Computer Science Foundation Exam

August 9, 2013

Section I B SOLUTION

COMPUTER SCIENCE

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

Name: ____

PID: ____

Question #	Max Pts	Category	Passing	Score
1	10	ANL	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) (10pts) ANL (Algorithm Analysis)

(a) (4 pts) After some experimental analysis, it is determined that the run-time of the function below is $O(n^2)$, where n represents the value of the input parameter. Assume that any function calls shown below are defined elsewhere. Give the most likely explanation for the observed run-time.

```
struct list* makeList(int n) {

    struct list* ans = NULL;
    int i;
    for (i=1; i<=n; i++)
        ans = f(ans, i);
    return ans;
}
```

The most likely explanation is that the function call f takes linear time in relation to the size of the list ans, (or potentially the other input value, i.) Thus, as the list grows, which may be what occurs in f, the time the function takes increases. Using this possibility we can roughly model the time of the function to be $1 + 2 + 3 + \ldots + n = O(n^2)$.

Grading: 3 pts for mentioning that the function call might not be O(1), 1 pt for explaining a likely run-time for it.

(b) (6 pts) Consider the function shown below. What is its run-time in terms of the input variable n? Assuming that array is sorted in ascending order and both array and vals are of length n, what does the function f compute?

```
int f(int* array, int* vals, int n) {
    int count = 0;
    for (i = 0; i < n; i++) {
        int low = 0, high = n-1;
        while (low < high) {
            int mid = (low+high)/2;
            if (array[mid] < val[i])
                low = mid+1;
            else if (array[mid] > val[i])
                high = mid-1;
            else {
                count++;
                break;
            }
        }
    }
    return count;
}
```

The run time is O(nlgn). (3 pts) The outer loop always runs n times and the inner loop runs a binary search over n elements, which takes O(lg n) time. (1 pt) f computes the number of items in the array vals that are stored in the array array. (2 pts)

2) (10 pts) DSN (Recursive Algorithms – Binary Trees)

Write a **recursive** function `sumLeafNodes` that returns the sum of the values stored in the leaf nodes of the tree pointed to by root, the input parameter to the function. Use the following struct definition:

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


int sumLeafNodes(struct treeNode* root) {

    if (root == NULL)                 // 1 pt
        return 0;                     // 1 pt

    else if (root->left == NULL && root->right == NULL) // 2 pts
        return root->data;                              // 1 pt

    return sumLeafNodes(root->left) + sumLeafNodes(root->right);

    // return and sum = 1 pt, each rec call = 2pts
}
```

3) (10 pts) DSN (Linked Lists)

Write a function, `inOrder`, that determines whether or not the linked list pointed to by front, the input parameter, contains values in sorted order from smallest to largest, with repeated values allowed. Your function should return 1 if the values are in order, and 0 if they are not. Use the struct definition provided below.

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

int inOrder(struct node *front) {

    if (front == NULL)                // 1 pt
        return 1;                     // 1 pt

    struct node* tmp;

    // 2 pts for iterating through the list exactly n-1 times for
    //       a list of n elements.
    // 3 pts for correctly returning 0 for an out of order pair
    // 2 pts for no NULL ptr errors in this part.
    // 1 pt for returning after checking the whole list

    for (tmp=front; tmp->next != NULL; tmp = tmp->next)
        if (tmp->data > tmp->next->data)
            return 0;

    return 1;
}
```

4) (10 pts) ALG (Problem Solving)

Write a function that determines whether or not a pair of values in a sorted integer array sum up to a given target. Your function should return 1 if such a pair exists and 0 otherwise. (The number of points awarded will be based on correctness and the run-time of your algorithm. In particular, the maximum score for an O(n^2) algorithm will be 4 pts, an O(nlgn) algorithm will be 8 pts and an O(n) algorithm will be 10 pts, where n repesents the input size of the sorted array.

```
// Pre-condition: array has n elements in sorted order from smallest
//                to largest.
// Post-condition: 1 is returned if 2 elements of array add exactly
//                 to target.
int hasMatch(int* array, int n, int target) {

    int low = 0, high = n-1;                            // 1 pt

    while (low < high) {                                // 2 pts

        if (array[low] + array[high] == target)         // 2 pts
            return 1;                                   // 1 pt

        else if (array[low] + array[high] < target)     // 1 pt
             low++;                                     // 1 pt

        else
             high--;                                    // 1 pt
    }

    return 0;                                           // 1 pt

}

// For less efficient solutions, just take off partial credit for
// major implementation errors.
```

5) (10 pts) ALG (Sorting)

(a) Consider the following function which is supposed to sort values in the array from smallest to largest:

```
// Pre-condition: array contains size elements.
// Post-condition: array will be in sorted order.
void sort(int* array, int size) {
    int i,j;
    for (i=1; i<size-1; i++) {              // line 4
        j = i;                              // line 5
        while (array[j] < array[j-1]) {     // line 6
            int temp = array[j];            // line 7
            array[j] = array[j-1];          // line 8
            array[j-1] = temp;              // line 9
            j--;                            // line 10
        }
    }
}
```

(a) (i) (4 pts) Make the two changes that need to be made in order to fix this function so it sorts properly. Indicate which two lines need to be fixed and what they need to be changed to.

```
for (i=1; i<size; i++) { // line 4
```

Since i represents the index of the item to be inserted in the already sorted list, we must set i = size-1 to insert the last item. In the old version, this item would just stay in its original location.

```
while (j > 0 && array[j] < array[j-1]) {          // line 6
```

The array index check is necessary to avoid an array out of bounds with the array[j-1] reference.
<u>Grading: 1 pt for the line number and 1 pt for the fix.</u>

(a) (ii) (2 pts) Once these fixes are made, what sort does this function implement?

Insertion Sort (2 pts all or nothing)

(d) (4 pts) Consider running a Merge Sort on the array shown below. Show the contents of the array right before the LAST merge is executed.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Values	37	98	5	41	25	44	99	79	92	6	2	11	87	45	63	18

(Write answer here)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Values	**5**	**25**	**37**	**41**	**44**	**79**	**98**	**99**	**2**	**6**	**11**	**18**	**45**	**63**	**87**	**92**

Grading: 4 pts all correct, 3 pts 3 or fewer items incorrect, 2 pts, 8 or more correct, 1 pt 4 or more correct, 0 pts otherwise