# Computer Science Foundation Exam

## May 18, 2019

## 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 <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 (Dynamic Memory Management in C)

Suppose we have an array of structures containing information about Cartesian points. The struct shown below contains two integers, one for the x coordinate and one for the y coordinate. For this problem, write a function, createPoints, to create some random Cartesian points with each coordinate set to a random integer in between 0 and 10, inclusive.

createPoints takes in the number of points to be created, *numPoints*. Your function should dynamically allocate an array of *numPoints* CartPoints structs and set each of their x and y coordinates with pseudorandom integer values in between 0 to 10, inclusive. You may assume that the random number generator has been seeded already. Your function should return a pointer to the array that was created and initialized.

```
typedef struct CartPoint {
    int x;
    int y;
} CartPoint;

CartPoint* createPoints(int numPoints) {

    int i;

    // LHS = 1 pt, 3 pts RHS
    CartPoint *somePoints = malloc(sizeof(struct CartPoint) * numPoints);

    for(i=0; i<numPoints; i++) {            // 1 pt
        somePoints[i].x = rand() % 11;      // 2 pts
        somePoints[i].y = rand() % 11;      // 2 pts
    }

    return somePoints;                      // 1 pt

}
```

**2)** (10 pts) ALG (Linked Lists)

Suppose we have a queue implemented as a doubly linked list using the structures shown below with head pointing to node at the front of the queue and tail pointing to the node at the end of the queue.

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

struct queue {
     int size;
     struct node *head, *tail;
}
```

Write an enqueue function for this queue. If the queue is already full, return 0 and take no other action. If the queue has not been created yet, return 0 and take no other action. If the queue isn't full, enqueue the integer `item` into the queue, make the necessary adjustments, and return 1. Since there is no fixed size, the queue will be considered full if a new node can't be allocated.

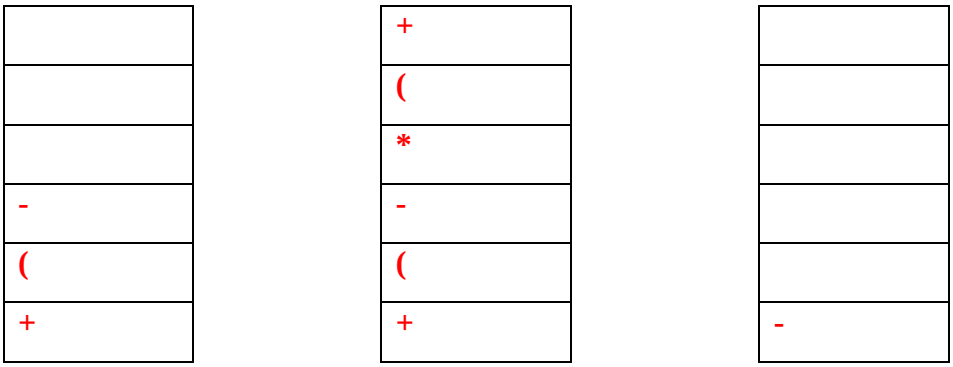
```
int enqueue(queue *thisQ, int item) {
     struct node *newNode = malloc(sizeof(struct node)) ;    // 1 pt
     if(thisQ == NULL) return 0;
     if(newNode == NULL) return 0;

     newNode->data = item;                  // .5 pts

     newNode->next = NULL;                  // .5 pts

     thisQ->size = thisQ->size + 1;         // .5 pts

     if(thisQ->head == NULL) {
          newNode->prev = NULL;             // .5 pts
          thisQ->head = newNode;            // .5 pts
          thisQ->tail = newNode;            // .5 pts
          return 1;
     }

