Homework 3 **Branch T. Archer, III**
Due date: Friday, October 30 – submit by 11:00 PM

1. *Purpose: reinforce your understanding of the matrix-chain multiplication problem, memorization, and recursion*.

a) (4 points) Please implement a recursive, dynamic programming, and a memoized version of the algorithm for solving the matrix-chain multiplication problem described in our textbook (Chapter 15), and design suitable inputs for comparing the run times of your algorithms. Describe input data, report, graph and comment your results, you don't have to submit your program.

Test Parameters

Computer:	Dell Inspiron I1521 (laptop)
Operating System:	Microsoft Vista
Processor:	AMD Turion 64 X2 Mobile Technology TL-56, 1.80 GHz
Memory:	2.00 GB
Language:	Java 1.6.0_14
Runtime:	Java(TM) SE Runtime Environment (build 1.6.0_15-b03)

Methodology

The execution time for three different algorithms (Dynamic Programming, Memoization, and Recursive) was measured for matrix-chains of the following lengths:

5, 10, 15, 16, 17, 18, 19, 20, 21, 50, 100, 150, 200, 300, 400, 500, 700, 850, 1000, 1500

The following considerations drove the choice of chain lengths:

- For a chain length ≤ 10, all algorithms required a time at or below the resolution of measurement ability (1 ms).
- Execution time was noted to grow rapidly in the 15-21 range for the Recursive algorithm, while remaining negligible for the other two algorithms.
- Beyond a chain length of 21, the Recursive algorithm was dropped from the study due to impractical growth of execution time. Chain lengths > 21 were selected to demonstrate the asymptotic behavior of the other two algorithms.

To avoid any bias toward a worst case or best case behavior (i.e., to obtain average case behavior), each dimensional element of the chain (i.e., the p array) was randomly chosen as an integer in the range [1,100]. Each algorithm was executed 5 times, and average execution times were recorded.

For the Recursive algorithm, I made a small modification to the algorithm provided on p. 345. Instead of passing p as a parameter, I made p a class-level variable. This change improves execution time and dramatically reduces the size of the stack.

Fig. 1 demonstrates the execution time of the algorithms, T(n), vs. n, the length of the matrix chain. Because the performance of the Recursive algorithm is so different from the other two methods, Fig. 2 separately demonstrates $\log_2 T(n)$ vs. n.

