited[v]) continue;
        dfs_centroid(v, u);
        size[u] += size[v];
    }
}

// Find centroid in subtree
int find_centroid(int u, int p, int total_nodes) {
    for (int v : graph[u]) {
```

```
            if (v == p || visited[v]) continue;
            if (size[v] > total_nodes / 2)
                return find_centroid(v, u, total_nodes);
        }
        return u;  // u is the centroid
    }

    // Building centroid decomposition
    int decompose(int u) {
        dfs_centroid(u, -1);
        int centroid = find_centroid(u, -1, size[u]);
        visited[centroid] = true;

        for (int v : graph[centroid]) {
            if (visited[v]) continue;
            int sub_centroid = decompose(v);
            centroid_tree[centroid].push_back(sub_centroid);
            centroid_parent[sub_centroid] = centroid;
        }
        return centroid;
    }

    // Initializing centroid decomposition
    void init_centroid(int root, int n) {
        memset(visited, false, sizeof(visited));
        memset(centroid_parent, -1, sizeof(centroid_parent));
        decompose(root);
    }
```

Key Features:

- Recursively decomposing tree into centroids and creating a balanced decomposition tree
- Enabling efficient divide-and-conquer on trees
- Time Complexity: `O(nlogn)` for decomposition.

Example Graph:

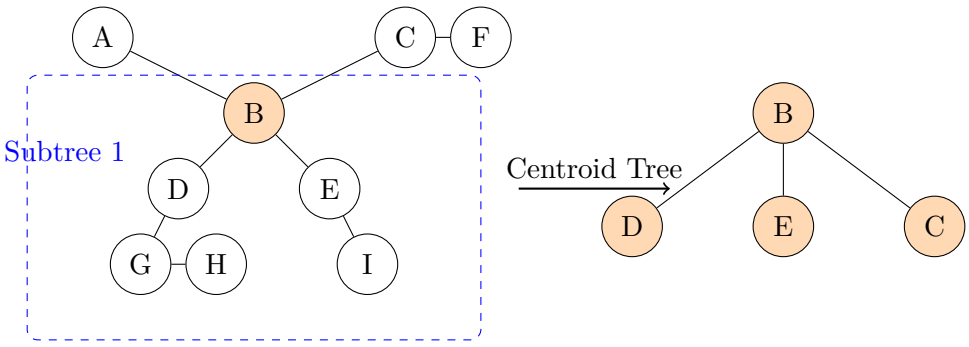


Figure 2: Centroid decomposition: (Left) Original tree with centroid B, (Right) Resulting centroid tree

Li Chao Tree

Intuition:

When you want to maintain a dynamic set of linear functions `y = mx + c` and quickly find the minmum (or maximum) value for a given x, a naive linear scan is too slow. The Li Chao Tree efficiently stores lines in a segment-tree-like structure.

How It Works:

1. `Insertion`:
   - Each node in the tree represents an interval of x-values
   - When inserting a line, you compare it to the currently stored line at relevant intervals and decide which line to keep and whether to recurse left or right
2. `Query`: Traverse the tree top-down to find the best line at a given x.
3. `Optimization`: Works best when x-values are bound or discrete.

Use-Cases:

- Convex hull trick problems in DP
- Dynamic Programming with cost functions like: $dp[i] = \min_{j<i} (dp[j] + a[j] \cdot x[i] + c)$
- Problems involving line envelopes or piecewise linear optimization

Code Implementation:

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

typedef long long ll;
const ll INF = LLONG_MAX;

struct Line {
    ll slope, intercept;
    ll evaluate(ll x) const {
        return slope * x + intercept;
    }
};

class LiChaoTree {
private:
    vector<Line> tree;  // Segment tree for lines
    vector<ll> xs;      // To coordinate compression
    int n;              // Size of compressed space

    // Recursive update: Adding line to appropriate segment
    void update(int idx, int l, int r, Line new_line) {
        if (l == r) {
```

```
            if (new_line.evaluate(xs[l]) < tree[idx].evaluate(xs[l]))
                tree[idx] = new_line;
            return;
        }

        int mid = (l + r) / 2;
        ll curr_mid = xs[mid];
        bool left_better = new_line.evaluate(xs[l]) < tree[idx].evaluate(xs[l]);
        bool mid_better = new_line.evaluate(curr_mid) < tree[idx].evaluate(curr_mid);

        if (mid_better)
            swap(tree[idx], new_line);

        if (left_better != mid_better)
            update(2 * idx, l, mid, new_line);
        else
            update(2 * idx + 1, mid + 1, r, new_line);
    }

    // Minimum value of Query at position x
    ll query(int idx, int l, int r, ll x) const {
        if (l == r)
            return tree[idx].evaluate(x);

        int mid = (l + r) / 2;
        ll res = tree[idx].evaluate(x);
        if (x <= xs[mid])
            res = min(res, query(2 * idx, l, mid, x));
        else
            res = min(res, query(2 * idx + 1, mid + 1, r, x));
        return res;
    }

public:
    // Initializing with query points (x-coordinates)
    LiChaoTree(const vector<ll>& query_points) {
        xs = query_points;
        sort(xs.begin(), xs.end());
        xs.erase(unique(xs.begin(), xs.end()), xs.end());
        n = xs.size();
        tree.resize(4 * n, {0, INF});  // Neutral line
    }
    // Adding line y = slope*x + intercept
    void add_line(ll slope, ll intercept) {
        update(1, 0, n - 1, {slope, intercept});
    }
    // Query minimum value at x
    ll query(ll x) const {
        return query(1, 0, n - 1, x);
    }
};
```

Key Features:

- Maintains set of Linear functions
- Answers point minimum queries efficiently
- Uses segment tree over compressed x-coordinates
- Time Complexity: `O(logn)` per insertion/query.

