CORE EXAMINATION
Department of Computer Science
New York University
May 14, 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, given this afternoon, lasts one and one-half hours, and covers algorithms.

Attempt all of the questions. Use the proper booklet for each question. Each booklet is marked with the Area and Question number, in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam. Good luck!

Programming Languages and Compilers

Question 1

For each of the following constructs, write down the code that will be generated by a typical compiler. Use your favorite assembly, or else conventional quadruples. In what follows A and B are local arrays of known size, X is a local integer variable, GX is an integer variable declared in an enclosing scope. S is a dynamic array whose size is not known at compile-time. Obj is a pointer to an object of class C, and Func is a virtual method of that class, which appears in the 7th entry of the vtable for C. First three cases use Ada syntax, last two use C++

```
(a)       A (5) := B (5);
(b)       A (I + 1) := B (I + 1);
(c)       A (GX) := 0;
(d)       for (int X = 0; X < N; X++) S[X] = 0;
(e)       Obj -> Func (X);
```

Question 2

In Java, to create a generic sorting method, we can define an interface as follows:

```
interface Comparable {
   boolean LessThan (Comparable X);
}
```

We can then write a sorting method that sorts an array of objects that implement this interface.

a) Write the body such a method (any simple sorting algorithm will do).

b) The method cannot be used to sort an array of ints. Write a wrapper class for integers, that will allow us to use the method written in a) to sort such an array.

c) In Ada or C++ write a generic procedure or template that has the same functionality as your Java method.

Question 3

Consider the following (not-quite-C) program

```
#include <stdio.h
voids main (char **argv; int argc) {
   int a = 1234a6754;
{
   printf ("the value of a is %d, a)
}
```

Assume you have a really clever compiler, that gives optimal error messages for the above pitiful attempt at C. Show the error messages that would be produced.

For each error message, indicate which component of the compiler will generate it, and describe how the compiler can recover from the error and proceed with the compilation. Write down the productions that are relevant.

Operating Systems

Question 1

The following 2-process critical section solution was invented in 1966 and was long considered to be correct. Show convincingly that the solution is not correct.

```
LOOP
    flag[i] = TRUE;
    WHILE (turn != i) DO {
       WHILE (flag[j]) DO;
       turn = i;
    }
    ...  Critical Section;
    flag[i] = FALSE;
    ...  Remainder Section;
END; /* process Pi */
```

Question 2

Allowing two portions of a process's virtual address space to map to the same set of physical pages enables different protections to be associated with each portion.

1. what additional operations does an operating system need to perform at page-fault time to provide this support?
2. How can such support be used to decrease the amount of time needed to copy a large amount of memory from one place to another? Your answer must ensure that updates to the destination are not visible in the source portion.
3. Suppose you have a system with 4 kilobyte pages, 4 bytes per word, and that the cost of a read/write per word is t_{ma}. The cost of a protection fault is t_{pf} , Given the following access pattern to memory:

   ```
   copy B pages from source saddr to destination daddr

   for (i = 0; i < K; i++)
      update the first word of page i, starting at daddr
   ```

   what is the maximum value of K (as a fraction of B) for which your copying scheme will perform better than the traditional scheme?

Part II: Basic Algorithms

Question 1

Describe a linear-time algorithm that takes a binary search tree of arbitrary shape and constructs a perfectly balanced binary search tree. Assume that extra storage is available as needed. Give details of the code, and show convincingly that the performance is linear in the number of nodes in the tree.

Solution

The solution consists in copying the values from the unbalanced tree into an array, sorted in increasing order. This is done by a single traversal of the tree copying the values from the nodes in "inorder" order, that is, left subtree, current vertex, right subtree. Next, the elements of the array are linked into a balanced tree using a recursive bisection algorithm. The value from the middle element of the array goes into the root of the tree.

The most common mistake was to try to rebalance the tree using rotations, such as are used in AVL or red-black trees. This cannot work in the required time. If the tree is very unbalanced, it can take many rotations just to put the median of the values into the root position, so the overall complexity gains a logarithmic factor.
The following is a very detailed answer to the question. Full credit did not require all of the following.
Suppose we have a data type

```
binaryTreeElement {
   binaryTreeElement* left;  // pointer to left child, NIL if none.
   binaryTreeElement* right; // pointer to right child, NIL if none.
   number             value;  // The key, or value, of this node.
 }
```

The owner of the unbalanced binary tree might or might not know the number of elements in the tree. If not, here is a procedure that does DFS (depth first search) to find the number:

```
int treeSize( binaryTreeElement v ) {
   int size = 1;  // Count this vertex.
   // If some subtree is present, count its vertices
   if ( v.left != NIL)  size += treeSize( *(v.left) );
   if ( v.right != NIL) size += treeSize( *(v.right));
   return size;
 }
```

