Akshay Iyer-190070006
Minor
Hackerrank id- h190070006

Honour code-
I pledge on my honour and the Gita, that I have not used(except assignment 1) and will not use any unfair means(give or receive unauthorised assistance of any kind) during this examination, or any other future examination or assignment in this course. I shared my code with a friend in assignment 1, which I THOROUGHLY regret, and which I sincerely beg your forgiveness for.

Q1:

1.1

1. I figured that we can't directly implement a tree. As paths and successive connections may be disjoint, also there's no way to know which is the parent and which is the child, without having all the information about all the connections. Also one parent can have multiple children. So vectors was the way to go. Every node stored a vector of all its neighbours addresses.
2. I first made the undirected graph structure of the tree. I stored a vector for every node, which contains all the other nodes its connected to. Then I needed to find which is the parent and child in that graph. I figured that the absolute children at the end, the lowest generation per sae, would be connected only to 1 other node. Thus I first checked all the nodes having only one connection and assigned those as the children to the node they were connected to. Thus I assigned a direction to those edges.
3. Then I removed those nodes from my vector. I also removed all other appearances of those nodes in other node children vectors. Thus as we have effectively removed the lowest generation, the second lowest generation now becomes the new lowest generation. Thus we repeat the process and eventually assign a direction to all the edges. I had to use special cases for the root of the entire tree here and there, but those were just the small technicalities of the implementation.
4. I implemented the minimum height of the tree function, by taking the minimum of the heights of the left and right subtrees, +1. Apart from that I just normally implemented the node struct etc.

1.2

Time complexity of converting the edges into directed tree edges, isn't too good, its 2n, as I check every node's neighbours for the popped lowest generation node. I do that n times- ((2n)+(2n-2)….) approx O(n^2).
Then to actually find the minimum, it took a worst case time complexity of O(n), where n is the number of nodes of the tree, as we pass through each of the nodes once, taking the minimum height of that subtree and returning it or something.

Q2

2.1

We basically need to return the number of nodes so that the parent can use it to calculate if it is a quad node or not. We also need to retain some additional information in the return value such that all ancestors of a non completely quad node subtree, will also become a non completely quad node subtree.
Define a function to count the number of nodes in a tree if it is a quad node else, it returns -1.
I have decided -1 as no other possibility can return aa negative number like that, except if some conditions are being violated.

```
Int k=0;
Int quadnode(Node* root)
{
I
If root==NULL
{
Return 0;
}
int a1= quadnode(root.left);
int b1= quadnode(root.right);
If a1<0 or b1<0
        return(-1)
Else
{
        If (b1-a1)^2<=16
                Return 1+a1+b1;
        Else
        {
                k++;
                Return -1;
        }
}
}
```

k will give us the total number of such nodes asked in the answer.
We increment k only if the condition that |left-right|<=4 is not satisfied, so that it only captures the first ancestor that is not a quad node. Any propagated -1 will not increase k, hence solving our question.

2.2

My algorithm is correct because every time we find that something is not a quad node, we return -1 which keeps propagating up in the tree. Thus all the ancestors store -1 which shows that they aren't quad nodes. It was also necessary for me to retain the number of nodes as information in the value returned by the function as that's the only way we can check if the present node is a quad node or not by checking if the difference is greater than 4 or not. Thus this is a nice and very efficient method. It also passes through all the nodes of the tree and tells us if each is a quad node or not. We increment k by 1, only if the -1 is generated due to difference in the tree branch sizes, and not if it is propagated, thus k finally gives us the total number of nodes satisfying the condition as asked in the question.

2.3

Thus, as we travel through all the nodes once and carry out all of my well crafted conditions!
Worst case total time complexity is O(n).(if all the nodes are quadnodes).

Q3

3.1

First of all use the compare function in order to compare all the characters. Let the middle character(in terms of their global hierarchy independent of the string, using the compare function to find the middle one) of the string be the root. Then keep comparing the rest of the characters(using the compare function) in the string and depending upon the compare key put them left or right. Now, as we are doing this we store a vector in each node that stores all the possible distinct characters that can appear in front of that particular letter.

If a particular letter is repeated x times, we will store x vectors in that node, each having only the characters ahead of that particular instance of appearance.
Let these instances be called o1, o2, o3 etc.(as in the example down below).
But c will store only o1. NOT o2 and o3, as then cases will repeat. o1 handles everything itself.

Every vector can only store the first instance of a repeated character in front of it in the original string.

Eg- in corona, o2 will be in front of r, and stored in r's vector.
Vector of o1, will contain r and o2(not o3). Node o contains a heap of vectors- o1's vector, o2's vector etc.

