Computer Science Foundation Exam

January 14, 2023

Section A

BASIC DATA STRUCTURES

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

SOLUTION

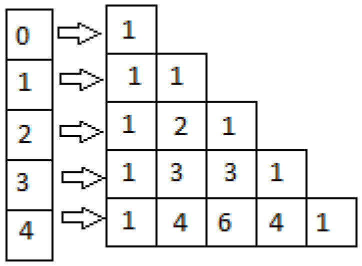
Question #	Max Pts	Category	Score
1	10	DSN	
2	10	DSN	
3	5	ALG	
TOTAL	25	----	

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)

Using 0-based indexing, on row i of Pascal's Triangle, there are i+1 positive integer values. One way we can efficiently store the triangle is to allocate the correct amount of memory for each row. Here is a picture of the first five rows of the triangle (rows 0 through 4, inclusive.):



If the name of the array is **tri**, then the rule to fill in the entries in the table are as follows:

```
tri[i][0] = 1, for all positive ints i
tri[i][i] = 1, for all positive ints i
tri[i][j] = tri[i-1][j-1]+tri[i-1][j], for all ints j, 0 < j < i
```

Write a function that takes in an integer, ***n***, dynamically allocates an array of ***n*** arrays, where the ith array (0-based) is allocated to store exactly i+1 ints, fills the contents of the array with the corresponding values of Pascal's Triangle as designated above, and returns a pointer to the array of arrays. **You may assume that 1 < n < 31.**

```
int** getPascalsTriangle(int n) {


    int** tri = malloc(sizeof(int*)*n);                 // 2 pts

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

        tri[i] = malloc(sizeof(int)*(i+1));             // 2 pts
        tri[i][0] = tri[i][i] = 1;                      // 1 pt

        for (int j=1; j<i; j++)                         // 1 pt
            tri[i][j] = tri[i-1][j-1] + tri[i-1][j];    // 2 pts
    }

    return tri;                                         // 1 pt


}
```

Grading Notes: Take off an integer number of points. For two small errors that you believe are each worth less than a point, take off 1 pt total. It there's only one tiny error (say one dot instead of arrow) correct it and give full credit.

2) (10 pts) DSN (Linked Lists)

Consider using a linked list to store a string, where each node, in order, stores one letter in the string. The struct used for a single node is included below. Write a function that takes in two pointers to linked lists storing 2 strings, and prints out the letters in the string in alternating order, starting with the first letter of the first string. If one string runs out of letters, just skip over it. For example, if the two strings passed to the function were "hello" and "computer", then the function should print "hceolmlpouter".

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

void printMixed(node* word1, node* word2) {

    while (word1 != NULL || word2 != NULL) {           // 2 pts

        if (word1 != NULL) {                           // 1 pt
            printf("%c", word1->letter);               // 2 pts
            word1 = word1->next;                       // 1 pt
        }

        if (word2 != NULL) {                           // 1 pt
            printf("%c", word2->letter);               // 2 pts
            word2 = word2->next;                       // 1 pt
        }
    }

}
```

Grading: If a student follows the method above, it should be fairly easy to award points. There are many other ways to solve this however, most of which will probably take more code. Here is an alternate "idea scheme" for grading responses that use a different approach:

1 pt – attempting to iterate through both lists.
1 pt – advancing a pointer through word1
1 pt – advancing a pointer through word2
2 pts – some sort of artifact to get alternating behavior
2 pts – printing each letter in each list (1 pt for each list)
3 pts – avoiding NULL pointer issues

It is likely that partial credit might be awarded for that last criteria of avoiding NULL pointer issues (ie in some places the solution avoids them, but not all). Give partial on this as you see fit.

3) (5 pts) ALG (Stacks)

Evaluate the following postfix expression shown below, using the algorithm that utilizes an operand stack. Put the value of the expression in the slot provided and show the state of the operand stack (in this case the stacks should just have numbers in them) at each of the indicated points in the expression:

```
                                   A                B                C
4   3   +   5   2   -   *   8   7   9   +   +   6   /   9   4   -   *   -
```

7	**6**	**5**
8	**24**	**4**
21	**21**	**21**
A	**B**	**C**

Note: A indicates the location in the expression **AFTER** 7 but before 9.
B indicates the location in the expression **AFTER** 6 but before the division.
C indicates the location in the expression **AFTER** the subtraction but before the multiplication.

Value of the Postfix Expression: **1**

Grading: 1 pt for each stack (all or nothing), 2 pts for the final answer (all or nothing)

Computer Science Foundation Exam

January 14, 2023

Section B

ADVANCED DATA STRUCTURES

SOLUTION

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 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 and Binary Trees)

Write a recursive function called *prune()* that takes the root of a binary tree and deletes all leaf nodes from the tree. Be sure to update any pointers that need to be updated to account for deleted nodes, carefully avoid any potential segmentation faults, and avoid memory leaks. (You may assume all the nodes in the tree have been dynamically allocated.)

For example:

```
           Initial Tree                              Resulting Tree
   (leaf nodes underlined and bold)                  (after pruning)

                 18                                        18
                /  \                                         \
              25    1                                         1
                   / \                                         \
                  2   9                                         9
                       \
                        16
```

This function should return NULL if the root it receives gets deleted. Otherwise, it should return *root* itself. Note that if *root* is NULL, the function should simply return NULL without taking any further action.

You cannot write any helper functions for this problem. All your work must be contained in a single function called *prune()*. The node struct definition and function signature are as follows:

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

node *prune(node *root)
{
      if (root == NULL)       // 1 pt
            return NULL;

      if (root->left == NULL && root->right == NULL)  // 1 pt for leaf check
      {
            free(root);                               // 2 pts for freeing
            return NULL;                              // 1 pt for returning NULL
      }

      // 2 pts for setting children to appropriate values somehow
      // 2 pts for appropriate recursive calls
      root->left = prune(root->left);               // We could check for non-NULL
      root->right = prune(root->right);             // before making the recursive
                                                    // calls, but it is not strictly
      // 1 pt for returning root                    // necessary.
      return root;
}
```

2) (5 pts) ALG (Hash Tables)

Consider the following problem: Suppose we have a rather large text file that contains up to 50 million strings of 9-digit integers (each separated by a space), and we know that **exactly one** of those integers occurs twice. (The rest, we know for certain, occur only once.) Suppose our goal is to find the single 9-digit integer that occurs twice. Some clever cheddar of a computer scientist thinks they have come up with a great solution to this problem. They have a rock-solid implementation of hash tables (written in C) that has been thoroughly tested by the open-source community and is known to be reliable and bug-free. This implementation of hash tables has some great features:

- The hash tables have a lovely, effective, widely-used hash function for dealing with integers.
- The hash tables expand automatically without creating any memory leaks.
- In addition to keeping track of the elements in the hash table, this implementation keeps track of a few extra pieces of information inside the HashTable struct. One of those extra pieces of information is how many collisions have occurred over the lifetime of this hash table. (That value is, of course, initialized to zero (0) by the function used to create new hash tables.)

It is this last property that has caught our clever cheddar's eye. They reason that they can loop through the input file, throw each integer into a hash table one-by-one, and as soon as a collision occurs in the hash table, they must have found the duplicate integer they were searching for! So, they write up a piece of code that effectively does this:

```
function find_that_duplicate(file):
    create a hash table, h
    for each integer, n, in file:
        insert n into h
        if h->num_collisions > 0:
            destroy h
            close file
            return n
```

The idea above is just pseudocode, of course. You may assume that the person implementing this in C is careful to avoid memory leaks and segmentation faults, that there is enough memory to hold all those integers in the hash table, that the hash table implementation works well and is rock solid, and that the code has no syntax errors. You may also assume this function will **only** be called with a file that actually exists, and that we know for sure there is exactly one duplicate integer in that file. (So, we do not need to worry about what happens if there is no duplicate.)

What is wrong with this proposed solution? In your response, be as clear and precise as possible. Long, rambling word salads will not receive credit.

