CSE 3101 Design and Analysis of Algorithms
Solutions for Practice Test for Unit 2
Recursion

Jeff Edmonds

1. We discuss two different purposes and definitions of "size". The running time of an algorithm is defined to be a mapping from the size of the input instance to the running time. Here "size" is the number of bits to represent the instance. If the instance is $\langle n, m \rangle$ then size is $s = \log n + \log m$. In the *friends* level of abstracting recursion, you can give your friend any legal instance that is "smaller" than yours according to some measure of "size". You must also solve on your own any instance that is sufficiently small according to this same definition of size. Here you can design your definition of "size" any way you like. Note that the number of bits to represent $\langle n-1, 2m \rangle$ is bigger than the number of bits to represent $\langle n, m \rangle$. Is this then a fair subinstance to give your friend? Maybe if you define size differently. Which of the following recursive algorithms has been designed correctly? If so what is your measure of the size of the instance? On input instance $\langle n, m \rangle$, either bound the depth to which the algorithm recurses as a function of n and m or prove that there is at least one path down the recursion tree that is infinite.

algorithm $R_a(n, m)$
$\langle$***pre−cond***$\rangle$: n & m ints.
$\langle$***post−cond***$\rangle$: Say Hi

```
begin
    if(n ≤ 0)
        Print("Hi")
    else
        R_a(n − 1, 2m)
    end if
end algorithm
```

algorithm $R_b(n, m)$
$\langle$***pre−cond***$\rangle$: n & m ints.
$\langle$***post−cond***$\rangle$: Say Hi

```
begin
    if(n ≤ 0)
        Print("Hi")
    else
        R_b(n − 1, m)
        R_b(n, m − 1)
    end if
end algorithm
```

algorithm $R_c(n, m)$
$\langle$***pre−cond***$\rangle$: n & m ints.
$\langle$***post−cond***$\rangle$: Say Hi

```
begin
    if(n ≤ 0 or m ≤ 0)
        Print("Hi")
    else
        R_c(n − 1, m)
        R_c(n, m − 1)
    end if
end algorithm
```

algorithm $R_d(n, m)$
$\langle$***pre−cond***$\rangle$: n & m ints.
$\langle$***post−cond***$\rangle$: Say Hi

```
begin
    if(n ≤ 0 or m ≤ 0)
        Print("Hi")
    else
        R_d(n − 1, m + 2)
        R_d(n + 1, m − 3)
    end if
end algorithm
```

algorithm $R_e(n, m)$
$\langle$***pre−cond***$\rangle$: n & m ints.
$\langle$***post−cond***$\rangle$: Say Hi

```
begin
    if(n ≤ 0 or m ≤ 0)
        Print("Hi")
    else
        R_e(n − 4, m + 2)
        R_e(n + 6, m − 3)
    end if
end algorithm
```

- Answer:
  (a) One might complain that if my instance is $\langle n, m \rangle$ then my friend's instance cannot be $\langle n-1, 2m \rangle$, because $2m$ is not "smaller" then m. However, we can define the "size" of instance $\langle n, m \rangle$ to be simply n. According to this measure, my friend's instance is indeed smaller. More over when the instance becomes of size zero or smaller, then $n \leq 0$ and the recursion stop. We prove that the depth of recursion is at most n as follows. On instance $\langle n, m \rangle$, the size starts at n and decreases by at least one every level of recursion, so after n levels the size is at most zero and the algorithm stops recursing further. For example, starting with $\langle 5, 2 \rangle$, it recurses on $\langle 4, 4 \rangle$, $\langle 3, 8 \rangle$, $\ldots$ $\langle 0, 64 \rangle$, and then halts.
  (b) One might claim that all is well because both friends get instances ($\langle n-1, m \rangle$ and $\langle n, m-1 \rangle$) that are smaller. However, for this to be true for both friends, then size must be something

like $n + m$. However, according to this definition, the instance $\langle 5, -5\rangle$ is small but the algorithm does not halt. There is a path down this recursive tree that is infinite, namely $\langle n, m\rangle$, $\langle n, m-1\rangle$, $\langle n, m-2\rangle$, $\ldots \langle n, 1\rangle$, $\langle n, 0\rangle$, $\langle n, -1\rangle$, $\langle n, -2\rangle \ldots$.

