Report on question 2

Problem:

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation. Each node of the tree holds, in addition to the pointers, the name of an employee and the employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

Remarks and assumptions of the solution:

- Remark : Data Structure to store the company employee details and organisational hierarchy is a Tree with left-child, right-sibling representation, where root of the tree is the president of the corporation and parents-child relationship represents the immediate supervisor-employee relationship. Each node will have the name of the employee and the conviviality rating (metric to maximize) of the employee.
- Assumption: an Employee has only one immediate supervisor(a child has only one parent according to our data structure)
- Assumption inviting President of the company is not compulsory

Approach on the solution:

Our solution uses Dynamic programming to solve this problem,
Corresponding recursion would be (to find the maximum conviviality ratings) :

```
maximumConvivality(Node):
    If Node.Children = null
        Return Node.rating
    Else
        maxRatingChildren = 0
        For C in Node.Children
            maxRatingChildren += maximumConvivality(C)
        maxRatingGrandChildren = Node.rating
        For G in Node.GrandChildren
            maxRatingGrandChildren += maximumConvivality(G)
        If maxRatingGrandChildren >= maxRatingChildren
            Return maxRatingGrandChildren
        Else
            Return maxRatingChildren
```

By applying Dynamic Programming to above recursion we calculate the maximum possible Conviviality up to that node and keep track of who are to be invited to maximize Conviviality. When applying dynamic programming we traverse the tree from bottom to top and if “maxRatingGrandChildren >= maxRatingChildren” is True we assign “fitt to be invited” of that node as True. Then to create the invitee list we traverse the tree top to bottom. If “fitt to be invited” is True for a node we add that node to the invitee list and sets “fitt to be invited” of it’s children False. If “fitt to be invited” is False for a node we traverse to the next node.

We apply dynamic programming by storing the maximumConvivality up to that node in the Node.rating field (we doesn’t use a separate data structure to store maximumConvivality since it’s not necessary). “fitt to be invited” parameter tells us that if it’s true, to achieve maximumConviviality marked on that node we must invite that node.

Algorithm(pseudo code) / Time complexity analysis :

Macro function(`getRatingSumOfChildren`) :

Child = no of child node each node has
C = constant

Time	Pseudo Code
	`def getRatingSumOfChildren(parent):`
C	`currentNode = parent.leftChild`
c	`ratingSum = 0.0`
Child	`while not currentNode is None:`
C	`ratingSum += currentNode.rating`
c	`currentNode = currentNode.rightSibling`
c	`return ratingSum`

Time complexity(`getRatingSumOfChildren`) macro = 3c + (Child x 2c) = **O(Child)**

Macro function(`getRatingSumOfGrandChildren`) :

Child = no of children each node has
Gchild = no of grandchildren each node has
C = constant

Time	Pseudo Code
	`def getRatingSumOfGrandChildren(parent):`
C	`ratingSum = 0.0`
C	`child = parent.leftChild`
Child	`while not child is None:`
C	`grandChild = child.leftChild`
Gchild	`while not grandChild is None:`
C	`ratingSum += grandChild.rating`
C	`grandChild = grandChild.rightSibling`
C	`child = child.rightSibling`
c	`return ratingSum`

Time complexity(`getRatingSumOfGrandChildren`) macro = 3c + Child x (2c + (Gchild x 2c)) =
= **O(Child x Gchild)**

Preprocessing :

C = constant
N = no of nodes in the tree
H = no of children a node can have
G = no of grandchildren a node can have

Time	Pseudo Code
C	`stack =  Stack() #to accesses the tree from bottom to top`
C	`queue = Queue()`
C	`queue.enqueue(tree.root)`
C	`stack.push(tree.root)`
C	`currentNode = tree.root`
N	`while not queue.isEmpty():`
C	`    currentNode = queue.dequeue()`
C	`    if not currentNode is None:`
C	`        currentNode = currentNode.leftChild`
C	`        if not currentNode is None:`
C	`            queue.enqueue(currentNode)`
C	`            stack.push(currentNode)`
H	`            while not currentNode.rightSibling is None:`
C	`                currentNode = currentNode.rightSibling`
C	`                queue.enqueue(currentNode)`
C	`                stack.push(currentNode)`
N	`while(not stack.isEmpty()):#calculate maximum possible rating for each sub`
	`tree intiating from Node`
C	`    currentNode = stack.pop()`
O(H)	`    childrenRatingSum = getRatingSumOfChildren(currentNode)`
O(HxG)	`    grandChildrenRatingSum = getRatingSumOfGrandChildren(currentNode)`
C	`    if((currentNode.rating + grandChildrenRatingSum)>=childrenRatingSum):`
C	`        currentNode.invite = True`
C	`        currentNode.rating += grandChildrenRatingSum`
C	`    else:`
C	`        currentNode.invite = False`
C	`        currentNode.rating = childrenRatingSum`

Time complexity(`Preprocessing`) = 5C +Nx(6C + Hx(3C)) + Nx(7C + O(H) + O(HxG))
= O(N x H + N x H x G)
= O(N x H x G)

Creating Invitee List :

C = constant
N = no of nodes in the tree
H = no of children a node can have
G = no of grandchildren a node can have

Time	Pseudo Code
C	`inviteList = []`
C	`queue = Queue()`
C	`queue.enqueue(tree.root)`
C	`currentNode = tree.root`
C	`if(currentNode.invite):`
C	`    inviteList.append(currentNode.name)`
C	`    childNode = currentNode.leftChild`
H	`    while not childNode is None:`
C	`        childNode.invite = False`
C	`        childNode = childNode.rightSibling`
N	`while not queue.isEmpty():`
C	`    currentNode = queue.dequeue()`
C	`    if not currentNode is None:`
C	`        currentNode = currentNode.leftChild`
C	`        if not currentNode is None:`
C	`            queue.enqueue(currentNode)`
C	`            if(currentNode.invite):`
C	`                inviteList.append(currentNode.name)`
C	`                childNode = currentNode.leftChild`
H	`                while not childNode is None:`
C	`                    childNode.invite = False`
C	`                    childNode = childNode.rightSibling`
H	`            while not currentNode.rightSibling is None:`
C	`                currentNode = currentNode.rightSibling`
C	`                queue.enqueue(currentNode)`
C	`                if(currentNode.invite):`
C	`                    inviteList.append(currentNode.name)`
C	`                    childNode = currentNode.leftChild`
H	`                    while not childNode is None:`
C	`                        childNode.invite = False`
C	`                        childNode = childNode.rightSibling`
C	`return inviteList`

Time complexity(`Creating Invitee List`) = 8C + Hx2C + Nx(8C + Hx2C +Hx(5C + Hx2C))

Time complexity(`Creating Invitee List`) = O(H + Nx(H + (HxH)))
= O(NxHxH)

Time complexity of the algorithm = Time complexity(`Creating Invitee List`) + Time complexity(`Preprocessing`)

Time complexity of the algorithm = O(NxHxH) + O(N x H x G)

Assume G = HxH
Then :
Time complexity of the algorithm = O(NxHxH) + O(N x H x H x H)
= **O(N x H x H x H)**

Where : N - Total no of nodes in the tree
H - Maximum no of children a node can have