Theoretical Assignment 2

Gurpreet Singh - 150259 . Nikita Awasthi - 150453

Hard Version

Question 1

Algorithm

We make use of layers of Convex hulls to solve this problem.

Data Structure:
Given n points, after sorting according to their x-coordinates, we compute the convex hull for the set of points. We then remove all the points present in the convex hull and recompute convex hull for the remaining points. We do this recursively until there are no points remaining.

In order to store a convex hull, we make use of two arrays -

- Array 1 stores all points in the upper half of the convex hull, i.e. points starting from the point having least x-coordinate and going clockwise until we observe the point having the highest x-coordinate (Do not include this point)
- Array 2 stores all points in the lower half of the convex hull, i.e. points starting from the point having least x-coordinate (not including this) and going anti-clockwise until we observe the point having the highest x-coordinate

Both these arrays are sorted in the increasing order of the x-coordinates of the points.

Note: In case we have multiple points remaining that lie on one line (Even if only one or two points), we consider all these points as one convex hull, and in this case, we need to store only one array, with all points in it sorted in increasing order of the x-coordinates of the points.

Algorithm for each query:

We first initialize a result list as null, and find all points that lie above the line and add those points to this list.

For each convex hull, we have at most two arrays. For each array, we need to find out the points that lie above the given line. If the line is $y - mx - c = 0$, we need to find all the points that lie above this line, i.e. for point (x_0, y_0), $y_0 - mx_0 - c > 0$.

In order to find all the points in this, we just need to find the first point which gives negative value for this, and report all the points before this point (In case m or slope is negative, then we check for the last point that gives negative value, and report all points after it). Hence, we simply binary search for the first point that gives negative value, say the index of this point is i, then we report all the points from index 0 to $i - 1$. If $i = 0$, we do not add any point (and vice versa for negative m).

After performing this operation for all (at most two) the arrays of this convex hull, we will have a list of, say t points, which lie above the given line. If $t = 0$, we stop the operation and report all the points observed until now *, else we add all these points from this list into our result list, and continue these set of operations for the inner convex hull, i.e. convex hull formed after removing all points from all previous convex hull.

* We can stop the iteration, as if all points in a convex hull lie below a line, it is obvious to see that all points inside that hull will also lie below the line.

Algorithm for Binary Search:

Considering the case for positive m
Take the middle point, if this point lies above the line, recursively call the second half of the array, excluding the middle point. Else, call the first half of the array, including the middle point. Report the final middle point observed, i.e. when the length of the array is one.
For the case of negative m
Take the middle point, if this point lies above the line, recursively call the first half of the array, excluding this point, else call the second half of the array, including this point. Return middle point when length is 1.

Proof of Correctness

Claim: All the points in the set of points are included in the layers of these convex hulls
Proof: It was discussed in class that there is a unique convex hull for a given set of n points. Our function for computation of the layers of convex hull terminates only when the set has no points left. At every step, the points included in the convex hull are removed from the set of points. Therefore, the layer of convex hulls cover all the points.

Claim: The Binary Search method reports all points above the line
Proof:
This claim is true only because the set of points are part of a convex hull.
On joining the adjacent points in one part of the convex hull, the line can intersect one of these lines or points only once. If this is not true, i.e. the line is intersecting at more than one points, then our original claim that the set of points are part of a convex hull fail. To visualize this, consider the upper half. Since the slopes of the lines of the hull need to keep decreasing, as it is a **convex** hull, hence if the line intersects at some point, all the points on the right will be below them only. (opposite for negative slope)

Since all the points are being processed, and for each iteration, we are reporting only those points which lie above the line, we can say our algorithm is correct.

Analysis

Space

Since each point is being stored only once, we can say that we are using $O(n)$ extra space.

Time

Every layer of convex hull is stored in two arrays. It is obvious that the size of these arrays is less than n, the total number of points.
For each such array, we are reporting all the points in the array in $O(log(l))$ where l is the length of the array. Therefore, we are finding all the correct points for one convex hull in $O(log(n))$ time.

Let us assume that there are k points which lie above the line. Since we are stopping the iteration if no point is found in the iteration, hence we can have at most k iterations. For each iteration, we take $O(log(n))$ time and hence in total, we will take only $O(klog(n))$ time.
Suppose reporting each point is $O(1)$, then our total time complexity will be $O(klog(n) + k)$ which is again $O(klog(n))$

Question 2

We can construct a Binary BST such that each node contains the minimum value, sum, size and increment over all nodes under it i.e. all nodes in the BST formed with this node as root.