(c) Here the size of the instance $\langle n, m\rangle$ can be defined to be $n + m$. According to this measure, each friend is given a smaller instance. More over if the size on the instance is zero then either $n \leq 0$ or $m \leq 0$. Either way the program halts. The depth of recursion can be at most $n + m$ because this is the initial size and the size decreases by one each iteration.

(d) Let the size of the instance $\langle n, m\rangle$ be $5n + 2m$. Then the first friend's instance $\langle n - 1, m + 2\rangle$ has size $5(n-1)+2(m+2) = 5n+2m-1$, which is one smaller. The second friend's instance $\langle n + 1, m - 3\rangle$ has size $5(n + 1) + 2(m - 3) = 5n + 2m - 1$, which is also one smaller. More over if the size on the instance is zero then either $n \leq 0$ or $m \leq 0$. Either way the program halts. The depth of recursion can be at most $5n + 2m$ because this is the initial size and the size decreases by one each iteration.

(e) I claim that there is a path down this recursion tree that is infinite. If my instance is $\langle n, m\rangle$, then my first friend has $\langle n - 4, m + 2\rangle$, his first friend has $\langle n - 8, m + 4\rangle$, his first friend has $\langle n - 12, m + 6\rangle$, his second friend has $\langle n - 6, m + 3\rangle$, and his second friend has $\langle n, m\rangle$ which is the same as my instance. This can be repeated infinitely often.
It is interesting that the last two examples can be generalized to the friend's instances be of size $\langle n - a, m + b\rangle$ and $\langle n + c, m - d\rangle$. If $ad > bc$ then the program halts, else it does not.

2. Review the problem on Iterative Cake Cutting. (Section 2.3) You are now to write a recursive algorithm for the same problem. You will, of course, need to make the pre and post conditions more general so that when you recurse, your subinstances meet the preconditions. Similar to moving from insertion sort to merge sort, you need to make the algorithm faster by cutting the problem in half.

(a) You will need to generalize the problem so that the subinstance you would like your friend to solve is a legal instance according to the preconditions and so that the post conditions states the task you would like him to solve. Make the new problem, however, natural. Do not, for example, pass the number of players n in the original problem or the level of recursion. The input should simply be a set of players and a sub-interval of cake. The post condition should state the requirements on how this subinterval is divided among these players. To make the problem easier, assume that the number of players is $n = 2^i$ for some integer i.

(b) Give recursive pseudo code for this algorithm. As a big hint, towards designing a recursive algorithm, we will tell you the first things that the algorithm does. Each player specifies where he would cut if he were to cut the cake in half. Then one of these spots is chosen. You need to decide which one and how to create two subinstances from this.

(c) Be sure to determine the running time.

(d) Formally prove all the steps similar to those for an iterative algorithm needed to prove this recursive program correct.

(e) Now suppose that n is not 2^i for some integer i. How would we change the algorithm so that it handles the case when n is odd? I have two solutions. One which modifies the recursive algorithm directly and one that combines the iterative algorithm and the recursive algorithm. You only need to do one of the two (as long as it works and does not increase the bigoh of the running time.)

- Answer:

(a) See below.

(b)

algorithm $Partition(P, [a, b])$

⟨pre−cond⟩: P is a set of players. (For now assume that $|P| = 2^i$ for some integer i). $[a, b] \subseteq [0, 1]$ is a subinterval of the whole cake.

⟨post−cond⟩: Returns a partitioning of $[a, b]$ into $|P|$ disjoint pieces $[a_i, b_i]$ so that for each $i \in P$, the player p_i values $[a_i, b_i]$ at least $\frac{1}{|P|}$ as much as he values $[a, b]$.

begin
if($|P| = 1$) then return("allocate $[a, b]$ to the single player in P")
loop $i \in P$
$v_i = Eval(p_i, [a, b])$
$c_i = Cut(p_i, a, \frac{v_i}{2})$
end loop
$c_{mid} = median(c_i)$
Partition players P into P_{left} and P_{right} so that
$|P_{left}| = |P_{right}| = \frac{|P|}{2}$
$\forall i \in P_{left},\ c_i \leq c_{mid}$
$\forall i \in P_{right},\ c_{mid} \leq c_i$
return($Partition(P_{left}, [a, c_{mid}]) \cup Partition(P_{right}, [c_{mid}, b])$)
end algorithm

(c) Give and solve the recurrence relation for the Running Time of this algorithm: If you spend $n \log n$ time sorting the c_i, then the recurrence relation is $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n \log n) = \mathcal{O}(n \log^2 n)$, which is too much. However, the median can be found in $\mathcal{O}(n)$ time. The notes even give a deterministic algorithm for it. This gives $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n \log n)$, just like for merge sort.