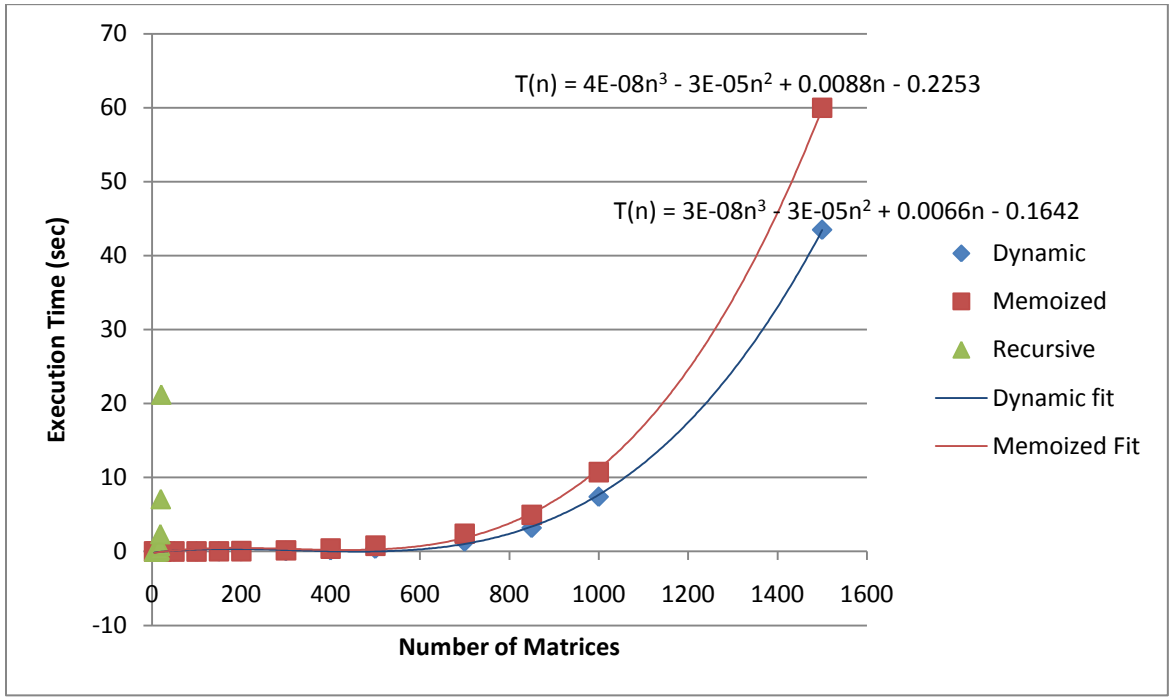

Fig. 1. Execution time (sec) vs. Number of Matrices for 3 different algorithms

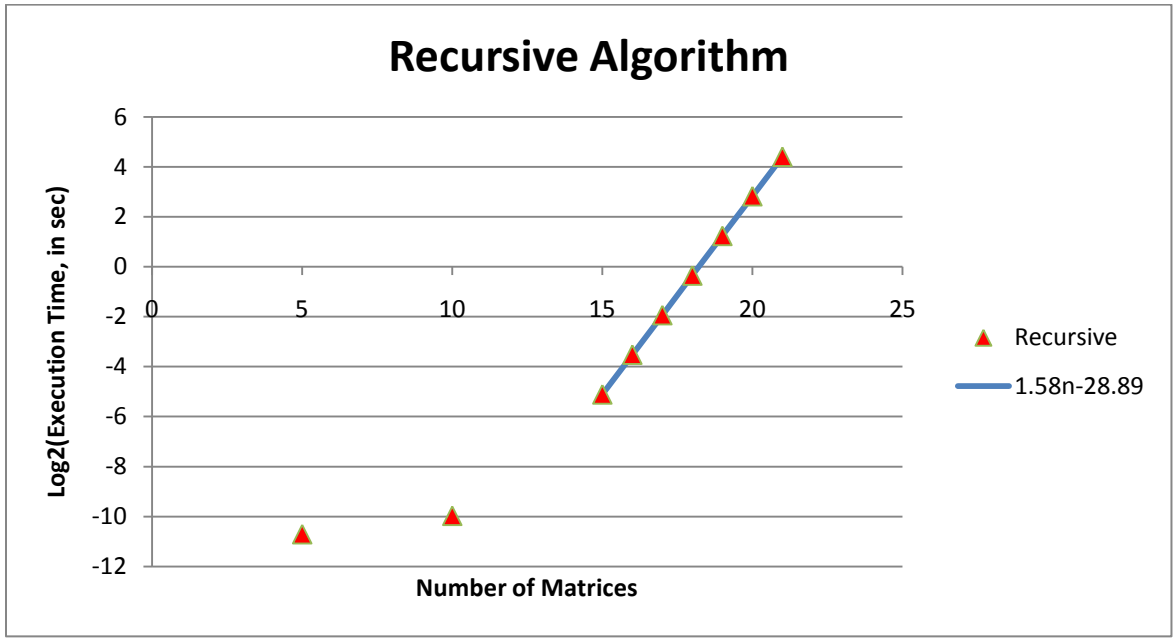

Fig. 2. $\log_2(T(\text{sec}))$ vs. Number of Matrices for Recursive algorithm

From these graphs, we see that the Dynamic algorithm has the best performance, followed closely by the Memoized algorithm, and with the Recursive algorithm a distant last. Table 1 gives formulas for fitting the data to curves. As the figures show, excellent fits are obtained for the Dynamic and Memoized algorithms with 3rd order polynomials (which is to be expected since 3 loops are involved). The coefficient of the n^2 term is identical for the two algorithms. The Dynamic algorithm is asymptotically superior by virtue of its smaller n^3 coefficient. An excellent exponential fit is achieved for the Recursive algorithm in the range $15 \leq n \leq 21$ (Fig. 2).

