CORE EXAMINATION
Department of Computer Science
New York University
January 22, 1999

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part lasts one and one-half hours, and covers algorithms.

Basic Algorithms

Question 1

Consider an array of positive integers of length n: A[1..n] whose entries are in a strictly increasing order, i.e. $A(1) < A(2) < ... < A(n)$.

1. Write a sublinear-time algorithm to compute a "fix-point" of A, that is to say a non-zero value i such that $A(i) = i$, if such an i exists. If no such i exists, the algorithm yields 0.
2. Write down the recurrence relation for the time complexity of your algorithm and solve it.

Solution

```
Fix-point(A, L, U) returns integer:
     if L > U then return 0;
     k = Floor(U+L/2);
     if (k = A(k)) then return k;
     if (k < A(k)) then return Fix-point(A, L, k-1);
     if (k > A(k)) then return Fix-point(A, k+1, U);
end{Fix-point}
```

Time complexity:

$$\begin{aligned} T(n) &= T(n/2) + c_0, \quad \text{if } n > 1 \\ T(1) &= c_1 \end{aligned}$$

Let $n = 2^k$ and $T(n) = T(2^k) = S(k)$. Then we have

$$\begin{aligned} S(k) - S(k-1) &= c_0, \quad \text{if } k > 0 \\ S(0) &= c_1. \end{aligned}$$

Telescoping, we have

$$T(n) = S(k) = c_0 k + c_1 = c_0 \lg n + c_1 = O(\log n).$$

Note, however that if we assume that the entries of the array are strictly positive then there are two situations of interest: (1) $A(1) = 1$, in which case 1 is the desired fix-point (in fact the smallest one), or (2) $A(1) > 1$, in which case $\forall i\ A(i) > A(1) + i - 1 > i$ and the array A has no fix point (and the algorithm returns 0).

Thus the algorithm only needs to check $A(1)$ and can do so in $O(1)$ time.

Question 2

We have a binary search tree containing N elements, not necessarily balanced. There are 2 fields, $NumLeft$, and $NumRight$, at each node, that need to be set. $NumLeft$ will contain the number of elements in the left subtree rooted at that node, and similarly for $NumRight$.

1. In the language of your choice, write the declaration for the data structure or class needed to represent this tree.
2. Write an O(N) algorithm that computes all the numbers $NumLeft$ and $NumRight$. Prove informally that your algorithm has this complexity.
3. Suppose d is the depth of the tree, and that all $NumLeft, NumRight$ numbers are known. Write an algorithm that finds the k^{th} largest element in the tree in work of order d.

Solution
Sample algorithm for computing numLeft and numRight. Initially called with:

```
if (T != NULL) computeNum(T);


procedure computeNum(v: node) {

   if (v.leftChild == NULL)
      v.numLeft = 0;
   else {
      computeNum(v.leftChild);
      v.numLeft = v.leftChild.numLeft + v.leftChild.numRight + 1;
   }
```

```
if (v.rightChild == NULL)
   v.numRight = 0;
else {
   computeNum(v.rightChild);
   v.numRight = v.rightChild.numLeft + v.rightChild.numRight + 1;
}
```

Part c was ambiguous: are you looking for the kth element in RANK (e.g. kth from the bottom), or the kth LARGEST element from the top. The following algorithm uses the former (the most common interpretation), but either interpretation was accepted.

Algorithm for finding kth element in a tree, using already computed numLeft and numRight fields:

```
procedure findkth(v: node, k: int) {

  if (v.numLeft >= k)
      foundNode = findkth(v.leftChild,k)
  else if (v.numleft+1 == k)
     return (v)   // found it!
  else foundNode = findkth(v.rightChild, k- v.numLeft -1);

  return foundNode;

}
```

Question 3

Let T be a tree, such that each node v has a field $v.wt$ that holds an integer, called the weight of the node. Let us define the weight of a path in the tree as the sum of the weights of the nodes on the path.

1. Describe, using some appropriate high-level pseudo-code, an efficient algorithm that computes, for each internal node v, into some field $v.path\_wt$, the weight of the lightest path from v to a descendant leaf. Your solution should have a linear cost in the number of nodes in the tree.

2. Let G be a DAG, i.e. a directed acyclic graph. A leaf of a DAG is a node with no outgoing edges. Assume that each node of the DAG has an associated weight, as above, and define the weight of path in similar fashion.
   Explain how to modify the code from part a) so that your algorithm computes the value of $v.path\_wt$, defined as before (the lightest weight

from v to some descendant leaf), in linear time overall in the size of G. DO NOT assume that the DAG has a single root (i.e. a node with no incoming edges).

Solution
Part (1) is similar to the recursive computation in problem no.2. Part (2) is a standard depth-first DAG traversal.