Computer Science Foundation Exam

May 6, 2011

Section I B

COMPUTER SCIENCE

SOLUTION

NO books, notes, or calculators may be used, and you must work entirely on your own.

Question #	Max Pts	Category	Passing	Score
1	10	ALS	7	
2	10	DSN	7	
3	10	DSN	7	
4	10	ALG	7	
5	10	ALG	7	
TOTAL	50			

You must do all 5 problems in this section of the exam.

Problems will be graded based on the completeness of the solution steps and not graded based on the answer alone. Credit cannot be given unless all work is shown and is readable. Be complete, yet concise, and above all be neat.

1) (10 pts) ALS (Order Analysis)

With justification, give the Big-Oh running times of each of the following segments of code, in terms of the variable n. Assume that n has been declared and initialized before each segment of code.

a)

```
int i;
for (i=0; i<n; i+=2) {
    for (j=i; j>0; j--)
        printf(“%d”, j);
    printf(“\n”);
}
```

The variable i takes on the values 0, 2, 4, …, n-1 (or n-2 depending on the parity of n). The inner loop runs exactly i times. Thus, the run time can be approximated by the sum 0 + 2 + 4 + … + n. (Summing to n creates a reasonable upper bound.) The second printf prints n/2 times, once for each i-loop iteration. Summing the number of times these printfs occur, we roughly get (2+n)n/4, using the formula to sum an arithmetic sequence with the first term 2 and the last term n that has n/2 terms. Adding n/2 to this, we get $n^2/2 + n$, roughly, which is $O(n^2)$.

A more simple argument that suffices to show that 2 + 4 + 6 + … + n is $O(n^2)$, is simply to make the observation that 1 + 2 + 3 + … + n is $O(n^2)$ and that these even terms exceed the odd terms, making up over 50% of the sum, which means that the sum of the even terms is proportional to a constant times n^2 as well.

$O(n^2)$

Grading: 2 points for discovering the sum 2 + 4 + … + n correlates to the run-time. 2 points for showing the sum is $O(n^2)$, 1 point for stating the result correctly.

b)

```
int a = 1, b = n, sum = 0;
while (a < b) {
    sum++;
    a = a*2;
    b = b/2;
}
```

Consider the ratio b/a. The loop stops when this ratio decreases to 1. For each loop iteration, the ratio decreases by a factor of 4. Let k be the number of loop iterations executed total. The operative equation is $1 = n/4^k$. Solving, we get $k = \log_4 k$.

O(log n)

Grading: 2 pts for the answer, 3 points for the justification

2) (10 pts) DSN (Binary Trees)

Write a recursive function that will return the number of leaf nodes in a binary tree pointed to by ptr, the input parameter. The necessary struct definition and function prototype are given below.

```
struct treeNode
    int data;
    struct treeNode *left;
    struct treeNode *right:
};



int numLeafNodes(struct treeNode* ptr)  {

    if(ptr == null)    // 2 pts
        return 0;      // 1 pt
    else if((ptr->left == null) && (ptr->right == null)) // 2 pts
        return 1;                                       // 1 pt
    else
        return numLeafNodes(ptr->left) + numLeafNodes(ptr->right);
        // 4 pts (1 pt ret, 1 pt each call, 1 pt +)
}
```

3) (10 pts) DSN (Linked Lists)

Imagine using a linked list of digits to store an integer. For example, a list containing 3, 6, 2, and 1, in that order stores the number 3621. Write an **iterative** function which accepts a linear linked list num that stores a number in this fashion and returns the value of the number. The function prototype is provided for you below. You may assume the list stores digits only and contains 9 or fewer nodes.

The node structure is as follows:

```
struct listNode {
    int data;
    struct listNode *next;
};

int getValue(struct listNode* num) {

    int sum = 0;              // 1 pt

    while (num != NULL) {          // 2 pts
        sum = 10*sum + num->data;  // 4 pts (1 for 10*sum, 1 for +
                                   //   1 for num->data,1 for syntax)
        num = num->next;           // 2 pts
    }

    return sum;                    // 1 pt
}
```

4) (10 pts) ALG (Heaps)

Consider inserting the following items in the order presented into an initially empty minimum heap: 19, 13, 17, 2, 6, 8, and 5. (The root node of a minimum heap stores the smallest item.) Draw the state of the heap after each insertion is completed.

```
1)     19          (1 pt)

2)     13          (1 pt)
      /
    19

3)     13          (1 pt)
      /   \
    19    17

4)     2           (2 pts)
      /   \
    13    17
    /
  19

5)     2           (1 pt)
      /   \
     6    17
    /  \
  19  13

6)     2           (2 pts)
      /   \
     6     8
    /  \   /
  19  13 17

7)     2           (2 pts)
      /   \
     6     5
    /  \   / \
  19  13 17  8
```

5) (10 pts) ALG (Sorting)

The following trace through shows the state of an array after each swap in an insertion sort. The state of the array at the start of each iteration of the algorithm is shown in bold. (The intermediate steps are not in bold.) Write a function that implements an insertion sort. The function prototype is given for you below. You may make use of the swap function provided as well.

```
3 2 8 6 4 (original array)
2 3 8 6 4 (end of iteration 1)
2 3 8 6 4 (end of iteration 2)
2 3 6 8 4 (end of iteration 3)
2 3 6 4 8
2 3 4 6 8 (end of iteration 4)
```

```
void insertion_sort(int array[], int length) {

    int x, j;

    for(x = 0; x < length-1; x++) {      // 2 pts (watch for AOB)

        for(j = x+1; j > 0; j--) {            // 2 pts
            if(array[j-1] > array[j] )        // 2 pts
                swap(&array[j-1], &array[j]); // 2 pts
            else
                break;                        // 2 pts
        }
    }
    // Grading note: Most students will probably write the inner
    //               loop as a while since it iterates a variable
    //               number of times. Watch out for array out of
    //               bounds errors. Take off 2 pts for each of these.

}

void swap(int* ptrA, int* ptrB) {
    Int temp = *ptrA;
    *ptrA = *ptrB;
    *ptrB = temp;
}
```