Method	T(n)
Dynamic	$3 \times 10^{-8}n^3 - 3 \times 10^{-5}n^2 + 0.0066n - 0.1642$
Memoized	$4 \times 10^{-8}n^3 - 3 \times 10^{-5}n^2 + 0.0088n - 0.2253$
Recursive	$2^{1.58n-28.89}$

Table 1. Functional approximations for execution times of 3 different algorithms for solving the matrix-chain multiplication problem.

b) (2 points) Find an optimal parenthesization of a matrix-chain product whose sequence of dimension is <5, 4, 4, 11, 7, 8, 7, 12, 6, 9, 8, 7, 5, 12>. How many multiplications does this parenthesization require?

((A1(A2(((((((((A3A4)A5)A6)A7)A8)A9)A10)A11)A12)))A13)

which requires 2728 multiplications.

How many multiplications are required by a naive algorithm that multiplies the matrices in their input order?

$$m = p_0p_1p_2 + p_0p_2p_3 + \cdots + p_0p_{n-1}p_n = p_0 \sum_{i=1}^{n-1} p_ip_{i+1}$$

$$= 5 \cdot (4 \cdot 4 + 4 \cdot 11 + 11 \cdot 7 + 7 \cdot 8 + 8 \cdot 7 + 7 \cdot 12 + 12 \cdot 6 + 6 \cdot 9 + 9 \cdot 8 + 8 \cdot 7 + 7 \cdot 5 + 5 \cdot 12)$$

$$= 5 \cdot (16 + 44 + 77 + 56 + 56 + 84 + 72 + 54 + 72 + 56 + 35 + 60) = \boxed{3410}$$

2. *Purpose: reinforce your understanding of chain matrix multiplication and greedy algorithms (when these fail).*
(4 points) Recall the matrix chain multiplication problem: minimize the number of scalar multiplications when computing $M = M_1M_2 \ldots M_n$ by choosing the optimum parenthesization. For each of the following greedy strategies, prove, by coming up with a counterexample, that it does not work. Recall also that the problem is fully defined by dimensions d_i for $i = 0, \ldots, n$, and M_i is a d_{i-1}x d_i matrix. Thus each example you give need only list the d_i values.
(a) First multiply M_i and M_{i+1} whose common dimension d_i is smallest and repeat on the reduced instance that has M_iM_{i+1} as a single matrix, until there is only one matrix.

The example on p. 332 in our textbook provides a counterexample:

$$d =< 10,100,5,50 >$$

This greedy algorithm suggest that we begin with the common dimension $d_2 = 5$. This choice produces a parenthesization $(A_1(A_2A_3))$ resulting in

$$(100 \times 5 \times 50) + (10 \times 100 \times 50) = 25{,}000 + 50{,}000 = 75{,}000$$

multiplications. However, the optimum parenthesization is $((A_1A_2)A_3)$, which results in

$$(10 \times 100 \times 5) + (10 \times 5 \times 50) = 5{,}000 + 2{,}500 = 7{,}500$$

multiplications.

(b) First multiply M_i and M_{i+1} whose common dimension d_i is largest and repeat on the reduced instance that has M_iM_{i+1} as a single matrix, until there is only one matrix.

My counterexample is:

$$d =< 10,10,20,50 >$$

This greedy algorithm suggest that we begin with the common dimension $d_2 = 20$. This choice produces a parenthesization $(A_1(A_2A_3))$ resulting in

$$(10 \times 20 \times 50) + (10 \times 10 \times 50) = 10{,}000 + 5{,}000 = 15{,}000$$

multiplications. However, the optimum parenthesization is $((A_1A_2)A_3)$, which results in

$$(10 \times 10 \times 20) + (10 \times 20 \times 50) = 2{,}000 + 10{,}000 = 12{,}000$$

multiplications.

(c) First multiply M_i and M_{i+1}that minimize the product $d_{i-1}d_id_{i+1}$ and repeat on the reduced instance that has M_iM_{i+1} as a single matrix, until there is only one matrix.

My counterexample is:

$$d =< 25,100,20,20 >$$

This greedy algorithm suggest that we begin with the common dimension d_2, since

$$d_1d_2d_3 = 40{,}000 < d_0d_1d_2 = 50{,}000$$

This choice produces a parenthesization $(A_1(A_2A_3))$ resulting in

$$(100 \times 20 \times 20) + (25 \times 100 \times 20) = 40{,}000 + 50{,}000 = 90{,}000$$

multiplications. However, the optimum parenthesization is $((A_1A_2)A_3)$, which results in

$$(25 \times 100 \times 20) + (25 \times 20 \times 20) = 50{,}000 + 10{,}000 = 60{,}000$$

multiplications.