Example Graph:

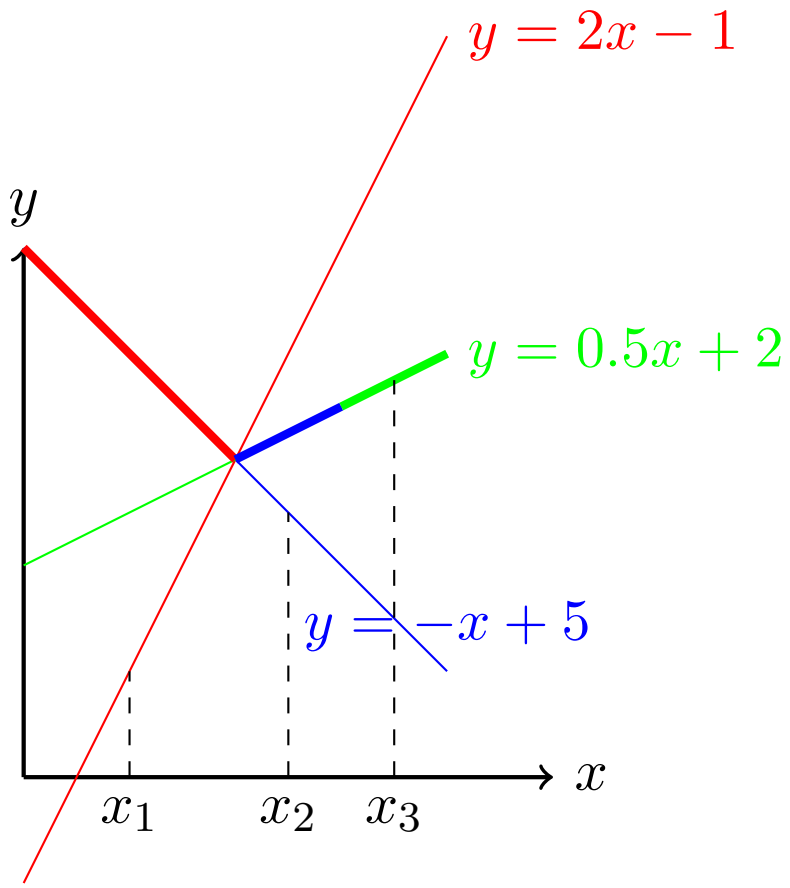
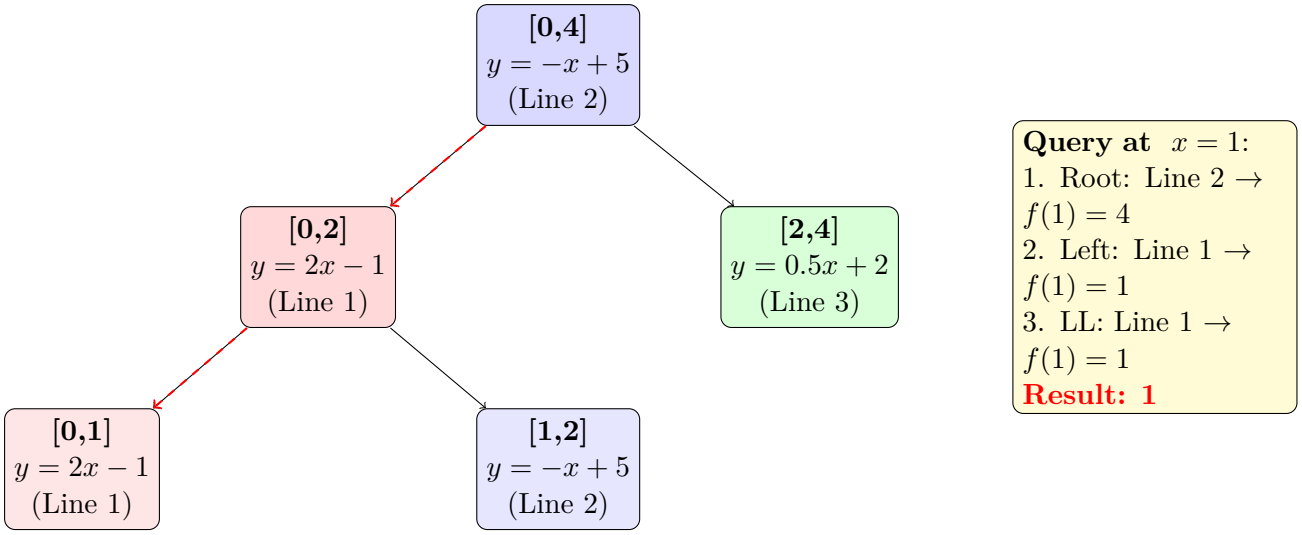


Figure 3: Compact Li Chao Tree: (Top) Lines with bold segments showing dominance regions (Bottom) Segment tree storing dominant lines per interval



4 Progress Assessment

- **Strengths**:
  - Mastered Segment Trees (Lazy Propagation)
  - FFT/NTT polynomial multiplication
  - Suffix Array construction
- **Areas for Improvement**:
  - Persistent Segment Tree implementation efficiency
  - Optimizing FFT/NTT for large convolution problems
  - Practical application of Ukkonen's Algorithm

5 Final Thoughts and Signing Off

I believe that this course has been a very informative and enjoyable way to spend my summer this year. I first became interested in Competitive Programming during my second semester. I have thoroughly enjoyed working on this course because it allowed me to approach problem-solving in a structured and efficient manner, exposing me to various types of problems.
This journey has been a valuable and significant step toward mastering CP.
So, yeah...
It was amazing — thank you!

Signing off,
Yours sincerely,
Y Harsha Vardhan
24b1069