Computer Science Foundation Exam

January 15, 2022

Section A

BASIC DATA STRUCTURES

SOLUTION

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)

Consider the following structures and the main function shown below:

```
typedef struct StringType {
   char *string;
   int length;
} StringType;
```

```
typedef struct Employee {
   StringType *ename;
   double salary;
} Employee;
```

```
#include <string.h>
#include <stdio.h>
int main() {
  //array of employees' names
  char nameList[][50] = {"Adam", "Josh", "Kyle", "Ali", "Mohammed"};
  //array of salaries, where 15.80 is the salary of Adam, 13.50 is
  // the salary of Josh, etc.
  double salaries[5] = {15.80, 13.5, 20.9, 12.99, 10.5};
  Employee *employees = createEmployees(nameList, salaries, 5);
   // Other code here…
   return 0;
}
```

Write a function createEmployees() that takes the list of employees' names, list of their salaries, and length of the list (empCount) as the parameters, **and returns a pointer to a dynamically allocated array of Employee storing the relevant information for empCount employees**. The function dynamically allocates memory for empCount number of employees and assigns the name and salaries for each of them from the input parameters. During this process, the names are stored in the dynamically allocated memory of StringType, and also make sure you assign the length of the name appropriately. Your code should use exact amount of memory needed to store the corresponding names. You may assume no name is longer than 49 characters.

```
Employee* createEmployees(char names[][50], double *salaries, int empCount) {

   Employee *employees = malloc (empCount  * sizeof(Employee));  // 2 pts

   for (int i = 0; i < empCount; i++) {                           // 1 pt

            employees[i].ename = malloc(sizeof(StringType));      // 1 pt
            int length = strlen(names[i])+1;                      // 0 pts

            employees[i].ename->string = malloc(length * sizeof(char)); // 2 pts

            strcpy(employees[i].ename->string, names[i]);         // 1 pt

            employees[i].ename->length = length-1;                // 1 pt

            employees[i].salary = salaries[i];                    // 1 pt
      }
      return employees;                                           // 1 pt
}
```

2) (5 pts) ALG (Linked Lists)

Suppose we have a linked list implemented with the structure below. We also have a function that takes in the head of the list and returns a node pointer.

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

node* something(node* head) {
    node* t = head;
    if(t==NULL || t->next == NULL )  return t;

    while(t->next->next != NULL)
        t = t->next;

    t->next->next = head;
    head = t->next;
    t->next = NULL;

    return head;
}
```

A linked list, **mylist,** has the following nodes: 1 -> 9 -> 6 -> 7 -> 4 -> 8, where 1 is at the head node of the list.

a) What will be the status of the linked list (draw the full list) after following function call.

`mylist = something(mylist);`

Draw the updated linked list after the function call:

`mylist -> 8 -> 1 -> 9 -> 6 -> 7 -> 4`

Grading: 3 pts for a correct list, 1 pt for a reverse list or a list that has a different front element, 0 pts otherwise

b) What general task does the function something perform? Please answer in a single sentence.

The function takes the last node of the list, moves it to the front, and returns a pointer to the front of the resulting list.

Grading: 2 pts (give full credit if the response is regular English and roughly correct, give 1 pt if the answer is in the right direction but has some clear inaccuracies, 0 pts otherwise)

3) (10 pts) DSN (Stacks)

Consider a string mathematical expression can have two kind of parenthesis '(' and '{'. The parenthesis in an expression can be imbalanced, if they are not closed in the correct order, or if there are extra starting parenthesis or extra closing parenthesis. The following table shows some examples:

Expression	Status	Expression	Status
({)}	Imbalanced due to incorrect order of).	({ }))	Imbalanced due to extra)
(( )	Imbalanced due to extra (	({ })	Balanced
{ ( ) }	Balanced		

Write a function that will take an expression in the parameter and returns 1, if the expression is balanced, otherwise returns 0. You have to use stack operations during this process. Assume the following stack definition and the functions already available to you. You may assume that the stack has enough storage to carry out the desired operations without checking.