Algorithm 1 Functions to update the BST

```
procedure updateMin(root)
    root → min = root → val
    if root → left ≠ NULL then
        root → min = min(root → left → min + root → left → inc, root → min)
    end if
    if root → right ≠ NULL then
        root → min = min(root → right → min + root → right → inc, root → min)
    end if
end procedure

procedure updateSize(root)
    root → size = 1
    if root → left ≠ NULL then
        root → size = root → size + root → left → size
    end if
    if root → right ≠ NULL then
        root → size = root → size + root → right → size
    end if
end procedure

procedure augmentedLeftRotate(root)          ▷ Code for Right Rotate will be similar
    child = root → right
    newRoot = leftRotate(root)

    updateMin(root)
    updateSize(root)

    updateMin(child)
    updateSize(child)
    return newRoot
end procedure
```

Algorithm 2 Functions to update the BST

```
procedure INSERT(root, el, i)
    if root = NULL then
        root = NEWNODE(el, el)
    else
        root → size = root → size + 1

        if root → left = NULL then
            sizeNodes = 1
        else
            sizeNodes = root → left → size + 1
        end if

        el = el − root → inc

        if i < root → left → size + 1 then
            root → left = INSERT(root → left, el, i)
        else
            root → right = INSERT(root → right, el, i − sizeNodes)
        end if

        root = BALANCE(root) ▷ Balances the tree at current node while maintaining the min, inc, sum and size values

        UPDATEMIN(root)
    end if
    return root
end procedure
```

Since we've already dealt with Balancing (Rotate), for deletion, we just need to decrement the size of all nodes in the path observed from root to the node to be deleted (after switching with the right leaf node) by 1 and also updating all these points' *min*, starting from the lowest point up to the root.

Algorithm 3 Multi-Add

```
procedure MULTIADD(root, inc, i, j)
    u = REPORT(root, i)                          ▷ Report Function discussed in class
    v = REPORT(root, j)

    w = LCA(u, v)

    if u ≠ w then
        u → val = u → val + inc
        if u → right ≠ NULL then
            u → right → inc = u → right → inc + inc
        end if
        while u → parent ≠ w do
            if u → parent → left = u then
                if u → parent → right ≠ NULL then
                    u → parent → right → inc = u → parent → right → inc + inc
                end if
                u → parent → val = u → parent → val + inc
            end if
            UPDATEMIN(u)
            u = u → parent
        end while
    end if

    if v ≠ w then
        v → val = v → val + inc
        if v → left ≠ NULL then
            v → left → inc = v → left → inc + inc
        end if
        while v → parent ≠ w do
            if v → parent → right = v then
                if v → parent → left ≠ NULL then
                    v → parent → left → inc = v → parent → left → inc + inc
                end if
                v → parent → val = u → parent → val + inc
            end if
            UPDATEMIN(v)
            v = v → parent
        end while
    end if

    w → val = w → val + inc
    UPDATEMIN(w)
end procedure
```

Algorithm 4 ReportMin

```
procedure ReportMin(root, i, j)
    u = report(root, i)
    v = report(root, j)

    w = LCA(u, v)

    minValLeft = Infinite
    minValRight = Infinite

    if u ≠ w then
        minValLeft = min(minValLeft, u → val)
        if v → right ≠ NULL then
            minValLeft = min(minValLeft, u → right → min + u → right → inc)
        end if
        minValLeft = minValLeft + u → inc

        while u → parent ≠ w do
            if u → parent → left = u then
                if u → parent → right ≠ NULL then
                    minValLeft = min(minValLeft, u → parent → right → min + u → parent → right → inc)
                end if
                minValLeft = min(minValLeft, u → parent → val)
            end if
            u = u → parent
            minValLeft = minValLeft + u → inc
        end while
    end if

    if v ≠ w then
        minValRight = min(minValRight, v → val)
        if v → left ≠ NULL then
            minValRight = min(minValRight, v → left → min + v → left → inc)
        end if
        minValRight = minValRight + v → inc

        while v → parent ≠ w do
            if v → parent → right = v then
                if v → parent → left ≠ NULL then
                    minValRight = min(minValRight, v → parent → left → min + v → parent → left → inc)
                end if
                minValRight = min(minValRight, v → parent → val)
            end if
            v = v → parent
            minValRight = minValRight + v → inc
        end while
    end if

    minVal = w → val
    minVal = min(minVal, minValLeft, minValRight)

    while w ≠ root do
        minVal = minVal + w → inc
        w = w → parent
    end while

    return minVal
end procedure
```

Proof of Correctness

Claim: The multi add function adds a value to all elements between indices i and j and no other index
Proof:
As explained in class, when performing range operations, the values that are affected are the ones which lie to the right when traversing from LCA to left extreme and those which lie to the left when traversing from LCA to right extreme. The *inc* field maintains the increment and when required is propagated to the nodes that need it.