(d) First multiply M_i and M_{i+1}that maximize the product $d_{i-1}d_id_{i+1}$ and repeat on the reduced instance that has M_iM_{i+1} as a single matrix, until there is only one matrix.

As in part (a), the example on p. 332 in our textbook provides a counterexample:

$$d =< 10{,}100{,}5{,}50 >$$

This greedy algorithm suggest that we begin with the common dimension d_2, since

$$d_1d_2d_3 = 25{,}000 > d_0d_1d_2 = 5{,}000$$

This choice produces a parenthesization $(A_1(A_2A_3))$ resulting in

$$(100 \times 5 \times 50) + (10 \times 100 \times 50) = 25{,}000 + 50{,}000 = 75{,}000$$

multiplications. However, the optimum parenthesization is $((A_1A_2)A_3)$, which results in

$$(10 \times 100 \times 5) + (10 \times 5 \times 50) = 5{,}000 + 2{,}500 = 7{,}500$$

multiplications.

3. *Purpose: reinforce your understanding dynamic programming.*
(3 points) Do problem 15-4 on page 367.

Planning a company party

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; 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 described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that 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.

Each node has a conviviality rating, call it c. We will associate 2 additional numbers with each node (including the leaves). The first number is the maximum conviviality rating for the sub-tree rooted at the given node which can be achieved by inviting the guest in that node (call this number i), and the second number is the maximum rating that can be achieved by excluding the guest in that node (call this number e).

The i value for a node is the c rating of that node plus the e value for all its child nodes (since the children nodes must be excluded if the given node is include).

$$i_n = c_n + \sum_{child\ nodes} e$$

The e value for a node is the sum of the maximum of i and e for each of its children, since the guests represented by the child nodes may be either included or excluded.

$$e_n = \sum_{child\ nodes} \max(i, e)$$

For leaf nodes, this formula gives $i = c$ and $e = 0$. We work our way recursively down the tree, starting with the root, calculating $\{i, e\}$ pairs for all nodes. For the root, if $i \geq e$, then the president is invited (in the case of equality, we give the nod to the president, since he/she is writing the checks). Otherwise, the president is not invited. We work our way back down the tree, selecting the optimum values at each node. For a given node, the guest is invited if $i \geq e$ and the guest's supervisor is not invited; the guest is otherwise not invited.

Pseudocode implementation:

We assume the following functions are available for a node:

$c(node)$ is the conviviality ranking for the employee represented by this node.

$i(node)$ is the best possible conviviality rating for sub-tree rooted at this node, with the node included in the invitation list; assumed to be -1 on initialization

$e(node)$ is the best possible conviviality rating for sub-tree rooted at this node, with the node excluded from the invitation list; assumed to be -1 on initialization

$\lambda(node)$ is a calculated value indicating whether the employee represented by this node is invited; **true** if $i(node) \geq e(node)$; otherwise **false**.

$N(node)$ is name of the employee associated with this node.

LEFT_CHILD(*node*), RIGHT_SIBLING(*node*), PARENT(*node*) – relatives of this node

```
OPTIMIZE_CONVIVIALITY(root)
    ▷ root is the node for the company president
    CALCULATE_CONVIVIALITY_FOR_NODE(root)
    ▷ is root invited?
    if i(root) ≥ e(root)
        then λ(root) ← i(root)
        else λ(root) ← e(root)

    PRINT_INVITATIONS(root)
end

CALCULATE_CONVIVIALITY_FOR_NODE(node)
    i(node) ← c(node)
    e(node) ← 0
    nextChild ← LEFT_CHILD(node)
    while (nextChild <> null)
        CALCULATE_CONVIVIALITY_FOR_NODE(nextChild)
        i(node) ← i(node) + e(nextChild)
        e(node) ← e(node) + Max{i(nextChild), e(nextChild)}
        nextChild ← RIGHT_SIBLING(nextChild)
end

PRINT_INVITATIONS(node)
    if λ(node)then
        PRINT N(node) "is invited"

    nextChild ← LEFT_CHILD(node)
    while (nextChild <> null)
        if λ(PARENT(nextChild)) = true      ▷ this node's supervisor is invited
            then λ(nextChild) ← false
        ▷ supervisor not invited, so this node can be invited or not invited
        else if i(nextChild) ≥ e(nextChild)
            then λ(nextChild) ← true
            else λ(nextChild) ← false
end
```

