Computer Science Foundation Exam

December 12, 2014

Section I B

COMPUTER SCIENCE

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

SOLUTION

Question #	Max Pts	Category	Passing	Score
1	10	ANL	7	
2	10	ANL	7	
3	10	DSN	7	
4	10	DSN	7	
5	10	ALG	7	
TOTAL	50		35	

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) (10pts) ANL (Algorithm Analysis)

Consider the following function shown below:

```
int countBitsOn(int n) {
    if (n == 0)
        return 0;
    return n%2 + countBitsOn(n/2);
}
```

(a) (3 pts) Let T(n) be the runtime of `countBitsOn` with an input of size n. Write a recurrence relation that T(n) satisfies, assuming that a constant number of simple operations takes 1 unit of time.

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Grading: 1 pt for T(n/2), 1 pt for plus, 1 pt for 1

(b) (7 pts) Use the iteration technique to solve for T(n). To simplify the calculation, you may assume that n = 2^m for some non-negative integer m. Make sure to substitute back and express your final answer in terms of n. Note: You may assume T(1) = 1.

$$T(2^m) = T\left(\frac{2^m}{2}\right) + 1$$
$$T(2^m) = T(2^{m-1}) + 1$$
$$T(2^m) = \left(T\left(\frac{2^{m-1}}{2}\right) + 1\right) + 1$$
$$T(2^m) = T(2^{m-2}) + 2$$
$$T(2^m) = (T(\frac{2^{m-2}}{2}) + 1) + 2$$
$$T(2^m) = T(2^{m-3}) + 3$$

We see that after k steps, we have $T(2^m) = T(2^{m-k}) + k$. Substitute k = m to get:

$$T(2^m) = T(2^{m-m}) + m = T(1) + m = m + 1$$

Finally, since n = 2^m, it follows that m = $\log_2$n. Thus, $T(n) = log_2 n + 1 = O(\lg n)$.

Grading: 1 pt for each of iteration, for 3 iterations, 2 pts for the general guess, 1 pt for plugging in k = m, and 1 pt for the final answer (either exact or big-oh answer is accepted.)

2) (10 pts) ANL (Algorithm Analysis)

(a) (5 pts) An algorithm for sorting student records runs in θ(*nlgn*) time. It takes 4 ms to sort 2^{16} student records. How much time will it take, approximately, ***in ms***, to sort 2^{20} student records?

Set up an equation for T(n), the run-time of this algorithm as follows:

$$T(n) = cnlog_2 n$$

for some constant c. Substitute in the first piece of information and solve for c:

$$T(2^{16}) = c2^{16}log_2 2^{16} = 4ms$$
$$c2^{16}(16) = 4ms$$
$$c = \frac{4}{2^{20}}ms$$

Note: we didn't simplify thsi fraction in the hopes that this form will be useful later. Now, solve for T(2^{20}):

$$T(2^{20}) = c2^{20}log_2 2^{20} = \frac{4ms}{2^{20}} \times 2^{20} \times 20 = 80\ ms$$

Grading: 1 pt for general form, 2 pts for solving for c, 2 pts for plugging in to get answer.

(b) (5 pts) An algorithm for finding the shortest distance between any pair of *n* locations runs in θ(n^3) time. For *n* = 125, the algorithm takes 100 ms. How long will it take, ***in seconds***, approximately, to run on an input with *n = 500*?

Set up an equation for T(n), the run-time of this algorithm as follows:

$$T(n) = cn^3$$

for some constant c. Substitute in the first piece of information and solve for c:

$$T(125) = c125^3 = 100ms$$
$$c = \frac{100}{125^3}ms$$

Now, solve for T(500):

$$T(500) = c500^3 = \frac{100ms}{125^3} \times 500^3 = (\frac{500}{125})^3 \times 100\ ms = 4^3 \times 100ms = 6400ms = 6.4sec$$

Grading: 1 pt for general form, 2 pts for solving for c, 1 pt for answer in ms, 1 pt to convert to seconds.

3) (10 pts) DSN (Linked Lists)

Write a function, `insertFront`, that inserts a value into the front of a circular linked list and returns the front of the resulting linked list. Your code should have a single malloc in it for the new node it creates and should make sure that the appropriate links between nodes are adjusted accordingly. Finally, make sure your code works for inserting into an empty circular linked list. In the prototype shown below, the first parameter is the pointer to the front of the current list and the second parameter is the value to be inserted into the front of the list.

Use the struct definition provided below.

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

node* insertFront(node* front, int value) {

    node* tmp = malloc(sizeof(node));     // 2 pts total
    tmp->data = value;
    tmp->next = front;

    if (front == NULL) {                  // 2 pts for this case
        tmp->next = tmp;
        return tmp;
    }

    node* iter = front;                  // 1 pt
    while (iter->next != front)          // 2 pts
        iter = iter->next;               // 1 pt

    iter->next = tmp;                    // 1 pt

    return tmp;                          // 1 pt
}
```

Note: there are many solutions, be very careful to trace through alternate solution ideas and award points proportionally to the subsections of the solution as indicated here.

4) (10 pts) DSN (Binary Trees)

Consider the problem of finding the kth smallest item in a binary search tree storing ***unique*** values. Write a ***recursive*** function with the prototype shown below to solve this problem. The first parameter to the function will be a pointer to the root node of the binary tree and the second value will be the 1-based rank of the item to be returned. (If k is 1, you should return the smallest item in the tree.) **Note that the number of nodes in the subtree at each node is stored inside the node.** You may assume that the value of k will be in between 1 and the value of numNodes in the struct pointed to by root.

```
typedef struct treenode {
    int data;
    int numNodes;
    struct treenode *left;
    struct treenode *right;
} treenode;

int kthSmallestValue(treenode* root, int k) {

    // 2 pts for this case.
    if (root->left == NULL && k == 1)
        return root->data;

    // 2 pts for this case.
    int leftCnt = root->left->numNodes;
    if (k - 1 == leftCnt)
        return root->data;

    // 3 pts for this case.
    if (k - 1 < leftCnt)
        return kthSmallestValue(root->left, k);

    // 3 pts for this case.
    return kthSmallestValue(root->right, k-leftCnt-1);
}
```

5) (10 pts) ALG (Sorting)

(a) (4 pts) Consider running a merge sort on the array shown below. What would the contents of the array be right ***before*** the very last merge finishes? (Note: There are a total of 7 calls to the merge function when the array below gets merge sorted.)

Index	0	1	2	3	4	5	6	7
Value	17	13	27	9	18	15	2	8

State of Array Right Before the Last Merge:

Index	0	1	2	3	4	5	6	7
Value	**9**	**13**	**17**	**27**	**2**	**8**	**15**	**18**

Grading: 2 pts for left half, 2 pts for right half, no partial credit for either side, so only possible scores are 0, 2 or 4.

(b) (6 pts) Which of the following recurrence relations best describes the worst case run time of Quick Sort of n elements? (Circle the correct answer.) Explain why the solution to this recurrence corresponds to the worst case run time of Quick Sort of n elements.

(a) $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ (b) $T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right) + O(n)$

(c) $T(n) = 2T(n-1) + O(n)$ **(d) $\boldsymbol{T(n) = T(n-1) + O(n)}$**

In Quick Sort's worst case, the partition of n elements, splits the numbers into one set of size n - 1 and another set of size 0. This partition itself takes O(n) time and then the remaining set of size n - 1 remains to be sorted, which takes T(n - 1) time. The total time taken is the sum of the partition and the recursive call on the array of size n - 1.

Grading: 2 pts answer, 2 pts to explain recursive call in worst case, 2 pts to explain where the O(n) comes from.