Computer Science Foundation Exam

August 25, 2018

Section I A

DATA STRUCTURES

SOLUTION

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

Name: ____

UCFID: ____

NID: ____

Question #	Max Pts	Category	Score
1	10	DSN	
2	10	DSN	
3	5	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) DSN (Dynamic Memory Management in C)

Consider allocating space for an array of arrays, where each of the individual lengths of the different small arrays may differ. For example, we might want 5 arrays, which have lengths 10, 5, 20, 100 and 50, respectively. Write a function `makeArray` that takes in an array of integers itself and the length of that array (so for the example above the first parameter would be the array storing 10, 5, 20, 100 and 50 and the second parameter would have a value of 5) and allocates space for an array of arrays where each of the individual arrays have the lengths specified by the values of the input array. Before returning a pointer to the array of arrays, the function should store 0 in every element of every array allocated.

```
int** makeArray(int* lengths, int numarrays) {

    int** array = malloc(sizeof(int*)*numarrays);
    int i;
    for (i=0; i<numarrays; i++)
        array[i] = calloc(lengths[i], sizeof(int));
    return array;

}
```

Grading:

3 pts initial malloc, (1 pt for **, 1 pt for sizeof(int*), 1 pt *numarrays) could use calloc as well.

2 pts for loop

4 pts for allocating each array (2 pts) and zeroing it out (2 pts), obviously calloc does this faster, but you can also malloc and run a loop inside a loop…

1 pt for the return

2) (10 pts) DSN (Linked Lists)

Consider storing an integer in a linked list by storing one digit in each node where the one's digit is stored in the first node, the ten's digit is stored in the second node, and so forth. Write a ***recursive function*** that takes in a pointer to the head of a linked list storing an integer in this fashion and returns the value of the integer. Assume that the linked list has 9 or fewer nodes, so that the computation will not cause any integer overflows. (For example, 295 would be stored as 5 followed by 9 followed by 2.) Use the struct shown below:

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

int getValue(node *head) {

    if (head == NULL)
        return 0;
    return head->data + 10*getValue(head->next);

}
```

Grading: 2 pts check head == NULL, 1 pt ret 0 (give full credit if base case is LL size 1), 1 pt return, 1 pt access head->data, 1 pt add, 1 pt 10*, 1 pt rec call, 2 pts parameter

3) (5 pts) ALG (Stacks/Queues)

Consider modeling cars lining up at a traffic light in a simulation. Would it be better to utilize a stack in the simulation or a queue, to store the cars? Clearly explain your choice.

A queue would be better. At a standard traffic light, the earlier a car gets to the light, the earlier the car gets to go through the light. Traffic works in this "fair" way - first in, first out.

In thinking about designing this system, we would have each road coming into a traffic light stored as a queue. If a particular light changes from red to green at some point in time, then the cars would get dequeued from that road/light pair and each one would potentially get enqueued into the next queue representing the road/light pair they were traveling to next. A good simulation would be able to handle some of this movement in parallel, carefully keeping track of time and would identify areas of bottlenecks. Namely, if a queue fills up and another car wants to enter the queue, it actually won't get to do so, since it'll get stuck at the previous light, even though it's green! (More than likely all UCF students who drive cars on campus have experienced this wonderful phenomenon.)

Grading: 0 pts if the answer is stack. 2 pts for saying queue, 3 pts for the explanation. The explanation can be as brief as the second sentence in this solution. The second paragraph is further explanation not required for full credit.

Computer Science Foundation Exam

August 25, 2018

Section I B

DATA STRUCTURES

SOLUTION

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

Name: ____

UCFID: ____

NID: ____

Question #	Max Pts	Category	Score
1	10	DSN	
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 <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>. 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 (Binary Search Trees)

Complete writing the function shown below ***recursively***, so that it takes in a pointer to the root of a binary search tree of strings, *root*, and a string, *target*, and returns 1 if the string is contained in the binary search tree and false otherwise. You may assume all strings stored in the tree contain lowercase letters only. In order to receive full credit, your function must run in O(h) time, where h is the height of the binary search tree storing all of the words.

```
typedef struct bstNode {
   struct bstNode *left, *right;
   char str[100];
} bstNode;

int search(bstNode *root, char* target){

    if (root == NULL) return 0;                         // 2 pts
    int cmp = strcmp(target, root->str);

    if (cmp < 0)                                        // 1 pt
        return search(root->left, target);              // 2 pts
    else if (cmp > 0)                                   // 1 pt
        return search(root->right, target);             // 2 pts
    return 1;                                           // 2 pts

}
```

2) (5 pts) ALG (Hash Tables)

There are two hash functions that take in strings as input shown below. Each returns an integer in between 0 and 1,000,002. (Note: 1,000,003 is a prime number.) Which of these two is a better hash function? Explain the weakness in the other function.