Proof of correctness

This problem has an optimal sub-structure. Consider an arbitrary node X. Node X must be excluded from the invitation list if the parent of X is included, by the problem statement. The optimal sub-tree originating from X in this case must exclude X. On the other hand, if the parent of node X is excluded, then there are two possible optimal sub-trees to consider, one which includes X and another which excludes X. The algorithm selects whichever sub-tree has the best conviviality rating.

Suppose we have a solution which we believe is optimal. If there is a node in the solution with sub-tree with a better conviviality rating, and is compatible with its parent, then we would replace that sub-tree with the new one and obtain a higher total conviviality rating. This is the optimal sub-structure.

Running time analysis

Let $N = number\ of\ employees$. The subroutine CALCULATE_CONVIVIALITY_FOR_NODE visits each node exactly once. Each node will be involved in 1 addition to calculate i for its parent, as well as 1 addition and 1 comparison to calculate e for its parent. Since the root of the tree has no parent, there are $2(N-1)$ additions and $N-1$ comparisons.

PRINT_INVITATIONS visits each node exactly once. The time consuming step is the actual PRINT statement, which is executed N times.

Thus the algorithm is a combination of $\Theta(N)$ algorithms, and is therefore itself $\Theta(N)$.

Please see Appendix A for an implemenation of this algorithm in C#.

4. *Purpose: reinforce your understanding of greedy algorithms.*
(3 points) Do problem 16-2(a) on page 402.

Scheduling to minimize average completion time

Suppose you are give a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where $a_i$requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5, c_1 = 8$, and the average completion time is $(5+8)/2 = 6.5$.

a. Give an algorithm that schedules tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i is started, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

This problem has an optimal substructure. Suppose $\{a_k', a_{k+1}', \dots, a_l'\}$ is a sub-schedule of a purported optimum schedule. If the completion time for the sub-schedule $\sum_{i=k}^{l} c_i'$ is not a minimum, then the sub-schedule can be replaced with a new sub-schedule such that the completion time is minimized. Therefore, each sub-schedule must have minimal completion time.

Consider a 2 item sub-schedule $S_k = \{a_k, a_{k+1}\}$. The average completion time for this sub-schedule is

$$C_k = 1/2[p_k + (p_k + p_{k+1})] = p_k + p_{k+1}/2.$$

Reversing the order would yield an average completion time of

$$C_{k+1} = 1/2[p_{k+1} + (p_{k+1} + p_k)] = p_{k+1} + p_k/2.$$

The optimum schedule is determined by the relation between C_k and C_{k+1}:

$$C_k - C_{k+1} = \frac{p_k - p_{k+1}}{2}, which\ is \begin{cases} \leq 0, p_k \leq p_{k+1} \\ > 0, p_k > p_{k+1} \end{cases}$$

Therefore

$$C_k \leq C_{k+1}\ when\ p_k \leq p_{k+1}\ and\ C_k > C_{k+1} when\ p_k > p_{k+1}$$

If $C_k > C_{k+1}$then sub-schedule S_k should be replaced by S_{k+1}. Thus, for each 2 element sub-schedule in the solution,

$$S_i = \{a_i', a_{i+1}'\},\ p_i' \leq p_{i+1}'$$

The condition $p_i' \leq p_{i+1}'$ is simply the condition that array p' is sorted in ascending order.

Therefore, the optimum schedule is one for which tasks are scheduled such that the corresponding array p' is sorted in ascending order.

We simply sort the keys (array p) in ascending order, and the schedule is the corresponding array of tasks a. Of course, the sorting problem has been studied extensively in this course. The worst-case running time for a comparison sort has been shown to be $\Omega(n \log n)$ (p. 167 in our textbook). If we use MERGE-SORT, the running time is $\Theta(n \log n)$.

Appendix A. Sample application of the algorithm for Problem 3 using the Tree illustrated in Fig. 10.10 in the textbook.

