Computer Science Foundation Exam

August 28, 2021

Section I A

DATA STRUCTURES

SOLUTION

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

Question #	Max Pts	Category	Score
1	10	DSN	
2	5	ALG	
3	10	DSN	
TOTAL	25	----	

You must do all 3 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. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we have an array to store all of the holiday presents we have purchased for this year. Now that the holidays are over and all the presents have been given out, we need to delete our list. Our array is a dynamically allocated array of structures that contains the name of each present and the price. The name of the present is a dynamically allocated string to support different lengths of strings. Write a function called delete_present_list that will take in the present array and free all the memory space that the array previously took up. Your function should take 2 parameters: the array called present_list and an integer, num, representing the number of presents in the list and return a null pointer representing the now deleted list. (Note: The array passed to the function may be pointing to NULL, so that case should be handled appropriately.)

```
struct present {
     char *present_name;
     float price;
};

struct present* delete_present_list(struct present* present_list, int
num) {

     int i;

     // Check for null pointers
     // 1 point
     if(present_list == NULL)
        return NULL;

     // Traversing the list to free character arrays
     // 4 points
     for(i = 0; i < num; i++)
        free(present_list[i].present_name);

     // Free the array pointer
     // 3 points
     free(present_list);

     // Return null
     // 2 points
     return NULL;

}
```

2) (5 pts) DSN (Linked Lists)

Suppose we have a singly linked list implemented with the structure below and a function that takes in the head of the list.

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

int whatDoesItDo (node * head) {
     struct node * current = head;
     struct node * other, *temp;

     if (current == NULL)
        return head;

     other = current->next;

    if (other == NULL)
        return head;

    other = other->next;
    temp = current->next;
    current->next = other->next;
    current = other->next;

    if (current == NULL) {
        head->next = temp;
        return head;
    }

    other->next = current->next;
    current->next = temp;

    return head;
}
```

If we call whatDoesItDo(head) on the following list, show the list after the function has finished.

head -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

Solution

head -> 1 -> 4 -> 2 -> 3 -> 5 -> 6 -> 7

Grading: 2 pts for having the 1, 5, 6 and 7 unchanged, 1 pt each for the position of 4, 2 and 3.

3) (10 pts) ALG (Queues)

Suppose we wish to implement a queue using an array. The structure of the queue is shown below.

```
struct queue {
    int *array;
    int num_elements;
    int front;
    int capacity;
};
```

The queue contains the array and three attributes: the current number of elements in the array, the current front of the queue, and the maximum capacity. Elements may be added to the queue not just at the end of the array but also in the indices at the beginning of the array before front. Such a queue is called a circular queue.

Write a function to implement the dequeue functionality for the queue, while ensuring that no null pointer errors occur. Your function should take in 1 parameter: a pointer to the queue. Your function should return the integer that was dequeued. If the queue is NULL or if there are no elements to dequeue, your function should return 0.

```
int dequeue(struct queue * q) {

     // Check for NULL - 1 point
     if (q == NULL)
          return 0;

     // Check for no elements - 2 points
     if(q->num_elements == 0)
          return 0;

     // Obtain value to return - 1 point
     int retval = q->array[q->front];

     // Update front, with circular logic - 3 points
     q->front = (q->front + 1) % q->capacity;

     // Update number of elements - 2 points
     q->num_elements = q->num_elements - 1;

     // Returns value - 1 point
     return retval;
}
```

Computer Science Foundation Exam

August 28, 2021

Section I B

DATA STRUCTURES

SOLUTION

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

Question #	Max Pts	Category	Score
1	10	ALG	
2	5	ALG	
3	10	ALG	
TOTAL	25		

You must do all 3 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. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (10 pts) ALG (Binary Trees)

Consider a function that takes in a pointer to a binary tree node and returns a pointer to a binary tree node defined below:

```
typedef struct bintreenode {
    int data;
    struct bintreenode* left;
    struct bintreenode* right;
} btreenode;

btreenode* somefunction(btreenode* root) {
    if (root == NULL) return NULL;
    somefunction(root->left);
    somefunction(root->right);
    btreenode* tmp = root->left;
    root->left = root->right;
    root->right = tmp;
    return root;
}
```

Let the pointer tree point to the root node of the depicted below:

```
              20
          /        \
        42          13
      /    \          \
     6      18         27
    /      /  \       /  \
  15     72    9     22  35
                         /
                        87
```

If the line of code `tree = somefunction(tree)` were executed, draw a picture of the resulting binary tree that the pointer tree would point to.