(d) Formal proof steps:

i. Prove that if your instance meets the preconditions, then your two subinstances also meet the preconditions: Clearly, each subinstance is given a subinterval of the whole cake. If $|P|$ is a power of 2 and $|P|$ is partitioned in half then the number of players given to each subinstance is a power of two. The only slightly tricky thing is to make sure that this is possible. Sort the elements in P according to c_i. $|P|$ being even, there is not an exact middle player. Let p_{mid} be the player to the left of the middle, let P_{left} consist of p_{mid} and everyone to the left of him, and let P_{right} consist of everyone to the right of p_{mid}. By this more detailed construction, it is more clear that the three conditions of the partition are met.

ii. Prove that the subinstances that you give to you friends are in some way "smaller" than your instance: There are fewer players being considered.

iii. Prove that if your friend's solutions meet the postconditions, then your solution meets the postcondition: If the friends return a partitioning of $[a, c_{mid}]$ and $[c_{mid}, b]$ into $|P_{left}|$ and $|P_{right}|$ disjoint pieces each, then we return a partitioning of $[a, b] = [a, c_{mid}] \cup [c_{mid}, b]$ into $|P| = |P_{left}| + |P_{right}|$ disjoint pieces. Consider a player $p_i \in P_{left}$. Our left friend allocates him a pieces $[a_i, b_i]$ that he values at least $\frac{1}{|P_{left}|} = \frac{2}{|P|}$ as much as he values $[a, c_{mid}]$. Because $c_i \leq c_{mid}$, he values $[a, c_{mid}]$ at least as much as $[a, c_i]$. Because he cut $[a, b]$ in half at c_i, he values $[a, c_i]$ at least $\frac{1}{2}$ as much as he values $[a, b]$. Combining these, he values $[a_i, b_i]$ at least $\frac{1}{|P|}$ as much as he values $[a, b]$. The case for a right player is similar.
Note that splitting the cake at a random c_i would not work because if you give your friend one too many players than they will not each be given enough.

iv. Prove that your solution for the base case meets the postconditions. If $|P| = 1 = 2^0$, then allocating all of $[a, b]$ to this one player, partitions $[a, b]$ into $|P| = 1$ disjoint pieces so this one player values his piece $[a, b]$ at least $\frac{1}{|P|} = 1$ as much as he values $[a, b]$.

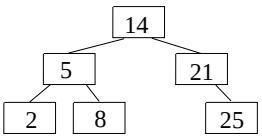
(e) Here are two solutions when n is not a power of 2.

i. Instead of everyone marking their half point c_i, each player marks the point of value $\frac{n-1}{2n}$. Then the players are partitioned so that $\frac{n-1}{2}$ of the players are happy recursing on the left and $\frac{n+1}{2}$ are happy recursing on the right. Because the left is worth at least $\frac{n-1}{2n}$ to a player on the left and recursively he gets at least $1/\frac{n-1}{2}$ of the left, it follows that he gets $\frac{n-1}{2n} \times 1/\frac{n-1}{2} = \frac{1}{n}$ of the cake. Because the right is worth at least $\frac{n+1}{2n}$ to a player on the right and recursively he gets at least $1/\frac{n+1}{2}$ of the left, it follows that he gets $\frac{n+1}{2n} \times 1/\frac{n+1}{2} = \frac{1}{n}$ of the cake.

ii. If n is odd, you can do one step of the iterative algorithm so that there are only $n-1$ players left. Because $n-1$ is even, you can partition these players evenly in half and recurse. The work you do is $\mathcal{O}(n)$ for the one iterative step and $\mathcal{O}(n)$ for the portioning for a total of $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

3. Develop an algorithm that searches for a key within a binary search tree.

- Answer: Search Binary Search Tree: This problem searches for a key within a given binary search tree.
Recall that in a *binary search tree* the nodes are ordered such that for each node all the elements in its left subtree are smaller than its own element and all those in the right are larger. The following recursive algorithm directly mirrors the iterative algorithm.



algorithm $SearchBST(tree, keyToFind)$

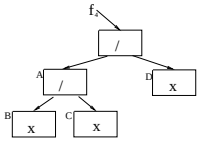
⟨*pre−cond*⟩: *tree* is a binary tree whose nodes contain key and data fields. $keyToFind$ is a key.