```
void initialize(stack* s); // initializes an empty stack.
void push(stack* s, char value); //pushes the char value to the stack
int isEmpty(stack* s); // Returns 1 if the stack is empty, 0 otherwise.
char pop(stack* s); // pops and returns character at the top of the stack.
char peek(stack* s); // returns character at the top of the stack.
```

Note: pop and peek return 'I' if the stack s is empty.

```
// Pre-condition: e only contains the characters '(',')','{' and '}'.
int isBalanced(char *e) {
    struct stack s;
    initialize(&s);
    for(int i=0; e[i]!='\0'; i++) {

        if(e[i] == '(' || e[i] == '{')          // 2 pts
            push(&s, e[i]);                     // 1 pt

        else if(e[i] == ')') {                  // 1 pt
            if(pop(&s) != '(') return 0;        // 2 pts
        }
        else if(e[i] == '}') {                  // 1 pt
            if(pop(&s) != '{') return 0;        // 2 pts
        }
    }

    return isEmpty(&s) ;                        // 1 pt
}
```

Computer Science Foundation Exam

January 15, 2022

Section B

ADVANCED DATA STRUCTURES

SOLUTION

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 (Binary Trees)

The goal of a function named *legacyCount()* is to take the root of a binary tree (*root*) and return the number of nodes that contain a value greater than at least one of their ancestors. For example, this function would return **4** for the following tree, since **60** is greater than both of its ancestors (3 and 14), **40** is greater than two of its ancestors (3 and 14) (even though 40 isn't greater than its parent!), **28** is greater than both of its ancestors (9 and 14), and **2** is greater than one of its ancestors (1).

```
          14
        /    \
       3      9
     / \       \
    1   60      28
   /     \
  2       40
```

Our node struct is as follows:

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

To make the code work, *legacyCount*() is a wrapper function for a recursive function called *legacyHelper*(). Included below is the code for legacyCount() as well as the function signature for *legacyHelper*(). Write all of the code for the *legacyHelper*() function. Note: If *root* is NULL, you should return 0.

```
int legacyCount(node *root) {
    if (root == NULL) return 0;
    return legacyHelper(root->left, root->data) +
           legacyHelper(root->right, root->data);
}

int legacyHelper(node* root, int minAncestor) {

      if (root == NULL)
            return 0;

      if (root->data > minAncestor)
            return 1 + legacyHelper(root->left, minAncestor) +
                       legacyHelper(root->right, minAncestor);

      return legacyHelper(root->left, root->data) +
             legacyHelper(root->right, root->data);

}
```

Grading:

+ 2 pt for correct root == NULL base case in recursive function
+1 for if checking root->data vs. smallest Ancestor
+3 for returning the right result in this case (1 pt for 1, 1 pt for each rec call)
+4 for return in other case, 1 pt for each rec call and 1 pt for updating the second parameter in both calls.

2) (5 pts) ALG (Hash Tables)

Suppose have some hash function that produces the following hash values for the following strings.

String	hash value
Wicked	35429
Cheesy	171745742
Lasagna	72457241
For	559079
Dinner	96879

Using the hash values above, insert the strings (one by one, in the order given above, from "wicked" down through "dinner") into the following hash table. Use **quadratic probing** to resolve any collisions. Note that there is a standard technique for dealing with hash values that exceed the length of a table (e.g., values that exceed 9 in the case of this particular table), and it's up to you to use that technique here.

Note: The length of the hash table is **10**.

For	Lasagna	Cheesy	Dinner						Wicked
0	1	2	3	4	5	6	7	8	9

Grading:
+1 point for each item being in the correct cell.

3) (10 pts) DSN (Tries)

Write an **iterative, non-recursive** function that takes the root of a trie (*root*) and a string (*str*) and returns the number of new nodes we would have to add to our trie in order to insert that string. You may assume that *str* is non-NULL, non-empty, and contains lowercase alphabetic characters only (i.e., it won't contain uppercase letters or non-alphabetic characters). However, you must handle the case where *root* is NULL.

Special Restrictions:

a. Please do not use pointer arithmetic (e.g., str + 1).
b. Do not modify or corrupt the trie or the string. (Do not add nodes to the trie!)
c. Do not call *strlen()* repeatedly, as it is an O(k) function (where *k* is the length of the string). If you need to call *strlen()*, find a way to do it only once for the given string.

The trie node struct and function signature are as follows. Do NOT write any helper functions.

