CS345A

Theoretical Assignment 1

DEEPANSHU BANSAL (150219)
MUKUL CHATURVEDI (150430)

Q1

Pseudocode

Data structure S is augmented binary search tree ordered according to x coordinate with augmented value y coordinates

Update(S, x, y){

```
while(S!=null && S.x_value<x){
        S = S->right
}
if(S == null){
Return; }
new node
node.x_value=x
node->right = S->right
node.y_value = y
S->right = node
while(node->left !=null && node->left.y_value =< y){
        node = node->left
}
S-right->left = node->left
}
```

Proof of Correctness

Our structure of binary tree is like it is sorted according to the x-coordinate of the non-dominated points . Now if we move to the right child of the node we are increasing our x-coordinate and decreasing our y-coordinate at the same time . Our node in the tree stores two values that are its x-coordinate and y-coordinate. We insert new points according to the update function.

As this binary tree is made up of non-dominated points thus whenever x-coordinate increases its y-coordinate decreases because of stair like structure. So we can say for a given node its x-coordinate is lesser than x-coordinate of its right child and greater than the x-coordinate of its left child . And at the same time its y-coordinate is greater than the y-coordinate of its right child and smaller than left child. This property helps us to move to the certain next level.

Now in update function we find an interval whose x-coordinate is just greater than x-coordinate of new point to be inserted and then the second while loop ensures that we remove points with x_value less than x and y_value also less than y of new point and finally we are changing the pointer of node in one step which ensures that all points which now are not non-dominated gets removed as in this process we came across only these points and any final non-dominated point doesn't come in way because of reducing x and increasing y.

Thus this algorithm with the help of BST keeps record of all non-dominated points in online fashion.

Time Complexity

For Update function

In our augmented binary search tree we first have to find node with x_value just greater than x coordinate of new point. This could be done in time h = height of data structure (<logi where i is ith point inserting) .Now we have to remove points with x_value less than x and y_value also less than y . So, for searching point with y_value greater than y will also be less than h = height of data structure(logi) . Thus, update function takes O(logi) for updating on ith point.

Thus, for updating total i points it will take O(ilogi).

Q2

Pseudocode

Int* FindingNewPoly(A, B,n)

A(x) <- polynomial from the set A

B(x) <- polynomial from the set B

C(x) <- multiply(A(x), B(x))

Initialize an array C[]

for(int i=0;i<20n;i++){

C[i] <- coefficient(C(x),i);

}

return C ;

Our function returns the array of numbers if C[i] = 0 that means i is not present in the result otherwise its value indicates the number of times it gets realized as sum of two elements one from A and other from B.

Proof of Correctness

Create two polynomial

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_{10n}x^{10n}$$

$$B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + ... + b_{10n}x^{10n}$$

Where a_i or b_i is 1 if that element is present in the set otherwise 0.

$$C(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + ... + c_{20n}x^{20n}$$

Now is $c_i = 0$ then i is not present in the set C . If c_i != 0 then then i is present in the set and we can infer that **"number of times "i" is realized as a sum of elements in A and B" is equal to c_i .**

