Computer Science Foundation Exam

May 2, 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	DSN	7	
3	10	DSN	7	
4	10	ALG	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 <u>not</u> 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 <u>be neat</u>.

1) (10pts) ANL (Algorithm Analysis)

In a game of golf, the winner is the person with the lowest number of strokes. Given *n* golf scores determine the following.

(a) (1 pt) What is the run time of generating an unsorted array of these *n* values? Provide a theta run-time in terms of *n*.

O(n). Reading in the values takes O(n) time, constant time for each value. (1 pt)

(b) (3 pts) What is the run-time of generating an sorted array of these n values, assuming a Merge Sort is used? Assume an efficient implementation. Provide a theta run-time in terms of *n*. Justify your answer.

O(nlogn). Reading in the values takes O(n) time and sorting those values takes O(nlogn), the worst case time of Merge Sort. Adding, we get O(nlogn). (1 pt ans, 2 pts explanation)

(c) (3 pts) Suppose we wished to show ONLY the winner's score. In this case, which of the following is more efficient? Justify your answer.

- Generating the array as in part a and performing a linear search
- Generating the array as in part b and performing a binary search

Linear Search. Since we are only interested in the lowest value this process takes O(n) to read in the values and O(n) to perform a linear search on the unsorted list. Added this requires O(n) time to complete.

Sorting the array using Merge Sort takes O(nlogn). Performing a binary search is not strictly necessary, we can find the lowest value in O(1) time. Added this requires O(nlogn) to complete.

(1 pt answer, 2 pts explanation)

(d) (3 pts) Suppose we wished to show all the scores in ascending order. In this case, which of the following is more efficient? Justify your answer.

- Generating the array as in part a and performing n linear searches
- Generating the array as in part b and outputting it

Binary Search. In this case we are interested in all n values in sorted order. Sorting the array using Merge Sort takes O(nlogn) time and printing each value takes O(n). Added, this required O(nlogn) time.

The linear search solution requires us to search for each new lowest value to print out, which is O(n) for each of the n values for a total time of O(n^2).

(1 pt answer, 2 pts explanation)

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

Write a **recursive** function `validTotal` that returns the sum of values in a binary tree whose data values fall within input parameters `min` and `max`, **inclusive.** The tree, the minimum valid number, and the maximum valid number are inputs to the function.

Use the following struct definition:

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


int validTotal(treenode* root, int min, int max) {

     // 2 pts for this case.
     if (root == NULL)
          return 0;

     // 4 pts for this case - 2 pts for root, 1 pt for each rec call
     if (root->data >= min && root->data <= max)
          return root->data + validTotal(root->left, min, max) +
                    validTotal(root->right, min, max);

     // 4 pts for this case - 2 pts for each rec call here
     else
          return validTotal(root->left, min, max) +
                    validTotal(root->right, min, max);




}
```

3) (10 pts) DSN (Linked Lists)

Write a **recursive** function, `insertName`, that adds a new node in **lexicographical** order, as defined by strcmp in string.h, to the list pointed to by the input parameter `front`. Your function should return the front of the resulting list. You may assume that string.h has been included.

Use the struct definition provided below.

```
typedef struct node {
    char* name;
    struct node* next;
} node;

node* insertName(node* front, char* newname) {

    //initialization - 3 pts 1 per line.
    node* tmp = malloc(sizeof(node));
    tmp->name = (char *)malloc(sizeof(char)*(1+strlen(newname)));
    strcpy(tmp->name, newname);


    //check front of list - 2 pts if, 2 pts link and return
    if (front == NULL || strcmp(tmp->name, front->name) <= 0) {
        tmp->next = front;
        return tmp;
    }

    // Recursive insert - critical to set return value to next
    // pointer of front node.
    front->next = insertName(front->next, newname);    // 2 pts
    return front;                                      // 1 pt
}
```

4) (10 pts) ALG (Stacks and Queues)

Complete a linked list implementation of stack functions empty() and push(), keeping your code consistent with what is provided below to maintain a stack of positive integers. (Note: This isn't necessarily the most desirable way to set up this code. You're simply being tested on the mechanics of how this works and writing code consistent to a convention you haven't chosen.)

```
#include <stdio.h>
#include <stdlib.h>

#define EMPTY 0

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

typedef struct stack {
    node* front;
} stack;

void init(stack* myStack) {
    myStack->front = NULL;
}

int empty(stack* myStack) {

    return myStack->front == NULL;        // 2 pts

}

void push(stack* myStack, int value) {

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

}

int pop(stack* myStack) {

    if (empty(myStack)) return EMPTY;

    int retval = myStack->front->data;
    node* freeNode = myStack->front;
    myStack->front = myStack->front->next;
    free(freeNode);
    return retval;
}

int top(stack* myStack) {
    if (empty(myStack)) return EMPTY;
    return myStack->front->data;
}
```

5) (10 pts) ALG (Sorting)

(a) (5 pts) Consider sorting the array below using Selection Sort, where after the first iteration, the **minimum value** in the array is in its correct location. Show the contents of the array after each iteration of the outer loop.

Original	3	2	5	1	6	4
1st iteration	**1**	**2**	**5**	**3**	**6**	**4**
2nd iteration	**1**	**2**	**5**	**3**	**6**	**4**
3rd iteration	**1**	**2**	**3**	**5**	**6**	**4**
4th iteration	**1**	**2**	**3**	**4**	**6**	**5**
5th iteration	**1**	**2**	**3**	**4**	**5**	**6**

Grading: 1 pt per row, award only if the row is perfectly correct.

(b) (5 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	11	48	83	7	1	77	67	61	90	75	54	23	64	42	65	93

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Values	**1**	**7**	**11**	**48**	**61**	**67**	**77**	**83**	**23**	**42**	**54**	**64**	**65**	**75**	**90**	**93**

Grading: Correct = 5 pts
If pairs are sorted only – 1 pt,
If list is all sorted – 0 pts,
If there are fewer than 5 errors, 1 pt off per error
If there are 5 or more errors and they don't appear to be systematic – 0