```
#include <string.h>
typedef struct TrieNode {
    struct TrieNode *children[26];
    int flag;  // 1 if the string is in our trie, 0 otherwise
} TrieNode;

int newNodeCount(TrieNode* root, char* str) {

    int len = strlen(str);

    for (int i=0; i<len; i++) {
        if (root == NULL) return len-i;
        root = root->children[str[i]-'a'];
    }

    return 0;

}
```

Grading: 2 pts for calling strlen only once.
1 pt for loop through string
2 pts for checking for NULL
2 pts for the return value when hitting NULL
2 pts for advancing to the appropriate next pointer
1 pt for returning 0 in the case the word is in the trie.

Computer Science Foundation Exam

January 15, 2022

Section C

ALGORITHMS ANALYSIS

SOLUTION

Question #	Max Pts	Category	Score
1	5	ANL	
2	10	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) (5 pts) ANL (Algorithm Analysis)

What is the best and worst case runtime for the following algorithm, in terms of the input parameter n? You may assume that the array pointed to by arr is of length n. Give a brief explanation for your answers.

```
int foo(int * arr, int n, int value){

    int cur = 0, jump = n/2;
    while (jump > 0) {
        if (value > arr[cur])
            cur += jump;
        else if (value == arr[cur])
            return cur;

        jump = jump/2;
    }

    return cur;
}
```

The best case run time is O(1). It's possible that on the very first loop iteration that the else if clause that returns cur triggers. In this situation, only a fixed number of statements, all of which are simple, run.

The worst case run time is O(lg n). The number of times the loop runs is controlled by jump. Each time, jump's value divides by 2 and the loop will end the iteration after jump equals 1. Since jump starts out as n/2, if we let k equal the number of loop iterations, then we get the equation $(n/2) / 2^k = 1$. Solving for k in this equation yields $k = \log_2 n - 1$. Since the work in each loop iteration is constant, the run time of O(lg n) follows.

Grading: 1 pt for the best case answer, 1 pt for its justification, 1 pt for the worst case answer, 2 pts for its justification

2) (10 pts) ANL (Algorithm Analysis)

An algorithm processes an array of size $\boldsymbol{r}$ by $\boldsymbol{c}$ in O($\boldsymbol{rc^2}$) time. For an array of size $\boldsymbol{r}$ = 200 and $\boldsymbol{c}$ = 500, the algorithm takes 5.0 seconds. How long, in seconds, will the algorithm take to process an input array of size $\boldsymbol{r}$ = 800 and $\boldsymbol{c}$ = 300? Please express your answer with exactly one digit after the decimal point.

Let T(r, c) = Mrc^2, for some constant M and represent the run time of the algorithm processing the array. Using the given information, we have:

$$T(200, 500) = M(200)500^2 = 5sec$$

$$50{,}000{,}000M = 5sec$$
$$M = 10^{-7}sec$$

Now, we must solve for T(800, 300):

$$T(800, 300) = (10^{-7}sec)(800)300^2 = (10^{-7}\text{sec})(72)(10^6) = 7.2seconds$$

Grading: 2 pts for setting up equation with constant, r and c.
2 pts for solving for the constant (M in what's above)
2 pts for setting up equation for solution
4 pts for properly simplifying the answer to 7.2 seconds (may award partial)

3) (10 pts) ANL (Summations)

With proof, find the ordered pair of values (a, b) which satisfy the equation below?

$$\sum_{k=1}^{2n}(ak+b) = 7n^2 + 3n$$

Simplify the left hand side in terms of a and b to get to this point:

$$\sum_{k=1}^{2n}(ak+b) = 7n^2 + 3n$$

$$\frac{a(2n)(2n+1)}{2} + b(2n) = 7n^2 + 3n$$