Then we define a recursive function-
Called **perm(char c)**, which takes the input as a character and returns all the number of mutants with that as the first letter.
Each perm(char) will be (the sum of all the perms stored in its vector) + 1(for the case that we take no further string and our mutant terminates there).
This recursion will keep continuing. The base case will be, when we eventually reach the last character in the original name of the virus. At that point we will return a 1.

Now to get the final answer of the total number of mutants, we see that it will be the sum of all the particular mutants starting at a particular character. Thus we do the sum of perm of all characters. But when it comes to repeated characters, we will **only take the first instance of that** in the sum, I.e. we take only o1. As any mutant starting with o1 handles all cases for that mutants starting with o.

This is my algorithm. Strangely enough I only used insert, at the start while building the tree; and find operations of the bst. No delete.

We will obviously drastically improve the efficiency by adding an int perm number in every node, which will store 0 initially. As soon as we calculate the actual value of that perm, we will store that number in that. Whenever we are returning perm, we will first check if a non 0 value is in the int perm number and return that. Only if int perm is 0, we carry out the recursion. Thus every recursion is carried out only once, and after that its direct stored value is accessed.

3.2

Correct as, if we only add the immediate next single appearance of the repeated character in the perm of that node, it also covers the cases where the sub mutant started from any future instance of that same repeated character. I've explained it pretty well above too maybe that will suffice for this question too. There is only one node in the binary tree given to every single distinct character. The number of vectors stored in that node is equal to the total number of appearances of that character in the original string.

3.3

The only data structures I used were the binary search tree, and vectors. Binary search tree helps a lot with time complexity, as the find operation etc takes a maximum worst case time complexity of O(log n) only. A normal linear search through a vector for the elements will result in a resultant time of worst case o(n). Thus the binary tree helps.
For the perm function, we have a worst case time complexity of O(n^2)(as even though elements are stored in the vector once calculated, it will take a maximum of n function additions to get the value of a single perm(even though any one individual of the n function calls will take O(1)))
(n is the number of characters in the original string). This is as perm of every node is calculated only once, and then stored in a vector, which can be accessed with O(1) later.

Q4

4.1

The only change I made to the original bst done in class was, I converted it into a balanced binary tree after every iteration. I did this using the concept of AVL trees, which I found on geeksforgeeks, and modified the geeksforgeeks functions, and added a few functions of my own, in order to suit this particular problem.

To be honest, if not for geeksforgeeks I would've never been able to think of this specific avl tree method within the assignment submission date and time, as it was pretty complex.

I defined 2 functions left rotate, and right rotate, which rotate any subtrees that were unbalanced after the operation we did, and made them balanced.
Unbalanced means difference in height was more than 1. As our tree was balanced before the operation, the only way it can get unbalanced is if the difference in height becomes 2 after the operations.
These 2 extra nodes on one side of the node we are currently correcting, can be **left left**(immediate extra node is on the left of node, the extra one after that is again to the left of the immediate extra node of the node), **left right**, **right left**, and **right right**. We modify the subtree by rotating it accordingly to each of the cases. We modify every such node (and that subtree) which is unbalanced, using recursion.

I have understood the geeksforgeeks concept very well, and implemented it similarly. As during every insertion(i) or deletion(d), the conversion of the tree into a balanced tree takes only O(1), and the tree is always balanced, any insertion or deletion takes a WORST CASE COMPLEXITY of O(log n), as the tree is balanced so maximum height will be log n.

I defined the addition(a) function myself, which will have a worst case complexity of O(n) if all the nodes lie in that range. Else we only check those nodes and modify them, which lie in the given range. I didn't even check the nodes that lied outside the range.

If we hadn't made this change in the original function done in class, the worst case height would've been n, hence out worst case time complexity for everything would've been O(n).

Even search(g) has a maximum time complexity of O(logn), as our tree is always balanced, which wouldn't have been the case in the bst done in class(in class, worst case O(n)).

Q5

5.2

First we find the total number of 4 node trees with height 2(by taking products as in the big explanation I have given below). That will be (number of height 1, 3 node trees)* (number of height 0, 0 node trees)*2+ ((number of height 1,2 node trees)* (number of height 1, 1 node trees)+ (number of height 1, 1 node trees)* (number of height 1, 2 node trees)) (All other cases will be 0). Similarly do for the other values of height of the final tree, and sum over them.
The individual number of trees I used to calculate the above, will again be calculated recursively, until they reach the base case.

Then total number of 4 node trees with height 3. Then total number of 4 node trees with height 4. Then we add all of them.

I have also explained my recursion algorithm very well in 5.1 diagrams and writeup, as it was easier to explain with diagrams and handwritten cases. Please check that if this isn't enough explanation.

5.3

I implemented a function using recursion that finds the total number of trees of some height h and nodes n.