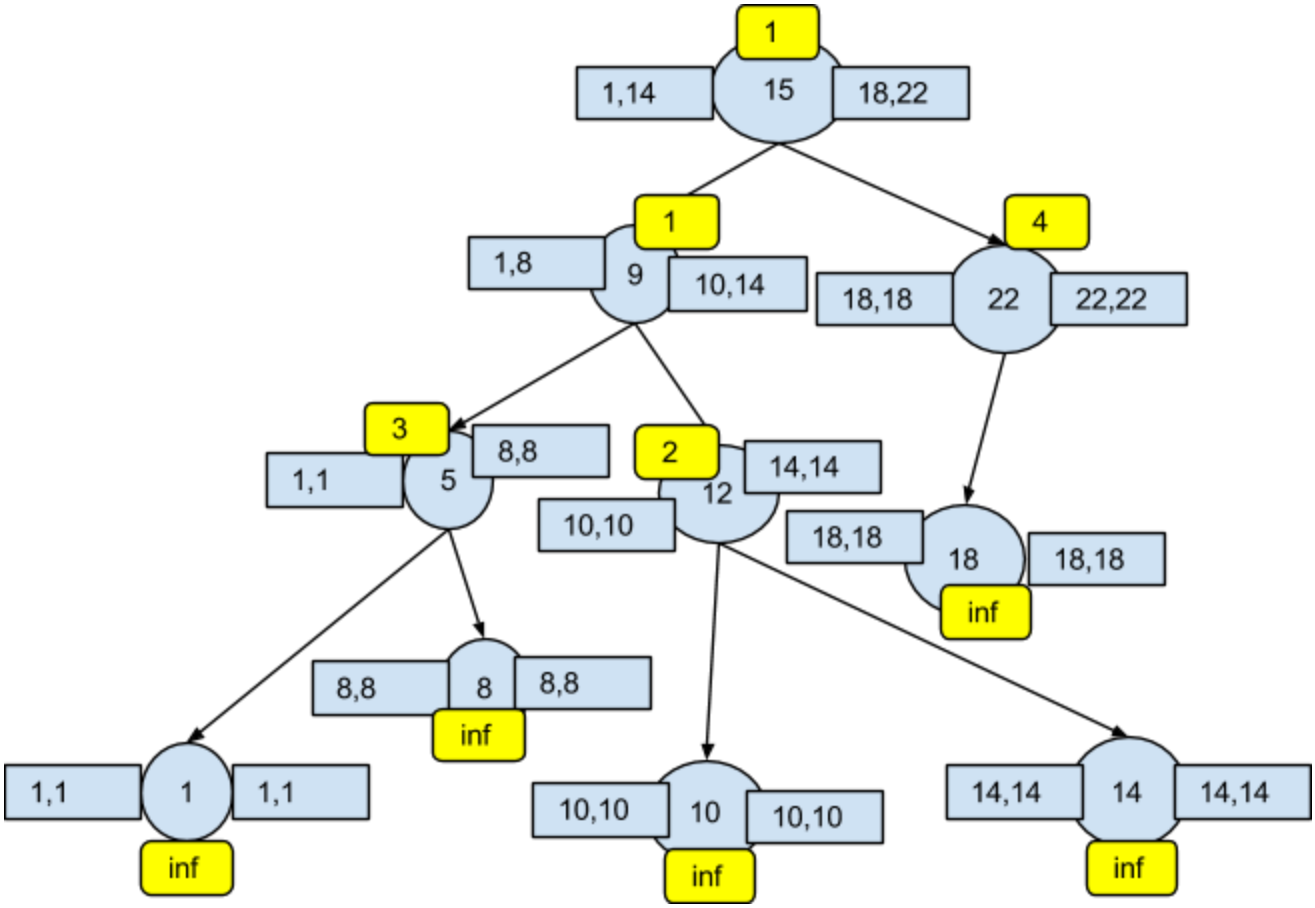
$$c_i = a_0b_i + a_1b_{i-1} + a_2b_{i-2} + ... + a_ib_0$$

Now a_rb_{i-r} == 1 if both are one that would mean "r" is present in A and "(i-r)" is present in B so their sum will be present in C. The number of time it is realized as a sum of two number if given by it and if it is never present in C then definitely it would be zero.

Time Complexity

We can form polynomials A(x) and B(x) in **O(n)** time just by traversing through the sets A and B by setting the coefficient of x^i to 1 if i is present in the set otherwise set it to 0 by default. Now all elements are smaller than 10n thus it takes O(n) time. As discussed in class multiplying two n degree polynomials takes **O(nlogn)** time and we can output the elements of C in **O(n)** time just by traversing through the the polynomial and return the coefficient of x^i denoting the number of times it is realized thus both tasks can be completed in **O(cn + nlogn)** that is **O(nlogn).**

Q3


1
1,14
15
18,22
1
1,8
9
10,14
4
18,18
22
22,22
3
1,1
5
8,8
2
10,10
12
14,14
18,18
18
18,18
inf
8,8
8
8,8
inf
1,1
1
1,1
inf
10,10
10
10,10
inf
14,14
14
14,14
inf