     newNode->prev = thisQ->tail;           // 2 pts

     thisQ->tail->next = newNode;           // 2 pts

     thisQ->tail = newNode;                 // 2 pts
     return 1;
}
```

**Grading Note: Please count total points and round down to record an integer, so 8.5 gets recorded as 8, and 8.0 also gets recorded as 8.**

**3)** (5 pts) DSN (Stacks)

Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (1, 2, and 3) in the infix expression.

| | 1 | | 2 | | 3 | |
|---|---|---|---|---|---|---|
| **A + ( B -** | | **C * (D +** | | **E) ) - F** | | *** G** |




1 2 3

Resulting postfix expression:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | **B** | **C** | **D** | **E** | **+** | ***** | **-** | **+** | **F** | **G** | ***** | **-** | | | | | | | |

**Grading: 1 point for each stack, 2 points for the whole expression (partial credit allowed.)**

# Computer Science Foundation Exam

## May 18, 2019

## 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:** _____

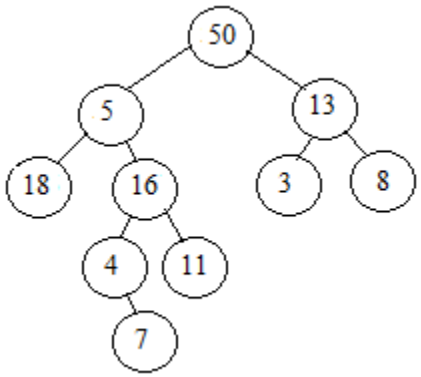
| Question # | Max Pts | Category | Score |
|---|---|---|---|
| 1 | 10 | ALG | |
| 2 | 5 | ALG | |
| 3 | 10 | DSN | |
| TOTAL | 25 | | |

**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) ALG (Binary Trees)

What does the function call `solve(root)` print out if root is pointing to the node storing 50 in the tree shown below? The necessary struct and function are provided below as well. Please fill in the blanks shown below. (Note: the left pointer of the node storing 50 points to the node storing 5, and all of the pointers shown correspond to the direction they are drawn in the picture below.)




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

int solve(bstNode* root) {

    if (root == NULL) return 0;

    int res = root->data;
    int left = solve(root->left);
    int right = solve(root->right);

    if (left+right > res)
        res = left+right;

    printf("%d, ", res);
    return res;
}
```

**18, 7, 7, 11, 18, 36, 3, 8, 13, 50,**

**Grading: 1 pt per correct number in the correct slot.**

**2)** (5 pts) ALG (Hash Tables)

Insert the following numbers (in the order that they are shown from left to right) into a hash table with an array of size 10, using the hash function, H($x$) = $x$ mod 10.

234 344 483 564 814

Show the result of the insertions, assuming any hash collisions are resolved through **quadratic probing.**

| Index | Value |
|---|---|
| **0** | **814** |
| **1** | |
| **2** | |
| **3** | **483** |
| **4** | **234** |
| **5** | **344** |
| **6** | |
| **7** | |
| **8** | **564** |
| **9** | |

**Grading: Give 1 pt for each value listed in the correct spot. If more than one value is in a single spot, give 0 pts for all values in that particular slot automatically.**

**3)** (10 pts) DSN (Tries)

In many word games, the player is given some tiles with letters and must form a word with those tiles. Given a trie that stores a dictionary of valid words and a frequency array storing information of the tiles a player has, determine the number of unique words she can form with those tiles. Complete the function shown below to solve the given problem. Note: the entry in freq[i] represents the number of tiles with the letter 'a' + i. **(Hint: recursing down the trie is exactly like placing a tile down, which means updating the freq array. When you have finished "trying a tile" you have to put it back into your pool, which means editing the freq array again.)**