We multiply the total number of sum trees in the left and right subtrees and add all of them.
We use the concept that total number of trees of height h is total number of(left subtree) trees of height h-1*total number of right subtrees of height less than h-1. We implemented this by using 2 for loops. One will control the height of the right tree(left tree fixed at h-1). Another loop will vary the number of nodes in the left subtree, hence we correspondingly get the number of nodes in the right subtree too. We take product of the 2 subtree recursions, and sum all of them corresponding to every variation in the 2 loops. This gives us total ways where left subtree has height h-1, right has height less than that. We multiply that by 2 for the cases where only the right subtree has height h-1.

Then we add all the cases where the 2 subtrees both have height h-1.
I took base cases as where h or n is 0, or where h>n(no such tree possible), or where h<log(n+1) (again no such tree is possible), or where h==n, where we analytically see that the number of trees will be 2^(n-1). I did all of this, as more the number of base cases, less recursions my program will take to get the answer, and hence my program's time complexity will decrease and it becomes faster. Also every time I found the value of the function for a particular case, I stored it as a case in the 51*51 vector that I created, so that no repeated recursions take place.

Thus we obtain a function that returns total trees of height h, nodes n. Then we add all these functions for height in the range l to r, in order to get the final answer. We also keep taking modulo 1000000007 everywhere to make the numbers not exceed the range of the data types we store them in.
I had to work very hard in order to make it pass all the test cases by increasing the number of cases to be treated as base cases in the recursion, and it became efficient enough eventually.

5.4

If n is the number of nodes, and h is the height of the tree,

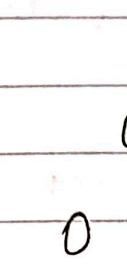
Space complexity - I stored everything in a 51*51 vector. Thus my space complexity was O(51*51) only. My stack space reached a maximum of O(n*h) as it will need the values of the functions with n and h less than equal to that value, and hence will call all of those functions recursively hence increasing stack size.

Time complexity - We find the value for each tree of height h, nodes n only once. We then store it in the vector, which we access later, so that we don't have to repeat the recursion everytime. Thus as we carry out the operations only once for every (h,n) , my time complexity will be a maximum of O((number of nodes as input * upper limit for the height as entered by the user)^2). This is as we care to fill values only until those values of (n, h) in the 51*51 vector, and complexity of filling every f(h,n) for the first time is O(h,n), as will nee about those many recursive function calls for all the lesser values of h and n before it.
Thus net worst case time complexity will be O(50^4).(as maximum input user will give is 50 only).

5.1) for $n = 4$, for $h = 4$,

i) if other subtree has height 0

Every shape corresponds to 1 arrangement of numbers only.



$\therefore$ each edge here can go left or right. $\therefore 2^3$ ways.

(h == n case).

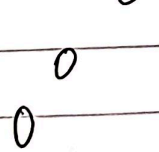
ii) for $n = 4, h = 3$

subtree

other can have height $h = 1$ (as $3-1$)

one subtree has height 2 → can have 2 or 3 nodes (nodes cannot be $< h-1$).

a) if 2 nodes, in left subtree



subtree

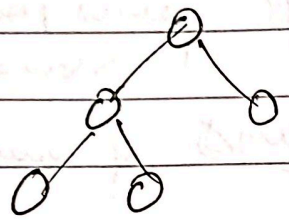
left no. of arrangements can be found by recursion of num(2,2).

it will come out to be 2^1. (h == n case)

we multiply that by 2, as height 3 can be on the left or right subtree

$= 4$

b) if left subtree has 3 nodes, right 2;



num(3, 2) will be calculated recursively

∴ 1 way × 2 = (2) Only one way to arrange 3 nodes in height 2 (found recursively in my program anyway)

(as left subtree or right subtree both could have height 3) i.e. o o o

ii) how can subtree has height 2

left has height 3, right has 2 (or vice versa).

