Computer Science Foundation Exam

December 16, 2011

Section I B

SOLUTION

COMPUTER SCIENCE

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)

(a) (4 pts) Determine the **Big-O** running time of the following code fragment. Do NOT use summations for this problem. Simply analyze the code and give the **Big-O** running time in terms of the variable n. **YOU MUST EXPLAIN YOUR ANSWER**.

An answer without explanation gets only 1 point.

```
int i, j, k, someValue = 0;
for(i = 1; i < 14*n*n; i++) {
     for(j = 1; j < 42*n*n*n*n*n; j++) {
          for(k = 1; k < n*n*n; k++) {
               someValue = someValue + 7;
          }
     }
}
```

Outer loop iterates n^2 times (from i = 1 to $14n^2$).
Inner loop iterates n^5 times (from j = 1 to $42n^5$).
Most inner loop iterates n^3 times (from k = 1 to n^3).
For each iteration of the inner loop, there is 1 operation (constant work).
So we have $14n^2*42n^5*n^3$, or $O(n^{10})$.

(b) (6 pts) Write, but DO **NOT** solve, a summation that describes the number of multiplications performed by the following code fragment, in terms of n.

```
weirdSum = 0;
for(i = 19; i < n + 292; i++) {
     weirdSum = weirdSum * 5 - i*2;
     for(j = i - 3; j < i + i + 19; j++) {
          weirdSum = weirdSum * n - j * 3 + i * j;
     }
}
```

Solution:

$$\sum_{i=19}^{n+291}\left(2+\sum_{j=i-3}^{2i+18}3\right)$$

2) (10 pts) DSN (Recursive Algorithms)

The 3n+1 problem is as follows:

Given an input value n, do the following:

1) If n is 1, stop.
2) If n is odd, produce the number 3n+1
3) If n is even, produce the number n/2
4) After step 2 or 3, go back to step 1.

The goal is to figure out how many steps it will take for the process to stop with a given number. For example, if we start with 3, the sequence calculated is 3, 10, 5, 16, 8, 4, 2, and 1. In this case, it took 7 steps to stop. **Write a recursive function that given an input value n, returns the number of steps it takes to reach 1. If n is 1, your function should return 0.** Assume that the return value will be less than 1000 for all the input values n for which your function will be tested. Also assume that no overflow errors will occur in the calculation.

```
int threenplusone(int n) {

    if (n == 1)                                 // 1 pt
        return 0;                               // 1 pt

    if (n%2 == 1)                               // 2 pts
        return 1 + threenplusone(3*n+1);        // 3 pts

    return 1 + threenplusone(n/2);              // 3 pts
}
```

3) (10 pts) DSN (Linked Lists)

Write a function that operates on an existing linked list of integers. Your function should insert a node with the input value newvalue into the linked list pointed to by head **in numerical order from lowest to highest** and return a pointer to the front of the list. **Assume that the linked list pointed to by head is already in numerical order from lowest to highest.** (Note: You must handle the case where head is NULL.)

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

struct node* insertInOrder(struct node* head, int newvalue)
{
    struct node *iter;

    struct node* temp = (struct node*)malloc(sizeof(struct node));
    temp->data = num;
    temp->next = NULL;

    if (front == NULL)
        return temp;

    if (temp->data < front->data) {
        temp->next = front;
        return temp;
    }

    iter = front;
    while (iter->next != NULL && temp->data > iter->next->data)
        iter = iter->next;

    temp->next = iter->next;
    iter->next = temp;
    return front;
}

// Grading: Creating a new node to store newvalue - 2 pts
//          Working in front == NULL case - 2 pts
            Working in new node goes first case - 2 pts
            Iterating to the right place otherwise - 2 pts
            Relinking items in this case and returning - 2 pts
```

4) (10 pts) ALG (Heaps)

Show the result of inserting the following integers into a minHeap: 18, 3, 14, 7, 1, 8, 9, 16, 4, 2 (Note: a minHeap is one where the root node stores the smallest item.) Draw a box around your final answer, in the form of a complete binary tree representing the final heap.

```
18      18       3        3        3        1        1
       /        / \      / \      / \      / \      / \
      3       18   14  18   14   7   14   3   14   3    8
                       /         / \     / \  /    / \  / \
                      7        18   1  18   7 8  18  7 14  9


     1              1              1              1
    / \            / \            / \            /   \
   3   8          3   8          3   8          2     8
  / \ / \        / \ / \        / \ / \        / \   / \
 18 7 14 9      16 7 14 9      4  7 14 9      4   3 14  9
 /              / \            / \ /          / \  /
16             18  4          18 16 2        18 16 7
```

Grading: 1 pt for each insertion, OR 1 pt for each position in the final tree – whichever score is higher.

5) (10 pts) ALG (Sorting)

(a) (6 pts) Show the contents of the array below, **after each Merge occurs**, in the process of running Merge-Sort on the array shown below. Assume that the Merge Sort is implemented recursively, where the first half of the array is completely sorted before the second half of the array is ever accessed.

Index	0	1	2	3	4	5	6	7
Initial Array:	**5**	**2**	**4**	**7**	**8**	**3**	**6**	**1**
after 1st Merge:	**2**	**5**	**4**	**7**	**8**	**3**	**6**	**1**
after 2nd Merge:	**2**	**5**	**4**	**7**	**8**	**3**	**6**	**1**
after 3rd Merge:	**2**	**4**	**5**	**7**	**8**	**3**	**6**	**1**
after 4th Merge:	**2**	**4**	**5**	**7**	**3**	**8**	**6**	**1**
after 5th Merge:	**2**	**4**	**5**	**7**	**3**	**8**	**1**	**6**
after 6th Merge:	**2**	**4**	**5**	**7**	**1**	**3**	**6**	**8**
Sorted Array:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

(b) (4 pts) Show the result of running **Partition** (from Quick sort) on the array below using the leftmost element as the pivot element. Show what the array looks like after each swap. And then show the array after the partition (**remembering the final swap**). Assume that the in-place implementation of partition is used.

Index	0	1	2	3	4	5	6	7
Initial Array:	**7**	**4**	**33**	**9**	**2**	**6**	**8**	**12**
after 1st Swap:	**7**	**4**	**6**	**9**	**2**	**33**	**8**	**12**
after 2nd Swap:	**7**	**4**	**6**	**2**	**9**	**33**	**8**	**12**
After Partition:	**2**	**4**	**6**	**7**	**9**	**33**	**8**	**12**