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

int countWords(TrieNode* root, int freq[]) {

    int res = root->flag ;                                    // 1 pt

    int i;
    for (i=0; i<26; i++) {


        if ( freq[i] == 0 || root->children[i] == NULL )      // 4 pts
            continue;


        freq[i]-- ;                                           // 1 pt


        res += countWords(root->children[i], freq) ;          // 3 pts


        freq[i]++ ;                                           // 1 pt
    }

    return res;
}
```

# Computer Science Foundation Exam

## May 18, 2019

## 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 <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.h, stdio.h, math.h, string.h) for that particular question have been made.**

**1)** (10 pts) ANL (Algorithm Analysis)

Consider storing a table with indexes 0 to N-1, where $N = k^2$, for some positive integer k, that starts with all entries equal to 0 and allows two types of operations: (1) adding some value to a particular index, and (2) querying the sum of all the values in the table from index 0 through index *m*, for any positive integer $m < N$. One way to implement a "table" to handle these two operations is to store two separate arrays, *groups*, of size k and *freq*, of size N. *freq stores* the current value of each index in the table. For the array *groups*, index i ($0 \le i < k$) stores the sum of the values in *freq* from index iN to index (i+1)N-1. (For example, if N = 25, then *groups[2]* stores the sum of the values of freq, from *freq[10]* through *freq[14]*, inclusive.

Determine, ***with proof***, the run-time of implementing operation (1) on this table using this storage mechanism and determine, ***with proof***, the run-time of implementing operation (2) on this table using this storage mechanism. (For example, if N = 100 and we had a query with *m* = 67, to get our answer we would add *groups[0]*, *groups[1]*, *groups[2]*, *groups[3]*, *groups[4]*, *groups[5]*, *freq[60]*, *freq[61]*, *freq[62]*, *freq[63]*, *freq[64]*, *freq[65]*, *freq[66]* and *freq[67]*. Notice that since the ranges 0-9, 10-19, 20-29, 30-39, 40-49, and 50-59 are fully covered in our query, we could just use the groups array for each of those sums. We only had to access the freq array for the individual elements in the 60s.)

Your answers should be Big-Oh answers in terms of N as defined above.

To add a value to a particular index in the table, we must do one update in each of our two arrays. For example, to add x to table index i, we would do these two updates:

```
freq[i] += x;
groups[i/k] += x;
```

Namely, we are redundantly storing our information in two places, so both places must be updated. This runs in O(1) time since each update is a simple statement/command.

A query has a different analysis since we are looking for the sum of items in the table from index 0 through some given index m, where m can range from 0 all the way to N-1. The key observation though is that we will never look at all items in freq for any query. If our query is to a "large value" of m, by adding multiple values in groups, we can do our work more quickly, adding k values at a time. In the worst case, we will add at most k values from the groups array. Notice that since the groups array entries represent table sums of k elements, when we have to add items from the freq array, we will never add more than k of them, since if we were to have added k of them, we could have just added one more value from the groups array. Thus, we do a maximum of k accesses to the groups array and a maximum of k-1 accesses to the freq array, for a total of O(k) time. Since the question asks to respond in terms of N, note that $k = \sqrt{N}$, so the run time of a query operation is $O(\sqrt{N})$

**Grading: 1 pt for update answer, 3 pts for proof, 2 pts for query answer, 4 pts for proof. Latter proof should explain why no more than k accesses of the freq array are necessary to handle any query. Give partial for proofs as you see fit. There is no need for descriptions to be as long or detailed as the solution given above.**

**2)** (5 pts) ANL (Algorithm Analysis)

An algorithm to process a query on an array of size n takes O($\sqrt{n}$) time. For $n = 10^6$, the algorithm runs in 125 milliseconds. How many ***seconds*** should the algorithm take to run for an input size of $n$ = 64,000,000?

Let the algorithm with input array size n have runtime $(n) = c\sqrt{n}$ , where c is some constant.

Using the given information, we have:

$$T(10^6) = c\sqrt{10^6} = 125ms$$

$$c(1000) = 125ms$$

$$c = .125ms = \frac{1}{8}ms$$

Now, solve for the desired information:

$$T(64 \times 10^6) = c\sqrt{64 \times 10^6}$$

$$= \frac{1ms}{8} \times \sqrt{64} \times \sqrt{10^6}$$

$$= \frac{8 \times 1000ms}{8} = 1000ms = 1second$$

**Grading: 2 pts solving for c, 2 pts for plugging 64,000,000 and canceling to get to 1000 ms, 1 pt to answer 1 second as the question requests.**

**3)** (10 pts) ANL (Summations)

Recall that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.

Use the iteration technique to find a Big-Oh bound for the recurrence relation below. Note: you may find the following mathematical results helpful: $2^{log_3 n} = n^{log_3 2}$, and $\sum_{i=0}^{\infty}(\frac{2}{3})^i = 3$. You may use these without proof in your work below.

$$T(n) = 2T\left(\frac{n}{3}\right) + O(n), for\ n > 1$$
$$T(1) = O(1)$$

$$T(n) = 2T\left(\frac{n}{3}\right) + cn$$
$$T(n) = 2(2T\left(\frac{n}{9}\right) + c\left(\frac{n}{3}\right)) + cn$$
$$T(n) = 4T\left(\frac{n}{9}\right) + c(\left(\frac{2n}{3}\right) + n)$$
$$T(n) = 4(2T\left(\frac{n}{27}\right) + c\left(\frac{n}{9}\right)) + c(\left(\frac{2n}{3}\right) + n)$$
$$T(n) = 8T\left(\frac{n}{27}\right) + c(\left(\frac{4n}{9}\right) + \left(\frac{2n}{3}\right) + n)$$

Now that we've done three iterations, we can guess the form of the recurrence after k iterations:

$$T(n) = 2^k T\left(\frac{n}{3^k}\right) + cn(\sum_{i=0}^{k-1}(\frac{2}{3})^i)$$

We want to plug in a value of k to this formula such that $\frac{n}{3^k} = 1$, which occurs when $n = 3^k$. By definition of log, we have that $k = log_3 n$. We will bound the summation by taking it to infinity instead of k-1:

$$T(n) \leq 2^{log_3 n} T(1) + cn(\sum_{i=0}^{\infty}(\frac{2}{3})^i)$$

Now, we can use both given hints to arrive at:

$$T(n) \leq n^{log_3 2} + 3cn = O(n)$$

Note that $\log_3 3 = 1$, so if follows that $\log_3 2 < 1$. Thus, the dominant term is 3cn, which is O(n).

**Grading: Part A - 1 pt for restating original recurrence, 1 pt for getting to second iteration, 2 pts for getting to third iteration, 2 pts for the correct guess of the general form after k iterations, 1 pt for getting the appropriate value of k to plug in, 2 pts to properly simplify both terms, 1 pt to decide which of the two terms is dominant and give the final answer.**

# Computer Science Foundation Exam

## May 18, 2019

## 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 <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 (Recursive Coding)

Consider writing a recursive method that counts the number of paths from a starting (x, y) location on the Cartesian plane to an ending (x, y) location. Let the starting location be *(sx, sy)* and the ending location be *(ex, ey)*, where all four coordinates are integers with $sx \le ex$ and $sy \le ey$, and for each step on a valid path, either 1 must get added to the current x coordinate or 1 must get added to the current y coordinate. In addition, some given locations are disallowed as intermediate locations on the path. **Complete the function shown below** to solve this task. The input to the function takes in *sx*, *sy*, *ex*, *ey* and a two dimensional integer array named *allowed*, such that *allowed[x][y]* is set to 1 if a path is allowed to go on coordinate (x, y) or set to 0 otherwise. It is guaranteed that *(sx, sy)* and *(ex, ey)* are coordinates which are both inbounds and an inbounds function is provided for you. It's not guaranteed that both *(sx, sy)* and *(ex, ey)* are valid locations to be on. In this case, the answer is 0.