iii) For $n = 4, h = 2$

one subtree has height 1, other has height 0 or 1

with $h = 2$, maximum 3 nodes. ∴ 4 nodes (4) not possible

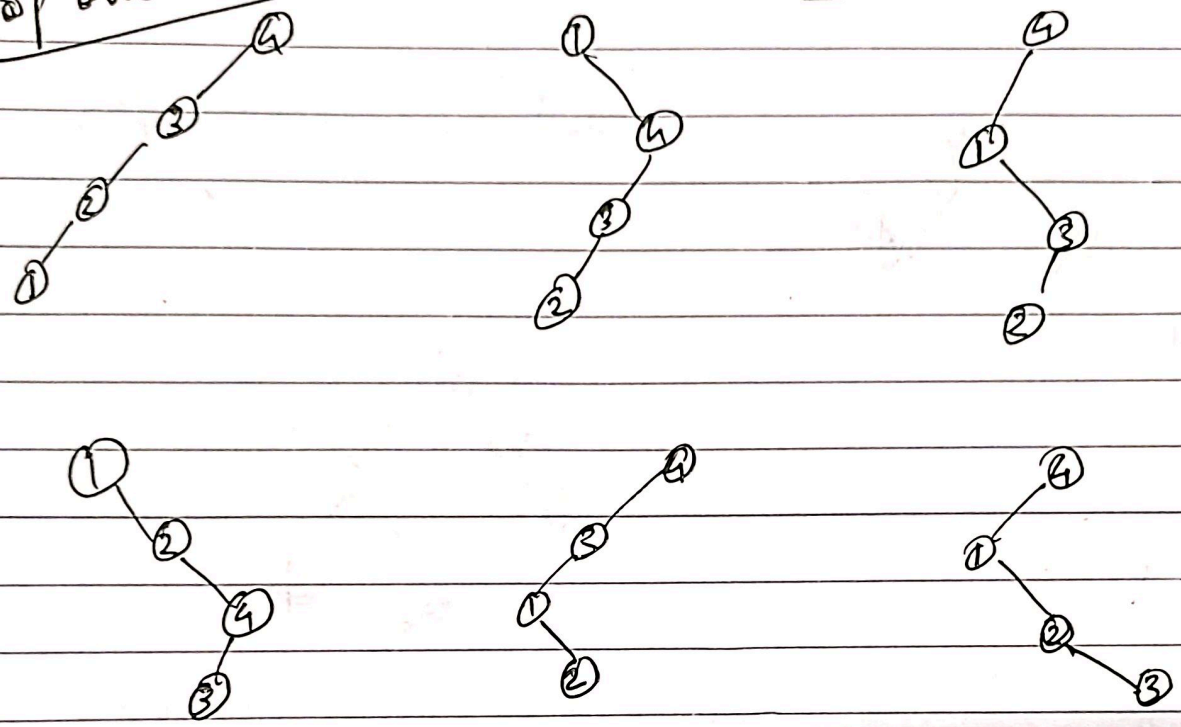
(In my code, this happened recursively, we got 0)

~~4+6+8~~ $8 + \underbrace{4+3}_{h=3} = 14$ cases. (as we add all $(4,h)$ for $h \in [1,r]$)

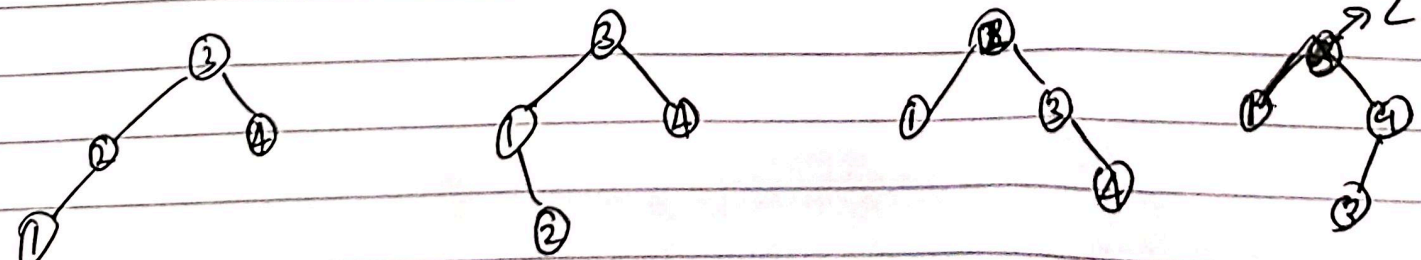
$\therefore$ 14 cases are -

Top one always root

For $h=4$

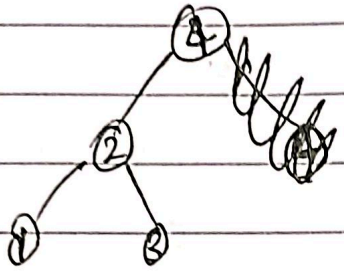


For $h = 3$

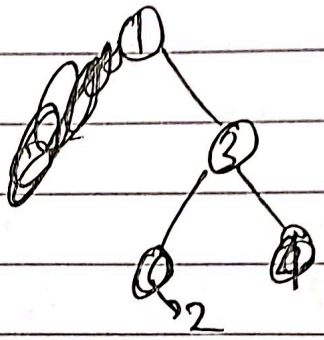


if $n_{\text{left subtree}} = 2$.

if $n_{left\ subtree} = 3$,



and similarly for right



for $h = 2$ and less, no cases.

∴ for $n = h$, $l = 2$, $r = h$,

output = 14.