$$an(2n+1) + 2bn = 7n^2 + 3n$$

$$2an^2 + (2b+a)n = 7n^2 + 3n$$

In order for this equation to always be true, we have to equate coefficients, giving us the two following simultaneous equations:

$$2a = 7 \qquad\qquad 2b + a = 3$$

Solving the first equation, we find that $a = \frac{7}{2}$. Plugging this into the second equation, we have

$$2b + \frac{7}{2} = 3$$

$$2b = -\frac{1}{2}$$

$$b = -\frac{1}{4}$$

Thus, the desired ordered pair (a, b) is $(\frac{7}{2}, -\frac{1}{4})$.

Grading: 2 pts sum of ak, 1 pt sum of b, 2 pts simplifying expression, 2 pts equating coefficients, 1 pt solving for a, 2 pts solving for b.

Computer Science Foundation Exam

January 15, 2022

Section D

ALGORITHMS

SOLUTION

Question #	Max Pts	Category	Score
1	5	DSN	
2	10	ANL	
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) (5 pts) DSN (Recursive Coding)

Consider the problem of a frog jumping out of a well. Initially, the frog is ***n*** feet below the top of the well. When the frog jumps up, it elevates ***u*** feet. If a jump gets the frog to the top of the well or past it, the frog escapes the well. If not, unfortunately, the frog slips down by ***d*** feet before clinging to the side of the well. (Note that ***d*** < ***u***.) Write a ***recursive*** function that takes in positive integers, ***n***, ***u***, and ***d***, and returns the number of times the frog must jump to get out of the well.

For example, if n = 10, u = 5 and d = 3, the function should return 4. On the first jump, the frog goes from 10 feet below the top to 8 feet below (5-3 is the progress). On the second jump, the frog goes from 8 feet below the top to 6 feet below the top. On the third jump, the frog goes from 6 feet below the top to 4 feet below the top. On the last jump, since 5 feet is enough to clear the top of the well, the frog does not slip down and gets out. In this case, had n = 11, the frog would have also gotten out in 4 jumps.

(Note: Although one can do some math to arrive at an O(1) solution without recursion, please use recursion to simulate the jumping process described as this is what is being tested - the ability to take a process and express it in code, recursively. Also, though this is a toy problem, it's surprisingly similar to the real life process of paying off a loan, though in the latter process, the amount you "slip down" slowly decreases, month after month.)

```
int numJumps(int n, int u, int d) {

    if (u >= n) return 1;                      // 2 pts
    return 1 + numJumps(n-(u-d), u, d);        // 3 pts
}
```

Grading: 1 pt for checking base case, 1 pt for the return in this case.
1 pt for adding one jump, 2 pts for the recursive call

2) (10 pts) ANL (Sorting)

Jesse has mistakenly written his Merge Sort so that instead of making recursive calls on each half of the array (code below), he calls a function that runs an Insertion Sort on each half of the array. You may assume the function insertionSort runs properly and executes the steps of an Insertion Sort. He tests his function on an array of size 100,000 in reverse sorted order, and discovers that instead of running in under one second, his code takes 9 seconds. How long (in seconds) would sorting the same array (100,000 elements in reverse order), ***on the same computer***, using a single Insertion Sort, be expected to take?

To earn full credit, you must justify your answer by looking at the number of simple operations in both algorithms and comparing the differences in multiplicative constants between the two algorithms.