```
#define N 10
int inbounds(int x, int y);

int numpaths(int sx, int sy, int ex, int ey, int allowed[][N]) {

    if (!allowed[sx][sy]) return 0;          // 1 pt

    if (sx > ex || sy > ey) return 0;        // 1 pt

    if (sx == ex && sy == ey) return  1;     // 1 pt

    int res = 0 ;                            // 1 pt

    if ( inbounds(sx+1, sy) )                // 1 pt

        res += numpaths(sx+1, sy, ex, ey, allowed );   // 2 pts

    if ( inbounds(sx, sy+1) )                // 1 pt

        res += numpaths(sx, sy+1, ex, ey, allowed );   // 2 pts

    return res;
}

int inbounds(int x, int y) {
    return x >= 0 && x < N && y >= 0 && y < N;
}
```

**Grading Note: To earn 1 pt slots, answers must be perfect. On the two pt lines, award 2 pts if all 5 slots are correct, award 1 pt if at least 2 slots are correct, the order of the if statements doesn't matter but the inbounds check must correspond to the recursive call in its if statement.**

**2)** (5 pts) DSN (Sorting)

In both Merge Sort and Quick Sort, in class we are taught to break down the sorting problem recursively such that the base case is a subarray of size 1 (or 0). It turns out that for both, on average, the implementation is *faster* if we have a base case with a subarray of size in between 20 and 50 and use a $O(n^2)$ sort (typically insertion sort) to sort the base case subarray. Even though insertion sort is $O(n^2)$, why does this modification to the algorithm result in a speed up for both Merge Sort and Quick Sort?

There is a great deal of overhead with recursive calls. Namely, when a new function call is executed, memory is allocated for that function on the call stack and parameters are passed (actual values copied into formal parameter slots), then the function can start running. A vast majority of the total # of recursive calls in the call branches of either of these functions occurs for small arrays. For an array of size 32, at least 31 recursive calls get made. While for large arrays an insertion sort is slower than Merge or Quick sort, for small arrays, the insertion sort if faster because of the overhead of all of these recursive calls. Also, insertion sort only does quick local array accesses so though it does more steps, they are generally faster steps. Thus, if we make our base case larger, what we are doing is substituting something that is slower (a Merge Sort or Quick Sort of 30 or so values) with something that is faster (an Insertion Sort of 30 or so values). Naturally, if we have a set of steps in an algorithm and substitute some of those steps with faster ones, our new algorithm is faster. The key is to set this base case right near that tipping point of the optimal difference between the two sorts for small values.

**Grading: There are quite a few ways to explain this that are valid. The crux of it is that for small arrays, the overhead of the recursion slows the algorithm down so much, it's slower than a simple sort that does more steps but does them without extra function calls and has quick array accesses. Making these substitutions speeds up the overall algorithm since we are substituting something slower for something faster. Give credit based on how complete and convincing the argument given is. Read several responses before calibrating the grading scale.**

**3) (**10 pts) ALG (Backtracking)

Consider an arbitrary permutation of the integers 0, 1, 2, …, *n*-1. We define the "jumps" in a permutation array *perm* to be the set of values of the form *perm[i] - perm[i-1]*, with $1 \le i \le n-1$. For this problem you will write a backtracking solution count the number of permutations that can be created given a limited set of jumps. The function will take in arrays *perm*, representing the current permutation array, *used*, storing which items were used in the current permutation, *k*, the number of fixed items in the current permutation, *jumps*, an array storing the valid jumps allowed, and *len*, representing the length of the *jumps* array. The length of the *perm* and *used* arrays will be the constant N. Note that the jumps array contains both positive and negative values. For example, the permutation 3, 0, 2, 1 has the following jumps: -3, 2 and -1. **Complete the framework that has been given below to solve the problem.**

```
#include <stdio.h>
#define N 10

int numperms(int perm[], int used[], int k, int* jumps, int len) {
    int i, j, res = 0;

    if (k == N) return 1;           // Grading 1 pt per slot, must
    for (i=0; i<N; i++) {           // be correct to earn point.

        if (used[i]) continue;

        int flag = 0;
        if (k == 0)
            flag = 1;
        else {
            for (j=0; j < len; j++)

                if ( i-perm[k-1] == jumps[j])

                    flag = 1;
        }

        if (flag) {
            used[i] = 1;
            perm[k] = i;
            res += numperms(perm, used, k+1, jumps, len);
            used[i] = 0;
        }
    }

    return res;
}
```