```
int f1(char* str) {

    int i = 0, res = 0;
    while (str[i] != '\0') {
        res = (256*res + (int)(str[i]))%1000003;
        i++;
    }
    return res;
}

int f2(char* str) {

    int i = 0, res = 0;
    while (str[i] != '\0') {
        res = (res + (int)(str[i]))%1000003;
        i++;
    }
    return res;
}
```

The better function is f1. This function is expressing the string in base 256, mod 1000003. The other function, f2, is simply adding the Ascii values of the individual characters of the string mod 1000003. One reason this is problematic is that the sum of the Ascii values of most strings tends to be pretty small. Each individual value is near 100 and most everyday strings have no more than 12 or 13 letters. So the probability of getting a hash value less than 1200 is extremely high and the probability of getting a hash value in between 1200 and 1,000,002 is extremely low. A good hash function has a roughly equal probability that any of the hash values will be generated on an arbitrary input. Secondly, any strings that are anagrams of one another are definitively going to create a collision, since order of the characters in a string doesn't at all affect the hash function value returned by f2. Good hash functions are such that there's no formulaic way to create two different inputs that map to the same output in a reliable way.

Grading: There is no need to have the depth of this response for full credit. Give 2 pts for answering that f1 is better, and upto 3 pts for the flaws of f2. Any cogent response which explains why the hash values are unlikely to be equally distributed should get full credit. Decide partial as necessary. Max score for those who pick f2 is 1 out of 5.

3) (10 pts) DSN (Tries)

The word “intention” is such that four of its prefixes, “i”, “in”, “intent” and “intention” are words themselves. Write a function that takes in a pointer to the root of a trie storing a dictionary of words and returns the maximum number of words that are prefixes of a single word. Use the struct definition and function prototype given below.

```
typedef struct TrieNode {
    struct TrieNode *children[26];
    int flag; // 1 if the string is in the trie, 0 otherwise
} TrieNode;

int max(int a, int b) {
    if (a > b) return a;
    return b;
}

int maxNumPrefixWords(TrieNode* root) {

    if (root == NULL) return 0;                      // 2 pts
    int maxChild = 0;
    int i;
    for (i=0; i<26; i++)                             // 2 pts
        maxChild = max(maxChild, maxNumPrefixWords(root->children[i]));
        // 4 pts, 3 pts for rec call, 1 for updating max

    return maxChild + root->flag;                    // 2 pts

}
```

Computer Science Foundation Exam

August 25, 2018

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	5	ANL	
3	10	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)

Below is a program which includes a single function call to the function mysqrt. The function mysqrt includes a while loop. Give an estimate as to how many times that while loop will run during the single function call from main. (Note: a portion of credit for this question will be awarded to the theoretical analysis of the code and the rest of the credit will be awarded to applying that analysis in concert with estimation techniques without a calculator.)

```
int main() {
    printf("%.6f\n", mysqrt(1000));
    return 0;
}

double mysqrt(double n) {
    double low = 1, high = n;
    if (n < 1) {
        low = n;
        high = 1;
    }
    while (high - low > .000001) {
        double mid = (low+high)/2;
        if (mid*mid < n)
            low = mid;
        else
            high = mid;
    }
}
```

The difference between high and low at the beginning of the function call is 1000 – 1 = 999. For the purposes of our rough analysis, we can simply call this 1000. The while loop repeatedly divides this difference by 2 for each iteration. Thus, after k iterations the difference between high and low is $\frac{1000}{2^k}$. Thus, we want the minimum value of k such that:

$$\frac{1000}{2^k} < 10^{-6}$$

$$\frac{10^3}{10^{-6}} < 2^k$$

$$2^k > 10^9$$
$$k > log_2 10^9$$

This value is irrational, but we can give an approximation for 10^9 in terms of a power of 2. Roughly speaking 2^{10} (1024) is fairly close to 1000 = $10^{3.}$ So, to get a reasonable rough estimate, we can substitute 2^{30} for 10^9, which will lead to the value of **k ~ 30.**

Grading: Eqn set up and explanation - 5 pts, solving equation - 3 pts, getting approximation - 2 pts

2) (5 pts) ANL (Algorithm Analysis)

An algorithm to process input data about n cities takes O($n2^n$) time. For *n* = 10, the algorithm runs in 5 milliseconds. How many ***seconds*** should the algorithm take to run for an input size of *n* = 20?

Let the algorithm with input array size n have runtime $T(n) = cn2^n$, where c is some constant.

Using the given information, we have:

$$T(10) = c(10)2^{10} = 5ms$$

$$c = \frac{5ms}{10 \times 2^{10}}$$

$$c = \frac{1ms}{2 \times 2^{10}}$$

$$c = \frac{1ms}{2^{11}}$$

Now, solve for the desired information:

$$T(20) = c(20)2^{20}$$

$$= \frac{1ms}{2^{11}} \times 20 \times 2^{20}$$

$$= 20 \times 2^9 ms = 20 \times 512ms = 10240ms = 10.24\ seconds$$

Grading: 2 pts solving for c, 2 pts for plugging 20 and canceling to get to 10240 ms, 1 pt to answer in seconds.

3) (10 pts) ANL (Recurrence Relations)

Use the iteration technique to solve the following recurrence relation in terms of n:

$$T(n) = 3T(n-1) + 1, \textit{for all integers } n > 1$$
$$T(1) = 1$$

Please give an exact closed-form answer in terms of n, instead of a Big-Oh answer.

(Note: A useful summation formula to solve this question is $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$.)

$$T(n) = 3T(n-1) + 1$$

$$= 3(3T(n-2) + 1) + 1$$

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

$$= 9(3T(n-3) + 1) + 3 + 1$$

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

After k steps, we have: $= 3^k T(n-k) + \sum_{i=0}^{k-1} 3^i$

Let k = n-1, then we have that $T(n) = 3^{n-1}T\big(n-(n-1)\big) + \sum_{i=0}^{n-2} 3^i$

$$= 3^{n-1}T(1) + \sum_{i=0}^{n-2} 3^i$$

$$= 3^{n-1} + \sum_{i=0}^{n-2} 3^i$$

$$= \sum_{i=0}^{n-1} 3^i$$

$$= \frac{3^n - 1}{3 - 1} = \frac{3^n - 1}{2}$$

Grading: 2 pts for iteration with T(n-2), 2 pts for iteration with T(n-3), 2 pts for general guess after k steps. 1 pt for plugging in k = n-1, 3 pts for simplifying that to the final answer.

Computer Science Foundation Exam

August 25, 2018

Section II B

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	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 (Recursive Coding)

The code below returns the number of zeros at the end of n!