Claim: The report min function gives the min value between the given indices
Proof:
For min range query, we attempt to look at all those nodes which lie between the two extremes. Any given node is considered in the range in the case when it is the right child of the the node which is the parent of node which is traversed from the path of the extreme left to LCA. A node on the path is only considered when its child on the path is a right child. This is done for $minValLeft$. Similar thing is done for the path from rightmost end of query to lca except that right is replaced by left. This ensures that the indices in consideration are within the range. For any given node, the *inc* factor is also added to the *min* of the node to ensure that the value considered incorporates $Multi - Add$. Thus at any point, we consider the running min value and update the value considering the *inc* factor. We also increment all the increment values observed from root to the LCA. Therefore, the $ReportMin$ always reports the current minimum value in the given range including *inc* factors.

Claim: Insertion and deletion maintain the structure of our augmented BST
Proof:
In the insertion function, the element to be inserted is decremented by the *inc* value of the nodes in the path to maintain the structure for subsequent propagation of *inc*. Similarly, before deleting the *inc* value is propagated to its children to preven loss of *inc* value. The $updateMin$ function maintains the min value.

Time Complexity

Insert function: In the insert function, we basically traverse through the depth of the BST to find the index. While traversing the path, we keep track of the *inc* values in the path. The $updateMin$ function checks for the left and the right subtree so that requires only $O(1)$. The traversal for finding index requires $O(logn)$. So insertion has worst case time-complexity of $O(logn)$.

Delete function: For the delete function, we make use of the augmented rotations. The rotate functions make use of the $updateMin$ and $updateSize$ functions which take $O(1)$ time to look at the node and its children. So, the only task is to traverse the path from the node to be deleted till the root and change the size and min value along the path. The height from the node to the root is $logn$. So finding the node and updating the values after deletion would be $O(logn)$.

Multi-Add: Given the two nodes, we find the least common ancestor of the two nodes. The node which is at a greater depth first traverses upwards so that both are at the same depth and then both of them traverse till the node is same. Since, the depth of tree is $logn$, this step takes $O(logn)$. Now we traverse the nodes from u and v, updating the nodes which are on the left and right of theri parent respectively. The traversal thus takes $O(logn)$.

ReportMin: As was the case with multi add, we traverse from the two nodes to their LCA, for u looking at the nodes on the path and their right child and for v looking at the nodes on the path and their left child giving a $O(logn)$ time complexity.

Question 3

Let T_1 be a tree such that T_1 is consistent with G.
Claim: $\exists$ a tree T_2 such that T_2 is a binary tree (i.e. no node has more than two subtrees hanging from it) and is equivalent to T_1 and hence, is consistent with G.
Note: Here equivalence means distance between v_i and v_j in T_1 and T_2 is equal $\forall\ v_i, v_j \in V$
Proof:
For any tree T_1 (that is consistent with G), consider the lowest node with number of subtrees > 2. Let this node be u.
Consider the subtrees of u, say $T_{S_1}, T_{S_2} ... T_{S_k}$ for some arbitrary $k > 2$. Consider another tree $\bar{T}$ which is same as T_1 except at u, where the subtrees $T_{S_{k-1}}$ and T_{S_k} are removed from u, joined by another node v, such that $val(v) = val(u)$ and v is hanged from u.
It is straight forward to see that the number of subtrees hanging from u have been reduced by 1, and the value of the LCA of no two leaf nodes has changed (as value of v = value of u). We can do this recursively until u has only two subtrees hanging from it. This whole process can be recursively repeated until the complete tree is binary.
Call this binary tree formed as T_2. Therefore T_2 is equivalent to T_1, and since the distance of no two leaf nodes has changed, T_2 is consistent with G.

Also, any node with degree $\neq 2$, can be removed as it cannot be the LCA of any two leaves, and is clearly redundant.
This suggests that we can work with full binary trees and ignore all other trees.

Part A

Note: We can always assume there exists a tree which is consistent with G, by setting all the nodes of that tree to 0.

Let the lowest mapping from (V, V) to R^+ is given by (u, v), i.e. $\arg\min_{v1,v2}\{d(v_1, v_2)\} = (u, v)$. Consider a full binary tree T, such that T is consistent with G, and u and v are not siblings in T.

Let $w = LCA(u, v)$. Consider the subtree with w as node, say T'.
Switching any leaves in the tree with w as node does not change the distance between any leaf node in this tree (x) and any leaf node in T (say y), not in this subtree, as the $LCA(x, y)$ remains the same, and hence the distance remains the same.

Now, we construct another tree $\bar{T}$, which is the same as the tree T, except at the tree at w, where the leaf node v is exchanged with the sibling subtree of u. Since the tree is complete, we know u will have a sibling subtree.

Claim: Distances of all leaf nodes in T will be less than those of $\bar{T}$.
Proof:
As discussed earlier, we need only compare the trees rooted at w (The LCA of u and v in T). Let the corresponding rooted trees for T and $\bar{T}$ be T' and $\bar{T}'$ respectively.