⟨*post−cond*⟩: If there is a node with this key is in the tree, then the associated data is returned.

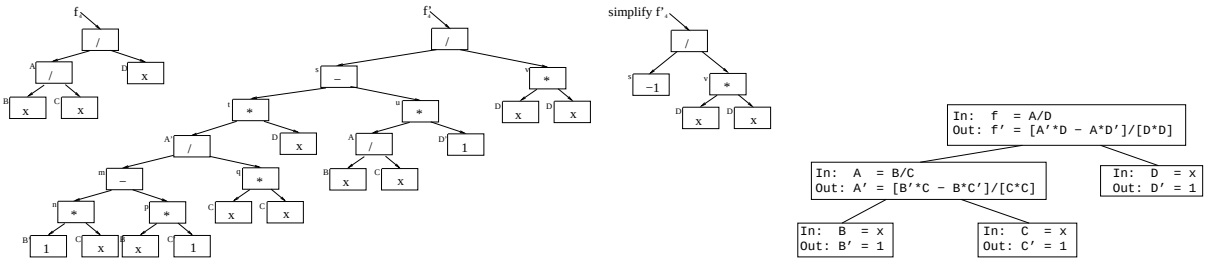
```
begin
    if( tree = emptyTree ) then
        result "key not in tree"
    else if( keyToFind < rootKey(tree) ) then
        result( SearchBST(leftSub(tree), keyToFind) )
    else if( keyToFind = rootKey(tree) ) then
        result( rootData(tree) )
    else if( keyToFind > rootKey(tree) ) then
        result( SearchBST(rightSub(tree), keyToFind) )
    end if
end algorithm
```

4. Trace out the execution of Derivative on the instance

$f = (x/x)/x$. In other words, draw a tree with a box for each time a routine is called. For each box, include only the function f passed and derivative returned.



- Answer:



5. Recursion GCD: Write a recursive program to find the GCD of two numbers. The program should mirror the iterative algorithm.

- Answer: Given the integers a and b, the iterative algorithm creates two numbers $x = b$ and $y = a \ mod \ b$. It notes that $GCD(a, b) = GCD(x, y)$, and hence it can return $GCD(x, y)$ instead of $GCD(a, b)$. This algorithm is even easier when you have a friend. We simply give the subinstance $\langle x, y \rangle$ to the friend and he computes $GCD(x, y)$ for us. For the iterative algorithm, we need to make sure we are "making progress" and for the recursive algorithm, we need to make sure that we give the friend a "smaller" instance. Either way, we make sure that in some way $\langle x, y \rangle$ is smaller than $\langle a, b \rangle$. For the iterative algorithm, we need an exit condition that we are sure to eventually meet and for the recursive algorithm, we need bases cases such that every possible instance is handled. Either way, we consider the case when y or b is zero. The resulting code is

  algorithm $GCD(a, b)$
  $\langle \boldsymbol{pre-cond} \rangle$**:** a and b are integers.
  $\langle \boldsymbol{post-cond} \rangle$**:** Returns $GCD(a, b)$.

  ```
  begin
      if(b = 0) then
          return( a )
      else
          return( GCD(b, a mod b) )
      end if
  end algorithm
  ```

  We get the recurrence relation as follows. There is one friend so $a = 1$.
  (Note not the a and b from the code.)
  The value goes down by a factor of two every other iteration, so lets say that the size $n = \#$ of bits gets smaller by half a bit to $n - b = n - \frac{1}{2}$.
  Note that this is an additive difference not a multiplicative difference so we use the bottom chart of unit 0 steps.
  I personally do $f(n) = \Theta(1)$ mod operations.
  The recurrence relation is
  $T(n) = a \cdot T(n - b) + f(n) = 1 \cdot T(n - \frac{1}{2}) + \Theta(1)$.
  Because $a = 1$, the table gives
  $T(n) = \Theta(n \cdot f(n)) = \Theta(n)$,
  but it is easy to solve on our own.
  $T(n) = T(n - \frac{1}{2}) + 1 = T(n - 2 \cdot \frac{1}{2}) + 2 = T(n - 3 \cdot \frac{1}{2}) + 3 = T(n - r \cdot \frac{1}{2}) + r = T(0) + 2n$.
  A theory person would instead say that we do not one mod operation but $f(n) = \Theta(n)$ bit operations.
  Then the time is $T(n) = \Theta(n^2)$.