```
int zeros(int n) {
    int res = 0;
    while (n != 0) {
        res += n/5;
        n /= 5;
    }
    return res;
}
```

Rewrite this method ***recursively***:

```
int zeros(int n) {

    if (n == 0) return 0; // 2 pts if n==0, 1 pt ret 0
    return n/5 + zeros(n/5); // 1 pt return, 2 pts n/5, 1 pt +
                             // 1 pt rec call, 2 pts n/5.

}
```

2) (5 pts) DSN (Sorting)

Show the state of the array below for each iteration of ***selection sort***:

Original	13	27	12	9	88	45	22	31
1st iteration	13	27	12	9	31	45	22	88
2nd iteration	13	27	12	9	31	22	45	88
3rd iteration	13	27	12	9	22	31	45	88
4th iteration	13	22	12	9	27	31	45	88
5th iteration	13	9	12	22	27	31	45	88
6th iteration	12	9	13	22	27	31	45	88
7th iteration	9	12	13	22	27	31	45	88

Grading: 1 pt per row, row has to be perfect to get the point for that row

3) (10 pts) ALG (Backtracking)

We define a number as Digit Sum Divisible if, for each value of i, the sum of its first i digits is divisible by i. For example, the number 6451 is Digit Sum Divisible because 6 is divisible by 1, 6 + 4 = 10 is divisible by 2, 6 + 4 + 5 = 15 is divisible by 3 and 6 + 4 + 5 + 1 = 16 is divisible by 4. Consider writing a backtracking function that outputs all Digit Sum Divisible numbers of a particular length given a particular prefix. This function would take in a prefix, such as "64" and a number of digits left to add to it (for this example, 2), and the function would print out each Digit Sum Divisible number starting with the digits 64 that are four digits long. The function would use backtracking because instead of adding each possible digit to the given prefix and making a recursive call, it would first check to see if doing so would maintain the divisibility requirement for the next length. (In this example, 640 would be skipped since 6 + 4 + 0 = 10 and 10 isn't divisible by 3.) ***Write out a tree structure that shows each unique prefix that occurs for each recursive call for the specific function call with the prefix 64 and 2 digits left to add to it.*** (Note: The root node of your tree should be 64, each child of 64 should be a three digit number, and each child of those children should have a four digit number. There should be eight leaf nodes in the tree, so make sure to leave enough room for eight nodes at the bottom level. These leaf nodes are the eight numbers the function would print out for this specific call.) **Note: Please do NOT write any code for this problem, just write out the underlined specified task above.**

```
                       64,2
          /              |           \
         642,1          645,1        648,1
     /    |    \      /  |    \      /   \
  6420 6424 6428   6451 6455 6459  6482 6486

(all calls on the last level have a k=0 with them)
```

Grading: 1 pt for level 0, 1 pt for level 1, 1 pt for each item in level 2. If extra items are included, subtract 1, cap at 0. Note: No need to add the value of k for each recursive call in the tree. They are printed in this solution for clarity's sake.