Pseudocode

INSERT(S,x)

```
insert(S,x){
        while(S!=null){
                if(S.value>x){
                        if(x<S.leftmin){
                                S.leftmin=x
                        }
                        if(x>S.leftmax){
                                S.leftmax=x
                        }
                        if(S->left!=null)
                        S=S->left
                        else{
                                S->left.value = x
                                S->left.rightmin = x
                                S->left.rightmax = x
                                S->left.leftmin = x
                                S->left.leftmax = x
                                S->left.diff = infinity
                                S=S->left
                        }
                }
                if(S.value<x){
                        if(x<S.rightmin){
                                S.rightmin=x
                        }
                        if(x>S.rightmax){
                                S.rightmax=x
                        }
```

```
                    if(S->right!=null)
                    S=S->right
                    else{
                            S->right.value = x
                            S->right.rightmin = x
                            S->right.rightmax = x
                            S->right.leftmin = x
                            S->right.leftmax = x
                            S->right.diff = infinity
                            S=S->right
                    }
            }
        }
}
```

Search(S,x)

```
search(S,x){
while(S!=null){
        if(S.value>x){
                S=S.right;  }
        else if(S.value<x){
                S=S.left;
        }
        else{
                return S;
        }
}
return -1;
}
```

Delete(S,x)

```
Delete(S,x){
s = search(S,x)
p = predecessor(S)
s.value = p.value
t = p.parent
p = null
while(t!=root){
        set(t)
        t = t.parent
        if(t  = root){
                break;
        }
}
}
```

Set(node)

```
Set(node){
if(node.left!=null||node.right!=null){
        node.leftmin = (node->left).leftmin
        node.leftmax = (node->left).rightmax
        node.rightmin = (node->right).leftmin
        node.rightmax = (node->right).rightmax
        diff =
min(node->left.diff),(node->right.diff),node->right.leftmin-node->value,node->value-(node->left).r
ightmax
}
if(node->left==null && node.right!=null){
        node.leftmin = node.value
```

```
        node.leftmax = node.value

        node.rightmin = (node->right).leftmin

        node.rightmax = (node->right).rightmax

        diff =
min(node->left.diff),(node->right.diff),node->right.leftmin-node->value,node->value-(node->left).r
ightmax

}

if(node->right =  null && node.left!=null){

        node.leftmin = (node->left).leftmin

        node.leftmax = (node->left).rightmax

        node.rightmin = node.value

        node.rightmax = node.value

        diff =
min(node->left.diff),(node->right.diff),node->right.leftmin-node->value,node->value-(node->left).r
ightmax

}

if(node->right = null && node->left  = null){

        node.diff = infinity

        node.leftmin = node.value

        node.leftmax = node.value

        node.rightmin = node.value

        node.rightmax = node.value


}

}
```

Proof of Correctness

For **Insert** function at every step we are updating the maximum value of left subtree of node and minimum value of right subtree of node. Now the minimum gap till that node is diff as set in **Set** function as minimum gap can be between node and its left subtree maximum value or diff btw node and minimum of its right subtree or its left or right difference.

For **Delete** function the nodes whose values will be changed lies on the path of node's predecessor to the root of the tree which are taken care of set function.

For **Search** function it is just a normal binary search.

For **Min-Gap** function it is returning true as at each node we are taking care of min-gap in the subtree of that node and by recursion we can assure that our min-gap function is returning the correct output.

Time Complexity

T(Insert(S,x)) = O(h)

As we all moving one height forward at each iteration and for BST h = O(logn);

T(Insert(S,x)) = O(logn)

T(Set(S)) = O(1)

As it involves only constant number of opertations.

T(Set(S)) = O(1)

T(Delete(S,x)) = O(h)

As we all moving one height forward at each iteration and for BST h = O(logn);

T(Delete(S,x)) = O(logn)

T(Search(S,x)) = O(h)

As we all moving one height forward at each iteration and for BST h = O(logn);

T(Search(S,x)) = O(logn)

T(Min-Gap(S,x)) = O(1)

As we will just report the (root->diff) each time.

T(*Min-Gap****(S,x)) = O(1)***