```
              20
           /      \
         13        42
       /          /  \
     27         18    6
    /  \       /  \    \
  35   22     9   72   15
   \
    87
```

Grading: 1 pt for 20 at root, 2 pts for swapping 13, 42, 3 pts for the left subtree of 13, 4 pts for the placement of 18, 6, 9, 72 and 15.

2) (5 pts) ALG (Heaps)

Suppose we construct a minheap where each node contains a string, and we order our strings according to the following rules:

1. Given strings *a* and *b*, we say *a* < *b* if *a* has fewer characters than *b*.
2. If two strings *a* and *b* have the same length, we say *a* < *b* if *a* comes before *b* in alphabetical order.

Suppose we construct a minheap of strings using these rules. Furthermore, suppose all the strings in our minheap contain lowercase letters only (so, no punctuation, spaces, uppercase letters, and so on), and we do not allow any duplicate strings into the minheap.

Given two arbitrary strings in our minheap, *x* and *y*, can we safely say that if *x* is a prefix of *y*, then *y* must be in one of *x*'s subtrees? Note that *x* may not be the root of the minheap. If so, explain why this must be the case. If not, draw a minheap of strings that very clearly shows this is not necessarily the case (and clearly label which string is *x* and which string is *y* in your counterexample).

Here's one such minheap, where *x* = "ef" and *y* = "efficacy":

```
                ab
              /    \
           cd       ef
         /
    efficacy
```

Notice that "ef" is a prefix of "efficacy", but "efficacy" is not in either subtree of "ef".

Grading:
+1 attempt to create a counter-example (0/5 if they say y must be a subtree of x)
+2 for creating a valid minheap using these rules
+2 for coming up with *x* and *y* that satisfy the problem (they should denote which nodes store x and y and x must be a prefix of y.)

3) (10 pts) ALG (AVL Trees)

Draw an AVL tree **of integers** and designate a single node in the AVL tree such that, if that node were to be deleted, two rebalance operations (not one double rotation, but two separate operations and two different nodes) would occur. Clearly label the node to delete which would precipitate those operations and show the result of deleting that node. (Thus, you should have two drawings, a before drawing of the original tree with the node to be deleted clearly designated, and an after drawing showing what the tree looks like after the node is deleted and goes through 2 rebalance operations.)

There are many, many solutions, perhaps the easiest to visualize is where one side of the tree is short and the other is tall. To force 2 rebalance operations, the tree must be a height of at least 4, with one subtree height 2 and the other height 3. The height 2 subtree has to rebalance to a height 1 subtree to force the second restructure. In this solution, the tree is "right" heavy:

```
                100
          /             \
        50               200
      /     \          /     \
    25       80      150     300
            /        /      /   \
          60       120    250   350
                                   \
                                   400
```

Deleting 25 from this AVL tree will cause a rebalance at 50 AND a rebalance at 100. Here is the resulting tree after deleting 25:

```
                200
          /             \
        100              300
      /     \          /     \
    60      150      250     350
   /  \     /                   \
  50  80  120                   400
```

Grading: 3 pts drawing a valid AVL tree BEFORE deletion,
1 pt to clearly indicate a node to delete,
3 pts if delete causes 2 rebalances, 1 pt if it causes 1 rebalance, 0 pts if no rebalances
3 pts for tree after deletion, grader decides partial.

If the tree is not a valid AVL tree, automatically 0/10.
If everything is good except there's only 1 rebalance 8/10.
If everything is good except there is no rebalance 5/10 since the deletion is easier.

Computer Science Foundation Exam

August 28, 2021

Section II A

ALGORITHMS AND ANALYSIS TOOLS

SOLUTION

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

Question #	Max Pts	Category	Score
1	10	ANL	
2	10	ANL	
3	5	ANL	
TOTAL	25		

You must do all 3 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. For each coding question, assume that all of the necessary includes (stdlib.h, stdio.h, math.h, string.h) for that particular question have been made.

1) (10 pts) ANL (Algorithm Analysis)

Consider the following problem: You are given a set of weights, $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$ and a target weight T. The target weight is placed on one side of a balance scale. The problem is to determine if there exists a way to use some subset of the weights to add on either side of the balance so that the scale will perfectly balance or not. For example, if T = 12 and the set of weights was {6, 2, 19, 1}, then one possible solution would be to place the weights 6 and 1 on the same side of the balance as 12 and place the weight 19 on the other side. Below is a function that solves this problem recursively, with a wrapper function to make the initial recursive call. In terms of n, the size of the input array of weights, **with proof,** determine the **worst case** run time of the wrapper function. **(Note: Since only the run time must be analyzed, it's not necessary to fully understand WHY the solution works. Rather, the analysis can be done just by looking at the structure of the code.)**