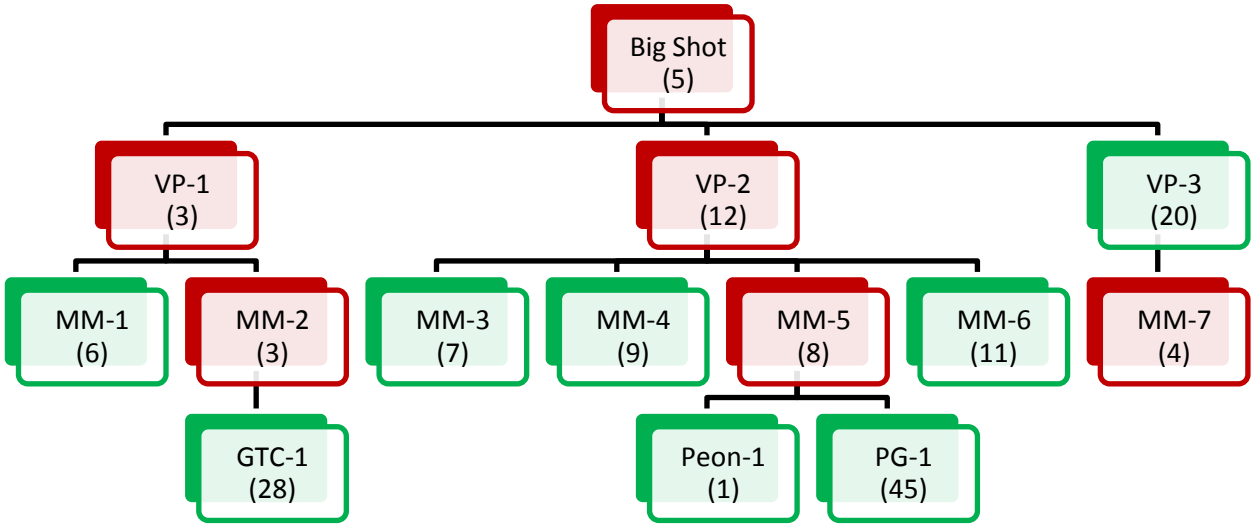

Fig. A1: Hierarchical conviviality tree modeled after the left-child, right-sibling representation illustrated in Fig 10.10 in our textbook (pointers omitted).

Key: Big Shot – President, VP – Vice President, MM – Middle Management, GTC – Good Time Charlie, PG – Party Girl

Green nodes identify employees who are invited; red nodes identify uninvited employees.

The algorithm was implemented in C# (source code and output follows).

File Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConvivialityTree
{
    class Program
    {
        static void Main(string[] args)
        {
            TreeNode[] nodes = new TreeNode[14];
            // President
            nodes[0] = new TreeNode("BigShot", 5, null);
            // Vice Presidents
            nodes[1] = new TreeNode("VP-1", 3, nodes[0]);
            nodes[2] = new TreeNode("VP-2", 12, nodes[0]);
            nodes[3] = new TreeNode("VP-3", 20, nodes[0]);
            // Middle Management
            nodes[4] = new TreeNode("MM-1", 6, nodes[1]);
            nodes[5] = new TreeNode("MM-2", 3, nodes[1]);
            nodes[6] = new TreeNode("MM-3", 7, nodes[2]);
            nodes[7] = new TreeNode("MM-4", 9, nodes[2]);
            nodes[8] = new TreeNode("MM-5", 8, nodes[2]);
            nodes[9] = new TreeNode("MM-6", 11, nodes[2]);
            nodes[10] = new TreeNode("MM-7", 4, nodes[3]);
            // Peons
            nodes[11] = new TreeNode("GTC-1", 28, nodes[5]);
            nodes[12] = new TreeNode("Peon-1", 1, nodes[8]);
            nodes[13] = new TreeNode("PG-1", 45, nodes[8]);
            // Left Child assignments
            nodes[0].LeftChild = nodes[1];
            nodes[1].LeftChild = nodes[4];
            nodes[2].LeftChild = nodes[6];
            nodes[3].LeftChild = nodes[10];
            nodes[5].LeftChild = nodes[11];
            nodes[8].LeftChild = nodes[12];
            // Right Sibling assignments
            nodes[1].RightSibling = nodes[2];
            nodes[2].RightSibling = nodes[3];
            nodes[4].RightSibling = nodes[5];
            nodes[6].RightSibling = nodes[7];
            nodes[7].RightSibling = nodes[8];
            nodes[8].RightSibling = nodes[9];
            nodes[12].RightSibling = nodes[13];