```
void mergeSort(int array[], int low, int high) {
    if (low >= high) return;
    int mid = (low+high)/2;
    insertionSort(array, low, mid);
    insertionSort(array, mid+1, high);
    merge(array, low, mid, high);
}
```

Let the T(n) be the run time of insertion sort on an array of size n. We know that for some constant c, T(n) = cn^2. The cost of a single merge on two arrays of size n/2 is O(n). Let S(n) = dn, for some constant d be the run time of a Merge. This means the run time of the code for "mergeSort" written above would take this much time to sort an array of size n.

$$\boldsymbol{2T\left(\frac{n}{2}\right) + S(n)}$$

For our situation we have n = 100,000 and the amount of time the code took is 9 seconds, plugging in, we find:

$$2c(50000)^2 + d(100000) = 9\ sec$$
$$2(2500000000)c + d(100000) = 9\ sec$$

Clearly, there are an infinite number of solutions for c and d to this equation, since there is only one equation. But, it's fairly reasonable to assume two things: (1) the constants are similar, and (2) the multiplier of c is so much smaller than the multiplier for d, that the second term adds a negligible amount of time (one ten-thousandth roughly, so if I am answering in seconds, there is no change), so for the purposes of this estimation, we can remove that term, so we have: $\boldsymbol{5,000,000,000c = 9\ sec.}$

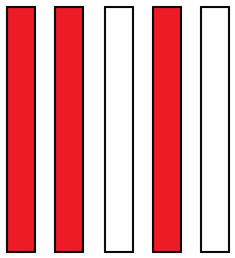
If we were to do one Insertion sort on an array of size 100000, our run time would be c(100,000)^2, so we have:

$$T(100{,}000) = \frac{9sec}{5 \times 10^9} \times (10^5)^2 = \boldsymbol{18\ seconds}$$

Grading: 5 pts for recognizing that the bulk of the code is 2 insertion sorts on arrays of size n/2. 5 pts for comparing the run time of that to 1 insertion sort of an array of size n and recognizing that the latter takes roughly double the time.

3) (10 pts) DSN (Bitwise Operators)

Imagine the task of painting a picket fence with 20 pickets. Let each picket be numbered from 0 to 19, from left to right and initially each is painted white. A pattern is 5 pickets long and can be placed with the pattern's left end aligned with any picket in between number 0 and number 15, inclusive. (If you line the pattern up with any of the pickets 16 through 19, the right end of the pattern goes past the right end of the fence and this isn't allowed.) Below is a picture of an example pattern, with the 3 of the 5 possible pickets painted:



If this pattern was lined up with picket number 4, then pickets 4, 5 and 7 would get painted. Think of the process as a placing a stamp on a portion of the whole fence. We can represent this pattern with the integer 11 ($2^0 + 2^1 + 2^3$), the integer where bits 0, 1 and 3 are set to 1. The bit positions represent, relative to the left end of the pattern, which positions have paint on them.

One way to paint a fence with a pattern is to line up the pattern with a few different picket numbers and apply the pattern. For example, if we lined up this pattern with the pickets at positions 1 and 4, then the pickets that would be painted would be at positions 1, 2, 4, 5 and 7, which corresponds to a bitmask value of 182. (Notice that painting an individual picket more than once leaves it still painted.)

Complete the function below so that it takes in an integer, ***pattern***, in between 0 and 31, inclusive, representing the pattern, an integer array, ***paintLoc***, which stores the locations to line up the pattern with for painting, and ***paintLen***, representing the length of the array ***paintLoc*** and returns a single integer storing the state of the painted fence (for each picket number, k, that is painted, bit k in the returned integer should be set to 1). Each of the values in ***paintLoc*** will be distinct integers in between 0 and 15, inclusive.

```
int paintFence(int pattern, int paintLoc[], int paintLen) {

    int res = 0;                                    // 1 pt
    for (int i=0; i<paintLen; i++)                  // 2 pts
        res = res | (pattern<<(paintLoc[i]));       // 6 pts
    return res;                                     // 1 pt
}
```

Grading Breakdown 3rd line: 1 pt for update res, 2 pts for |, 2 pts for shifting pattern, 1 pt for shifting it by paintLoc[i].