```
int makeBalance(int weights[], int n, int target) {
    return makeBalanceRec(weights, n, 0, target);
}
int makeBalanceRec(int weights[], int n, int k, int target) {
    if (k == n) return target == 0;
    int left = makeBalanceRec(weights, n, k+1, target-weights[k]);
    if (left) return 1;
    int right = makeBalanceRec(weights, n, k+1, target+weights[k]);
    if (right) return 1;
    return makeBalanceRec(weights, n, k+1, target);
}
```

In the worst case, a recursive call with input k makes 3 recursive calls with input k+1. Upon examining the code, what we really see is that as k increases, the subarray of weights we are considering decreases in length. Thus, a better way of interpreting the code is that given an array of size n, at worst, 3 recursive calls are made on an array of size n-1. The extra work beyond the recursive calls is all constant work (some additions and if statements). Thus, if we let T(n) represent the run time of the wrapper function in terms of the size of the input array, the worst case run time of the function is represented by the recurrence relation: $T(n) = 3T(n-1) + 1$, (note for a more accurate analysis one may say +c, a constant.)

Iterate twice:

$T(n) = 3(3T(n-2) + 1) + 1 = 9T(n-2) + 4$
$T(n) = 9(3T(n-3) + 1) + 4 = 27T(n-3) + 13$

In general after k steps we get: $T(n) = 3^k T(n - k) + \sum_{i=0}^{k-1} 3^i$. Since $T(0) = 1$ (base case), plug in n = k into the recurrence to get:

$T(n) = 3^n T(0) + \sum_{i=0}^{n-1} 3^i = 3^n + \sum_{i=0}^{n-1} 3^i = \sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{3-1} =$ **$O(3^n)$.** (Note: Since $O(3^{n-1})$ and $O(3^{n+1})$ are equivalent to $O(3^n)$, these get full credit also.) Also, okay just to look at the bottom level…

Grading: 4 pts for recurrence relation, 4 pts for solving it, 2 pts for the final answer. May use some leeway to award partial credit, but please give 0/10 for $O(n^2)$ or other polynomial answers.

2) (10 pts) ANL (Algorithm Analysis)

A program processing an array of size 100 took 50 ms to finish and on an array of size 1000 it took 75 ms to finish. What Big Oh runtime would be most reasonable for the program? (Hint: make a couple guesses to the function and see if those guesses are consistent with the run-times listed.)

A quick analysis shows that the function must be sub-linear since multiplying the input size by 10 resulted in a multiplicative factor of only 1.5 (less than 10) to the run-time.

So, two candidates we might try for the run-time are f(n) = $\sqrt{n}$ or f(n) = lg n.

Let's try the first one: Let T(n) be the run time of the function for an array of input size n, then we have

T(n) = $c\sqrt{n}$, for some constant c. Thus:

T(100) = $c\sqrt{100} = 50\ ms \rightarrow$ c = 5 ms

T(1000) = = $c\sqrt{1000} = 75\ ms \rightarrow c = \frac{75}{10\sqrt{10}} = \frac{7.5}{\sqrt{10}} ms$. Note that the square root of 10 is greater than 3, so this second equation indicates that c is in between 2 and 3 ms and isn't very consistent with the previous result.

Let's try the second guess: : Let T(n) be the run time of the function for an array of input size n, then we have

T(n) = $c lg\ n$, for some constant c. Thus:

T(100) = $clg(100) = 50\ ms \rightarrow$ c = $\frac{50\ ms}{\lg(100)} = \frac{50\ ms}{2\lg(10)} = \frac{25ms}{\lg 10}$

T(1000) = $clg(1000) = 75\ ms \rightarrow$ c = $\frac{75\ ms}{\lg(1000)} = \frac{75\ ms}{3\lg(10)} = \frac{25ms}{\lg 10}$

Notice that both pieces of information result in the same value of c. Thus, the given data points are consistent with the hypothesis that the run-time of the program is **O(lg n).**

Grading: **5 pts for the correct answer without any proof**
5 pts for the logic behind the answer. There are many ways to quantitatively make the argument.
3 pts for trying to verify any answer at all.
3 pts for any sublinear guess without any verification or proof.

3) (5 pts) ANL (Summations)

What is the closed form of the following summation? Please show each step of work. **(Note: the bounds on the inner summation are NOT a misprint!!!)**

$$\sum_{a=0}^{n}(\sum_{b=a}^{a} 4b)$$

$$\sum_{a=0}^{n}(\sum_{b=a}^{a} 4b) = \sum_{a=0}^{n} 4a = \frac{4n(n+1)}{2} = 2n(n+1)$$

Grading: 2 pts for getting that the sum of 4b is just 4a. 2 pts for the sum of a, 1 pt for multiplying by 4 and simplifying. (Polynomial form also accepted.)

Computer Science Foundation Exam

August 28, 2021

Section II B

SOLUTION

ALGORITHMS AND ANALYSIS TOOLS

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

Question #	Max Pts	Category	Score
1	5	DSN	
2	10	DSN	
3	10	DSN	
TOTAL	25		

You must do all 3 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. For each coding question, assume that all of the necessary includes (stdlib, stdio, math, string) for that particular question have been made.

1) (5 pts) DSN (Recursive Coding)

Write a ***recursive*** function that returns the number of bits set to 1 in the binary representation of its input parameter, *n*. **(Note: In order to receive full credit, your function's run time must be O(b), where b is the total number of bits in n. Since this isn't the bitwise operator question, you don't HAVE to use bitwise operators for full credit, but that's probably the most natural route to the solution.)**