            calculateConvivialityForNode(nodes[0]);
            // is President (root node) invited?
            nodes[0].Invited = (nodes[0].InclusiveSum >= nodes[0].ExclusiveSum);
            printInvitations(nodes[0]);
        }

        private static void calculateConvivialityForNode(TreeNode treeNode)
        {
            treeNode.InclusiveSum = treeNode.ConvivialityRanking;
            treeNode.ExclusiveSum = 0;
            TreeNode nextChild = treeNode.LeftChild;

            while (nextChild != null)
            {
                calculateConvivialityForNode(nextChild);
                treeNode.InclusiveSum += nextChild.ExclusiveSum;
                treeNode.ExclusiveSum += Math.Max(nextChild.InclusiveSum,
nextChild.ExclusiveSum);
                nextChild = nextChild.RightSibling;
            }
        }

        private static void printInvitations(TreeNode treeNode)
```

```
        {
            Console.WriteLine("{0} is {1}invited. (I = {2}, X = {3})",
                treeNode.EmployeeName, treeNode.Invited == true ? "" : "not ",
                treeNode.InclusiveSum, treeNode.ExclusiveSum);

            TreeNode nextChild = treeNode.LeftChild;
            while (nextChild != null)
            {
                if (nextChild.Parent.Invited == true)
                {
                    nextChild.Invited = false;
                }
                else
                {
                    nextChild.Invited = (nextChild.InclusiveSum >= nextChild.ExclusiveSum);
                }
                printInvitations(nextChild);
                nextChild = nextChild.RightSibling;
            }
        }
    }
}
```

File TreeNode.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConvivialityTree
{
    class TreeNode
    {
        private TreeNode parent = null;
        private TreeNode leftChild = null;
        private TreeNode rightSibling = null;
        private string employeeName;

        private double convivialityRanking = -1;
        // best conviviality rating for this subtree if this guest is invited
        private double inclusiveSum = -1;
        // best conviviality rating for this subtree if this guest is not invited
        private double exclusiveSum = -1;

        // is this guest invited?
        bool? invited;

        public TreeNode(string employeeName, double convivialityRanking, TreeNode parent)
        {
            this.employeeName = employeeName;
            this.parent = parent;
            this.convivialityRanking = convivialityRanking;
        }

        public string EmployeeName
        {
            get { return employeeName; }
            set { employeeName = value; }
        }

        public double InclusiveSum
        {
            get { return inclusiveSum; }
            set { inclusiveSum = value; }
        }

        public double ExclusiveSum
        {
            get { return exclusiveSum; }
            set { exclusiveSum = value; }
        }

        internal TreeNode Parent
        {
            get { return parent; }
            set { parent = value; }
        }

        internal TreeNode LeftChild
        {
            get { return leftChild; }
            set { leftChild = value; }
        }


        internal TreeNode RightSibling
        {
            get { return rightSibling; }
            set { rightSibling = value; }
        }
        public double ConvivialityRanking
        {
            get { return convivialityRanking; }
```

```
            set { convivialityRanking = value; }
        }

        public bool? Invited
        {
            get { return invited; }
            set { invited = value; }
        }
    }
}
```

Output:

```
BigShot is not invited. (I = 116, X = 127)
VP-1 is not invited. (I = 31, X = 34)
MM-1 is invited. (I = 6, X = 0)
MM-2 is not invited. (I = 3, X = 28)
GTC-1 is invited. (I = 28, X = 0)
VP-2 is not invited. (I = 58, X = 73)
MM-3 is invited. (I = 7, X = 0)
MM-4 is invited. (I = 9, X = 0)
MM-5 is not invited. (I = 8, X = 46)
Peon-1 is invited. (I = 1, X = 0)
PG-1 is invited. (I = 45, X = 0)
MM-6 is invited. (I = 11, X = 0)
VP-3 is invited. (I = 20, X = 4)
MM-7 is not invited. (I = 4, X = 0)
```