6. Recursion Smallest: Explain the recursive algorithm given in class for finding the k^{th} smallest element in an array. Explain what the running time is.

   - Answer: Ignoring input k, a pivot element is chosen randomly, and the list is split into two sublists, the first containing all elements that are all less than or equal to the pivot element and the second those that are greater than it. Let n_{left} be the number of elements in the first sublist. If $n_{left} \geq k$, then we know that the k^{th} smallest element from the entire list is also the k^{th} smallest element from the first sublist. Hence, we can give this first sublist and this k to a friend and ask him to find it. On the other hand, if $n_{left} < k$, then we know that the k^{th} smallest element from the entire list is the $(k - n_{left})^{th}$ smallest element from the second sublist. Hence, giving the second sublist and $k - n_{left}$ to a friend, he can find it.

     In a little bad but still random looking case, the recurrence relation is $T(n) = 1T(\frac{2}{3}n) + \Theta(n)$.
     The number of base cases is $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 1}{\log 3/2}}) = \Theta(n^0) = 1$. The work at the top is $\Theta(n^c) = \Theta(n^1)$. Note $1 > 0$. Hence, the time is dominated by the top level giving $T(n) = \Theta(n)$.

7. Recursion Smallest in Tree: Our task now is the same as the last question, except that the input is a binary search tree and an integer k. Recall that in a binary search tree all the values smaller than the root are already in the left subtree and all those larger are in the right. As before, it returns the k^{th} smallest element in the tree. Explain what the running time is.

   - Answer: The same basic algorithm will be used. We won't, however, have to choose a pivot element or partition the nodes because the root of the tree already splits the nodes in this way. The algorithm without change would then be as follows. Let n_{left} denote the number of elements in the left subtree. If $k \leq n_{left}$, then we give our left friend our left subtree and the same k. He returns the answer. If $k = n_{left} + 1$, then the answer is the root. If $n_{left} + 1 < k \leq n$, then we give our right friend our right subtree with the new $k_{right} = k - n_{left} - 1$.

     The problem is that this algorithm needs to know n_{left} before it starts. We could call an extra program to count this number but this would take to much extra time. Hence, we simply get our left friend to return this information as well. To do this we need to change the post conditions so that the number of nodes in the tree are also returned. Once this is done, we and our right friend must do the same. In the above algorithm, only one of the two friends is called. However, because both must now return the number of nodes, both must be called. One might be tempted compare k to n_{left} as soon as it is found, but the left and right friends might as well be called first. No harm done if their k is out of their range. At the end, one can test whether the left, root, or right answer is correct.

     If the tree is empty, then the element being looked for is clearly not there and the number of elements is zero.

     algorithm $Smallest(tree, k)$

     ⟨*pre−cond*⟩: $tree$ is a binary search tree and k is an integer.

     ⟨*post−cond*⟩: Outputs the k^{th} smallest element s and the number of elements n.
     If this index is out of range, we output $s = NotPossible$.

```
begin
    if( tree = emptyTree ) then
        result( ⟨NotPossible, 0⟩ )
    else
        ⟨s_l, n_l⟩ = Smallest(leftSub(tree), k)
        % There are n_l + 1 nodes before the right subtree
        ⟨s_r, n_r⟩ = Smallest(rightSub(tree), k − (n_l + 1))
        n = n_l + 1 + n_r
        if( k ∈ [1..n_l] )then
            s = s_l
        elseif( k = n_l + 1 )then
            s = root(tree)
        elseif( k ∈ [n_l + 2..n] )then
            s = s_r
        else then
            s = OutOfRange
        endif
        result( ⟨s, n⟩ )
    end if
end algorithm
```

Here is another version that only calls to the right if necessary. Note that the post condition changes.

algorithm $Smallest(tree, k)$

$\langle$***pre−cond***$\rangle$**:** $tree$ is a binary search tree and $k > 0$ is an integer.