Here is the procedure that does the "inorder" traversal of the tree to copy its elements into the array:

```
void treeToArray( binaryTreeElement root, number A[], int *next ) {

/* Copy the tree starting at root, including root into the array A
   starting at location "next".  ASSUME someone has allocated a
    large enough array.
 */

   if ( root.left != NIL )                  // If present,
      treeToArray( *(root.left), A, next ); // copy its elements to A.
   A[ (*next)++ ] = root.value;             // Next, the root,
   if ( root.right != NIL )                 // then the right.
      treeToArray( *(root.right), A, next); // if present.
 }
```

This procedure creates a new balanced binary search tree from the elements of a sorted array.

```
*binaryTreeElement arrayToBalancedTree( number A[], int n ) {

/*  Create a balanced binary search tree from the elements of A, which
    is sorted in increasing order. n is the number of elements in A */

   binaryTreeElement* root;

   //Create the root element and set it up.
   root  = new( binaryTreeElement );
   int     median, leftSubtreeSize, rightSubtreeSize;
   median           = n/2;        // rounded to (n-1)/2 if n is odd.
   *root.value = A[ median - 1 ]; // The first element of A is A[0].
   *root.left  = NIL;             // overridden later if there are subtrees.
   *root.right = NIL;

   leftSubtreeSize  = median - 1;
   if ( leftSubtreeSize > 0 )
      *root.left = arrayToBalancedTree( A, leftSubtreeSize );
   rightSubtreeSize = n - median - 1;
   if ( leftSubtreeSize > 0 )
      *root.right = arrayToBalancedTree( &A[median], rightSubtreeSize );
  }
```

Each of the procedures, `treeSize`, `treeToArray`, and `arrayToBalancedTree`, visits each element once. Therefore, the total work is $O(n)$, for the n elements.

Question 2

We want to store $N = 10^6$ (roughly 2^{20}) telephone numbers in a balanced search tree with branching factor b and height h. The tree is stored on a slow disk. Each internal node in the tree has b children, and can be read into memory in time $T = T_0 + T_1 b$, where $T_0 = 25 msec$ and $T_1 = 2 msec$. All phone numbers are stored in the leaves.

a) What is the relationship between N, b and h?

b) Suppose that b and h are much larger than one. Find an approximate expression for the maximum time taken to find a given phone number in the tree. Ignore the cost of miscellaneous arithmetic operations.

c) Use the result of part (b) to show that $b = 16$ is a better value than 4 or 100.

Solution

a) The number of leaves in a tree with height h and branching factor b is b^h. If all the numbers are stored in leaves, this means

$$N = b^h \quad .$$

Note: a tree with a single element will have height zero in this convention.

b) To get to a leaf from the root, you have to read h nodes. (This assumes that the root is always in memory, otherwise we have to read $h+1$ nodes, this does not significantly affect the result of the following computations). Each node takes $T = T_0 + h \cdot T_1$, so the total time is

$$T_{\text{tot}} = h\,(T_0 + h \cdot T_1) \quad .$$

c) We need a formula for h as a function of N and b. Take logarithms in the most convenient base of $N = b^h$ to get

$$\begin{aligned} \lg(N) &= h \cdot \lg(b) \qquad \text{or} \\ h &= \lg(N)/\lg(b) \quad . \end{aligned}$$

For $b = 4$, $\lg(b) = 2$, $\lg(N) = 20$, this gives:

$$\begin{aligned} h &= 20/2 = 10 \\ T &= 25 + 4 \cdot 2 = 33 \ msec \\ T_{\text{tot}} &= 10 \cdot 33 = 330 \ msec \quad . \end{aligned}$$

For $b = 16$, $\lg(b) = 4$, $\lg(N) = 20$, this gives:

$$\begin{aligned} h &= 20/4 = 5 \\ T &= 25 + 16 \cdot 2 = 25 + 32 = 57 \ msec \\ T_{\text{tot}} &= 5 \cdot 57 = 285 \ msec \ . \end{aligned}$$

For $b = 128 \approx 100$, $\lg(b) = 7$, $\lg(N) = 20$, this gives:

$$\begin{aligned} h &= 20/7 \approx 3 \\ T &= 25 + 128 \cdot 2 = 25 + 256 \approx 280 \ msec \\ T_{\text{tot}} &\approx 3 \cdot 280 = 560 \ msec \ . \end{aligned}$$

similar result if we take $log_{10}(N) = 6$, $log_{10}(b) = 2$ The time for $b = 16$ is the smallest of the three.