Since T is consistent, the node values in T' shall be non-decreasing when going from the leaf to the root (i.e. w). Since, $w = LCA(u, v)$, hence $val(w) \leq d(u, v)$ and $val(x) \leq val(w)$ where x is any internal node $\neq w$. However since we need a maximal tree, and since $d(u, v)$ is the minimum weight amongst all edges, therefore all the internal node values in T' will be equal to $d(u, v)$.

When we exchange v with the sibling subtree of u, we can see that the above condition does not hold true now. Therefore, for $\bar{T}'$, all internal node values $\geq d(u, v)$.
This suggests that at least one of the optimal tree has u and v as siblings, where (u, v) is the minimum edge weight.

Greedy Step: Considering that u and v are siblings, our greedy step will be to combine these nodes/vertices, thus reducing the size of the Graph G.

Let the new Graph after combining u and v be G', then G' will be defined as -

- G' will be a complete graph

- $V(G') = (V(G) \cup \{u'\}) \setminus \{u, v\}$ where u' is the combined vertex of u and v.
- We define another mapping $d' : (V', V') \rightarrow R^+$ such that
  - $d'(x, y) = d(x, y)\ \forall\ x, y \in V(G') \setminus \{u'\}$
  - $d(x, u') = min(d(x, u), d(x, v))\ \forall\ x \in V(G') \setminus \{u'\}$

Part B

Note: u, v and u' follow the same definition as in Part A
We have already established the relation between G and G'

Theorem: Constructing a tree from $T^*(G')$ by adding child nodes (leaf nodes) corresponding to u and v, to the node corresponding to the vertex u' and changing the value of this node to $d(u, v)$ gives us $T^*(G)$.
Proof:

As discussed in the previous part, there exists at least one optimal full binary tree for any (complete) Graph. Let $T^*(G')$ be one such tree. According to the greedy step, we just replace the leaf node of u' in $T^*(G')$ by another node with value $d(u, v)$ and child nodes as leaf nodes corresponding to nodes u and v in G. Let this new tree be T.

Claim: The new tree, T is consistent with G
Proof:
From the construction of T, we know that $LCA(u, x) = LCA(u', x) = LCA(v, x)$ where x is another node in the graph G. Since $d'(u', x) = min(d(u, x), d(v, x))$, therefore value of $LCA(u', x) \leq min(d(u, x), d(v, x))$ as $T^*(G')$ is consistent with G'. Since there is no difference in the rest of the tree, and the mappings d' and d are defined in such a way that there can be no inconsistency elsewhere. But, since the conditions for $LCA(u, x)$ and $LCA(v, x)$ satisfy all conditions necessary for consistency, hence we can say that T is consistent with G.

Claim: The new tree, T is optimal
Proof:
Also from the previous part, we know that replacing either u or v (not both) with any one other nodes in this tree either reduces or doesn't change the node values, and hence u and v can be siblings. Since $T^*(G')$ is optimal and no other node value needs to be changed as $d(u, v)$ is the minimum, we can argue that by construction, $T^*(G)$ is optimal.

From the previous two claims, we can say that our relation/theorem between $T^*(G)$ and $T^*(G')$ holds.

Part C

A simple implementation for the algorithm comprises of these steps -

1. For each node of the Graph G, store a parent pointer (pointer to a node in the to be constructed tree $T^*(G)$). (Let us define the mappings to be as $parent()$)
2. For each node u, initialize the $parent(u)$ to $NULL$.
3. Sort the edges according to their weights (ascending order), where weight is defined as the distance i.e. $d(u, v)$ for edge $(u, v) \in E(G)$
4. Iterating through these sorted edges, for each edge, say (u, v) -
   - Update parents of both u and v
     i.e. while $((parent(parent(u)) \neq NULL)$ $parent(u) = parent(parent(u))$ and same for v
   - If parents of u and v are same, then continue
   - Else, create a new node, say x and assign $parent(parent(u)) = x$ and $parent(parent(v)) = x$ and change $parent(u)$ to x and same for v.

This algorithm is $O(|E|log(|E|) + |V|^2)$. ($E = E(G)$ and $V = V(G)$)

The first term is contributed by the sorting of the edges.
For the second term -
We know from the construction of the tree that the maximum height of the tree can be $O(V)$. Therefore, updating the parent of a vertex will be $O(V)$. However, once updated, we will never see the nodes below it again. Hence, no matter how many times called, the update of parent will take only $O(V) + k$ time to reach the root of the final constructed tree (k is the number of times update is called). Hence, if we assume that update parent is called for each vertex, then the complexity will $O(|V|^2 + |E|)$.

Hence, we have an order $O(|E|log(|E|) + |V|^2)$. Since G is a complete graph, $|E| = O(|V|^2)$, therefore total complexity will be $O(|E|log(|E|))$