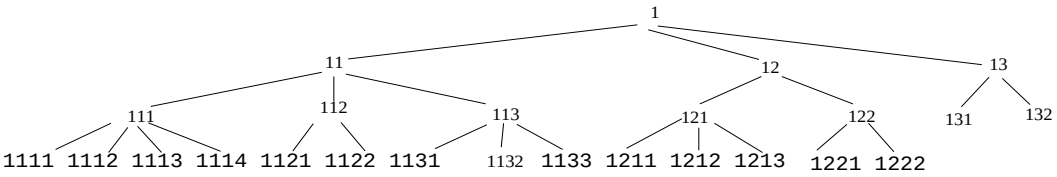
$\langle$***post−cond***$\rangle$**:** Outputs the k^{th} smallest element s.

If k is more than the number of elements in the tree, then we output $s =''$ $toobig''$ and the number of elements n is also returned.

```
begin
    if( tree = emptyTree ) then
        result( ⟨toobig, 0⟩ )
    ⟨s_l, n_l⟩ = Smallest(leftSub(tree), k)
    if( s_l ≠ toobig ) then
        result( ⟨s_l, ?⟩ )
    % Because s_l = toobig, we know that n_l is the number of nodes in the left subtree
    if( k = n_l + 1 ) then
        result( ⟨root(tree), ?⟩ )
    ⟨s_r, n_r⟩ = Smallest(rightSub(tree), k − (n_l + 1))
    result( ⟨s_l, n_l + 1 + n_r⟩ )
end algorithm
```

The running time is still $\Theta(n)$ as it is for all recursive programs on binary trees that recurse on the both the left and right subtrees and do a constant amount of work in each stack frame.

8. The algorithm $TreeToMatrix(tree, M, i, j)$ is passed a non-empty tree $tree$, a matrix M (which is passed both in and out), and two indexes i and j. The result of the algorithm is to place the data items in the tree $tree$ into the matrix M. The data item at the root, denoted $rootData(tree)$, is placed in $M(i, j)$. Its children, denoted $child(tree, 1), child(tree, 2), \ldots, child(tree, nChild(tree))$, are placed one column to the right. Its first child is placed just to the right of the root in $M(i, j+1)$. Its other children are shifted down to make room for the subtrees rooted at the previous children. An example is given below, with the input tree on the left and the resulting matrix on the right. Be sure to document any extensions you make to the pre or post conditions. Only the commented code is required. MORE DISCUSSION ALLOWED.



i/j	1	2	3	4
1	1	11	111	1111
2				1112
3				1113
4				1114
5			112	1121
6				1122
7			113	1131
8				1132
9				1133
10		12	121	1211
11				1212
12				1213
13			122	1221
14				1222
15		13	131	
16			132	

- Answer: I put my root at $\langle i, j\rangle$. I trust my first friend to put his tree at $\langle i, j + 1\rangle$. TRUST HIM. DONT MICRO MANAGE HIM. Then I must trust my second friend must put his tree at $\langle k, j + 1\rangle$ for some strange k. But it is my job to tell him what k is. Who knows k? It depends on the first tree. In fact, we need to shift by the number of leaves in the first tree. Only my first friend knows the first tree. Hence, I trust him to tell me this information as well. But then I must put this in the post condition and also compute this information. Dont have the postcondtion something strange like "where the next item goes". It needs to be a well defined function of my input. I kept asking people, "What value k should you return, given you have the entire tree", but they did not know. I kept seeing ++i, which looked global and iterative to me. ++i in the loop over the children just increases to three.

algorithm $TreeToMatrix(tree, M, i, j)$

⟨pre−cond⟩: *tree* is a tree, M a matrix, and i and j are indexes into the matrix M.

⟨post−cond⟩: The data items in *tree* are placed in M with the root at $M(i,j)$. Returned is the number of leaves in the tree. Note that this is the number of rows used in the matrix.

```
begin
    M(i,j) = rootData(tree)
    if( nChild(tree) = 0 ) then return(1)
    nLeaves = 0
    loop r = 1...nChild(tree)
        nLeaves += TreeToMatrix(child(tree,r), M, i+nLeaves, j+1)
    return(nLeaves)
end algorithm
```

9. Recursion Inversions: Given a sequence of n distinct numbers $A = \langle a_1, a_2, \ldots, a_m \rangle$, we say that a_i and a_j are inverted if $i < j$ but $a_i > a_j$. The number of inversions in the sequence A, denoted $I(A)$, is the number of pairs a_i and a_j that are inverted. $I(A)$ provides a measure of how close A is to being sorted in increasing order. For example, if A is already sorted in increasing order, then $I(A)$ is 0. At the other extreme, if A is sorted in decreasing order, every pair is inverted, and thus $I(A)$ is $\binom{n}{2}$. Describe a divide and conquer algorithm that finds $I(A)$ in time $\Theta(nlogn)$. You should assume that the sequence A is given to us in an array such that we can test the value of a_i in unit time, for any i. Note that the obvious algorithm is to test all pairs a_i and a_j; that algorithm runs in $\Theta(n^2)$ time. Hint: you should modify merge sort to also compute $I(A)$. The work each stack frame requires an iterative algorithm. Describe this iterative algorithm using loop invariants.