```
int numBitsOn(int n) {

    // Base case - 2 pts
    if (n == 0)
        return 0;

    // 1 pt extract lsb, 1 pt add, 1 pt recursive call
    return (n&1) + numBitsOn(n>>1);
}
```

Note: Bitwise operators aren't needed, so n%2 == 1 for the bit check and n//2 for the input to the recursive call also get full credit. Also, one can make a second if statement and get equivalent behavior, so just watch out for correct solutions that are expressed differently and make sure to award all of these full credit. Map the points as shown above for each subtask.

Correct non-recursive solution: 3/5
Non-recursive solution with minor bug: 2/5

2) (10 pts) DSN (Sorting)

The critical part of the Quick Sort algorithm is the partition. One important part of the partition is the selection of the partition element. In general, it's better to have a "middle value" relative to the other values in the subarray to be sorted to be chosen as the partition element. One simple strategy to improve the probability of a good partition element is to select three items at random from the subarray to be sorted and use the middle value of those three as the partition element. In this particular problem, complete the function below so that it takes in an array, a low index to the array and a high index to the array, designating a subarray, generates three random indexes in between low and high inclusive (let these be indexA, indexB and indexC), and the returns the corresponding index (either indexA, indexB or indexC) to the middle value of the three designated values array[indexA], array[indexB] or array[indexC]. A function randInt(a, b) is provided for you to call, which returns a random integer in between a and b, inclusive. **(Note: To make the problem a bit easier, there is no need to make sure that indexA, indexB and indexC are all distinct.)**

```
// Pre-condition: low and high are valid indexes into array with
//                high - low > 2
int getPartitionIndex(int array[], int low, int high) {

    int indexA = randInt(low, high);          // 1 pt
    int indexB = randInt(low, high);          // 1 pt
    int indexC = randInt(low, high);          // 1 pt

    if (array[indexA] <= array[indexB]) {     // 3 pts looking at
        if (array[indexA] >= array[indexC])   // array values not the
            return indexA;                    // index values.
        else if (array[indexB] <= array[indexC])
            return indexB;                    // 4 pts for the middle
        else                                  // logic
            return indexC;
    }
    else {
        if (array[indexB] >= array[indexC])
            return indexB;
        else if (array[indexA] <= array[indexC])
            return indexA;
        else
            return indexC;
    }

}

int randInt(int a, int b) {
    int base = (rand()<<15) + rand();
    return a + base%(b-a+1);
}
```

3) (10 pts) DSN (Backtracking)

Consider printing out all strings of x A's and y B's, where x ≥ y-1 such that no two consecutive letters are Bs, in alphabetical order. For example, if x = 5 and y = 3, one of the valid strings printed would be AABABABA. One way to solve this problem would be to use backtracking, where a string is built up, letter by letter (first trying A, then trying B in the current slot), but skipping trying A, if doing so would leave 2 more Bs to place than As, and skipping trying the B if the previous letter is a B. **Complete the code below to implement this backtracking solution idea.** (Hint: it's always okay to place B as the first letter. But if not placing the first letter, multiple conditions must be checked.)

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

void printAll(char buffer[], int k, int a, int b);
void printWrapper(int x, int y);

void printWrapper(int x, int y) {
    char* buffer = malloc(sizeof(char)*(x+y+1));
    buffer[x+y] = '\0';
    printAll(buffer, 0, x, y);
    free(buffer);
}

void printAll(char buffer[], int k, int x, int y) {

    if (x == 0 && y == 0) {
        printf("%s\n", buffer);
        return;
    }

    if (x > y-1) {
        buffer[k] = 'A' ;

        printAll(buffer, k+1 , x-1 , y );
    }

    if ( y > 0  && ( k == 0  || ( k > 0  && buffer[k-1] == 'A') ) ){

        buffer[k] = 'B' ;

        printAll(buffer, k+1  , x  , y-1 );
    }
}
```

Grading: 1 pt per slot, needs to be perfectly correct to get the point.