Question 3

A spreadsheet in a spreadsheet program contains an N*M array of cells. Each cell holds a numeric value. A value may be supplied directly by the user, or it may be calculated in term of values in other cells:

```
B2 = B4 * (A1 + C23 + D5)
```

By convention, rows are indexed by letter and columns by number. Thus cell B4 corresponds to the [2, 4] entry in the array. The dependency list for a cell is the list of cells its value depends on. For example, the dependency list for B2 given above is (B4, A1, C23, D5). If a cell has a constant numeric value its dependency list is empty.
Suppose you are given a method *Dependency_List (cell X)* which returns a p ointer to the first element of the dependency list of the given cell. Describe an algorithm that checks whether a given cell has a circular dependence, that is to say depends directly or indirectly on itself (such circularities are not allowed in normal usage). For example, given the previous definition for B2, if we have the additional dependencies:

```
A1 = A2 + A5
```

then B2 has a circular dependency through A1 and A2.
Your algorithm should have a complexity proportional to the number of cells on which the given cell depends (directly or indirectly) and should be independent of the total size of the spreadsheet.

Solution

The following answer was good for 9 points out of ten. A completely correct answer is more complicated.

Answer 1: The cells in the spreadsheet form the vertices of a directed graph. There is an edge from X to Y if Y is in the dependency list of X. The cell X has no circular dependence if and only if the directed graph rooted at X has no cycles. We can test this using depth-first search (DFS) starting at X. The vertices searched will be those on which X depends, either directly or indirectly. The work for this is proportional to $V + E$, where V is the number of cells on which X depends (directly or indirectly) and E is the number of dependency relations involving those cells.

Criticism of Answer 1: To do DFS, we must first initialize by marking all the vertices as not visited. If we initialize every cell in the spread sheet, the work will be $\Theta(NM)$, which is not what the question askes for.

Answer to Criticism: When we visit a node, we must check whether the node has been visited before. This can be done by putting the visited notes into a hash table. The trouble is that we do not know how large the table should be. If we make it big enough for the whole spreadsheet, it will take $\Theta(NM)$ work to initialize. A solution is to use an *expanding hash table.* Start with $S = 2$. If the number of elements in the table exceeds $S/2$, create a new table with twice the size and hash all the old entries into the new table. An amortized analysis shows that the work to insert k elements into an expanding hash table is $\Theta(k)$.

Most common error: The most common mistake was is illustrated by the following pseudocode:

```
boolean find (cell X, cellList L) {
   if (L == NIL) return FALSE;
   boolean found = FALSE;
   for (cell Y in L ) {
      if ( Y == X ) return TRUE;
      found = found or find( X, dependency_List(Y);
    }
   return found;
 }
```

This will fail on the example

```
C1 = A2 + D4
A2 = D5 + B6
D5 = E7 + A2
```

Here, `C1` has a circular dependcence that does not include `C1`. The above code would be an infinite loop going back and forth between `A2` and `D5`.

Question 4

We have two lists of words, A[N] and B[M]. We need to determine the number of words that are common to both lists. We have a method

```
boolean Equals (word X, word Y);
```

whose performance takes constant time (words all have roughly the same size). Assume that each list has no duplicates, but that many words in A also occur in B. There are two ways of approaching this problem: sorting, and hashing.

a) Assuming that storage is available as needed, which method will be faster? Explain, and discuss worst cases.

b) Give details of the program for the faster algorithm.

Solution

Let's suppose we have another method,

```
boolean Before( word X, word Y) ,
```

thar returns `TRUE` if `X` comes before `Y` in lexicographic order, and `FALSE` otherwise.

a) Sorting: Suppose that `A` is the shorter list: $N \leq M$. We sort `A` using the `Before` method and quicksort or mergesort in $O(N \log(N))$ time. In practice, it would be a packaged sort program that uses some optimized version of quicksort. Without any information on the lengths of the words, it is doubtful that radix sort will be faster. For each word in `B`, we can perform a bisection search of the sorted `A` to find whether that word is in `A`. Each such search takes $\lg(N)$ comparisons, so the total is $M \lg(N)$. The total work, then, is $O((N+M)\lg(N)) = O(M \lg(N)$. The equality is because $N \leq M$.

b) hashing: Suppose $N \leq M$. We hash the elements of `A` into a hash table of size $2N$, using, e.g. closed hasning to resolve collisions. If we have a fast and good hash function

```
int hashFunction( word X )
```

this will take $O(N)$ time in all. Now we probe the table with each word from `B` and cound the hits (not the collisions). This takes $O(M)$ time in practice. The total work is $O(N+M) = O(M)$.

Comparing these shows that *hashing* is faster. Note that if we are using quicksort, the worst case is $O(N^2)$ instead of $O(N \lg(N))$. We neve expect to see this in practice. The worst case for hashing is also $O(N^2)$, which we also do not expect to live to see.