   - Answer: The recursive algorithm is given an array A and returns $\langle S, I \rangle$, where S is the array sorted and I is is the number of inversions $I(A)$. When A contains at least 2 elements, the code cuts the input A into two halves A_1 and A_2 and recurses on both. Let $\langle S_1, I_1 \rangle$ and $\langle S_2, I_2 \rangle$ be their output. The code then calls an iterative algorithm that takes as input two sorted arrays S_1 and S_2 and returns $\langle S, I_{\langle 1,2 \rangle} \rangle$, where $S = merge(S_1, S_2)$ is the merging of these and $I_{\langle 1,2 \rangle}$ is $I(S_1 + S_2)$ which is the number of inversions in the array containing S_1 followed by S_2. The output of the recursive algorithm is $\langle S, I_1 + I_2 + I_{\langle 1,2 \rangle} \rangle$.

     The code for the iterative algorithm maintains two three sorted arrays S'_1, S'_2, and S' and one integer I'. The first loop invariant is that $merge(S'_1, S'_2) + S'$ is the merged array $S = merge(S_1, S_2)$ that we need to output. The second loop invariant is that $I(S'_1 + S'_2) + I'$ is the number of inversions $I_{\langle 1,2 \rangle} = I(S_1 + S_2)$ that we need to output. We establish this easily by setting $S'_1 = S_1$, $S'_2 = S_2$, $S' = \emptyset$, and $I' = 0$.

     The step is as follows. Let $a_{\langle |S'_1|,1 \rangle}$ be the last elements of S'_1 and $a_{\langle |S'_2|,2 \rangle}$ be the last elements of S'_2. There are two cases. If $a_{\langle |S'_1|,1 \rangle} > a_{\langle |S'_2|,2 \rangle}$, then because S_2 is sorted, $a_{\langle |S'_1|,1 \rangle}$ is greater than all elements in S_2. Just as in the standard merging algorithm, $a_{\langle |S'_1|,1 \rangle}$ is removed from the end of S'_1 and added to the beginning of S. This maintains the first loop invariant. The element $a_{\langle |S'_1|,1 \rangle}$ contributes $|S_2|$ to $I(S'_1, S'_2)$ because it creates an inversion with each of the elements of S_2. Hence, we maintain the second loop invariant by adding $|S_2|$ to I'.

     In the second case, $a_{\langle |S'_1|,1 \rangle} \leq a_{\langle |S'_2|,2 \rangle}$. In this case, $a_{\langle |S'_2|,2 \rangle}$ is greater or equal to all elements in S_1. Just as in the standard merging algorithm, $a_{\langle |S'_2|,2 \rangle}$ is removed from the end of S'_2 and added to the beginning of S. This maintains the first loop invariant. The element $a_{\langle |S'_2|,2 \rangle}$ contributes zero to $I(S'_1, S'_2)$ because it creates an inversion with none of the elements of S_1. Hence, we maintain the second loop invariant by adding nothing to I'.

     The exit condition is when one of S'_1 or S'_2 is empty. This will eventually occur because the sum of their lengths deceases by one each iteration.

     We establish the post condition as follows. By the exit condition, one of S'_1 or S'_2 is empty. Assume S'_1 is empty. (The other case is the same.) This gives that $merge(S'_1, S'_2)$ is simply S'_2 and hence $merge(S'_1, S'_2) + S' = S'_2 + S'$. The first loop invariant gives that this is the required output S.

Hence, the first post condition is maintained by outputting $S = S'_2 + S'$. Having S'_1 empty and S'_2 sorted gives us that the number of inversion $I(S'_1 + S'_2)$ is zero and hence $I(S'_1 + S'_2) + I' = I'$ The second loop invariant gives that this is the number of inversions $I_{\langle 1,2 \rangle}$ that we need to output. Hence, the second post condition is maintained by outputting $I_{\langle 1,2 \rangle} = I'$.