A collision does **not** imply equality. Two distinct elements can collide in a hash table. So, this solution does not necessarily find the duplicate integer. It could return prematurely with an incorrect integer.

Grading: 5 pts for correct answer. (They just need the part about collision not implying equality.) Award only 3 pts if they seem to be on the right track but are being imprecise and possibly cheesing it. Award 1 pt if they say, "duplicate value could end up in different indexes."

3) (10 pts) ALG (Tries and AVL Trees)

(a) (5 pts) (Tries) Suppose we insert the following strings into an initially empty trie:

```
cat    cap    catapult    catalyst    phosphorescent    pancake    pancakes
```

How many nodes will the resulting trie contain? Please count carefully, as credit for this question may be all or nothing

Count the root node, and then below each letter that is the first time that node is created will be underlined:

```
root                →  1
cat                 →  3
cap                 →  1
catapult            →  5
catalyst            →  4
phosphorescent      → 14
pancake             →  6
pancakes            →  1
                    ------
                      35
```

Grading
5 pts for a correct answer, 4 for answering 34, 3 for answering 36, 2 pts (30 – 40),
1 pt – (20 – 50), 0 pts - otherwise

(b) (5 pts) Show what the following AVL would look like after inserting the value 42 and performing any necessary rotations.

```
    18
   /  \
  7    38
   \     \
   12     59
```

```
     18
    /    \
   7      42
    \    /  \
    12  38   59
```

Grading: 5 pts for correct answer,
4 pts if just one node (such as 12) is missing but rebalance was correct,
0 pts otherwise

Computer Science Foundation Exam

January 14, 2023

Section C

ALGORITHM ANALYSIS

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

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 big O runtime for the following segment of code in terms of N and M? (Note: let min(x, y) denote the minimum of x and y and max(x, y) denote the maximum of x and y. You may use either of these in your answer. **In addition to your answer, please provide justification for your answer.**

```
int fun(int N, int M, int ** grid) {
   int a1 = 0, a2 = 0;
   for (int i = 0; i < N || i < M; i++) {
      if (i < N) a1 += grid[i][0];
      if (i < M) a2 += grid[0][i];
   }
   if (a1 < a2) return a2;
   return a1;
}
```

The value i increments by 1 until it passes both the value N and M (technically not the sum). The value will run max of N and M time. In terms of Big-Oh we can express the solution as **O(max(N, M))**.

Since the sum of N + M can be bounded by 2*max(N, M), an equally valid way to express the same answer is **O(N + M).**

Grading: 5 pts for correct answer; 2/5 for O(n) or O(m), 3/5 for O(n || m), 3/5 for O(min(n,m))

2) (10 pts) ANL (Algorithm Analysis)

You are using an algorithm that can multiply 2 N-digit integers in $O(N^{1.5})$ time. It takes $(10/13)^3$ seconds to multiply 2 numbers that have 100,000 digits. What is the expected number of digits of 2 numbers we could multiply together that would take exactly 1 second? **Please show all your work, including algebraic simplification, which is part of what is being tested with this question.**

Let T(N) be the run time for the algorithm to multiply two N-digit integers. Using the given information, there exists a constant c such that $T(N) = cN^{1.5}$

Let's plug in N = 100,000 into this equation to find c:

$(\frac{10}{13})^3 = c(100{,}000)^{3/2}$ **Grading: 2 pts**

$\frac{10^3}{13^3} = c(10)^{5\times(\frac{3}{2})}$ **Grading: 1 pt**

$\frac{10^3}{13^3} = c(10)^{\frac{15}{2}}$ **Grading: 1 pt**

$c = \frac{10^3}{13^3 \times 10^{\frac{15}{2}}} = \frac{1}{13^3 \times 10^{\frac{9}{2}}}$ **Grading: 1 pt (5 pts total to find c)**

Now, let N be the answer to the question, plugging in 1 second for the time and the value of c found above:

$1s = \frac{1}{13^3 \times 10^{\frac{9}{2}}} \times N^{1.5}$ **Grading: 2 pts**

$N^{1.5} = 13^3 \times 10^{4.5}$ **Grading: 1 pts**

$N = 13^{\frac{3}{1.5}} \times 10^{\frac{4.5}{1.5}} = 13^2 \times 10^3 = \mathbf{169{,}000}$ **Grading: 2 pts**

3) (10 pts) ANL (Recurrence Relations)

What is the closed form for the following recurrence relation, T(N)? For full credit your work must be shown. Note: Your answer should be **EXACT** and not a Big-Oh bound.

$$T(N) = 2T(N-1) + N \text{ (for N ≥ 1)}$$
$$T(0) = 0$$

Note: this problem does not fit the form for using master's theorem. Use the iteration technique:

$T(N) = 2T(N-1) + N$ **Grading: 1pt**

$$T(N-1) = 2T(N-1-1) + N - 1$$
$$T(N-1) = 2T(N-2) + N - 1$$
$$T(N) = 2(2T(N-2) + N - 1) + N$$

$T(N) = 4T(N-2) + 2(N-1) + N$ **Grading: 1 pt**

$$T(N-2) = 2T(N-2-1) + N - 2$$
$$T(N-2) = 2T(N-3) + N - 2$$
$$T(N) = 4(2T(N-3) + N - 2) + 2(N-1) + N$$

$T(N) = 8T(N-3) + 4(N-2) + 2(N-1) + N$ **Grading: 1 pt**

General Form after *k* iterations

$T(N) = 2^k T(N-k) + \sum_{i=0}^{k-1} 2^i (N-i)$ **Grading: 2pts**

We want *k* to be large enough to terminate the recursion i.e. *T* (*N* - *k*) = *T* (0). Thus *N* - *k* = 0, or *N* = *k*

Plugging in *N* for *k* we get

$T(N) = 2^N T(0) + \sum_{i=0}^{N-1} 2^i (N-i)$ **Grading: 2 pts**

$$T(N) = 2^N 0 + (2^{N-1}(1) + 2^{N-2}(2) + 2^{N-3}(3) + \ldots + 2^0(N))$$
$$T(N) = (2^{N-1}(1) + 2^{N-2}(2) + 2^{N-3}(3) + \ldots + 2^0(N))$$

Multiply the equation above through by 2 to obtain: **Grading: 1 pt**

$$2T(N) = (2^N(1) + 2^{N-1}(2) + 2^{N-2}(3) + \ldots + 2^1(N))$$

Take the difference of 2T(N) and T(N)

$$2T(N) - T(N) = (2^N(1) + 2^{N-1}(2) + 2^{N-2}(3) + \ldots \qquad + 2^1(N)) -$$
$$(2^{N-1}(1) + 2^{N-2}(2) + 2^{N-3}(3) + \ldots + 2^1(N-1) + 2^0(N))$$
$$T(N) = (2^N(1) + 2^{N-1}(1) + 2^{N-2}(1) + \ldots + 2^1(1) - 1(N))$$

$$T(N) = (\mathbf{2^{N+1}} - \mathbf{2}) - \boldsymbol{N}$$

Grading: 2pts to subtract and get to final answer.

Grading Notes: Give 6/10 if wrong general guess but the rest of it is correct after the general guess. (So 4 pts for upto the general part, 2 pts for solving the incorrect recurrence.)

Computer Science Foundation Exam

January 14, 2023

Section D

ALGORITHMS

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

SOLUTION

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 (Recursive Coding)

Define an extreme permutation of the integers 0 to n – 1 as any permutation where one where each value in the permutation (from left to right) is either the smallest or largest value not yet placed. For example, for n = 6, [0, 1, 5, 2, 3, 4] is an extreme permutation but [0, 5, 2, 4, 1, 3] is not. The latter is not because the only valid values that can be placed where the 2 is are either 1 or 4, the smallest and largest values, respectively, that have not been placed. Complete the recursive function below so that it prints out all extreme permutations of length n. A completed wrapper function has been provided. Note: low represents the lowest unplaced value, high represents the highest unplaced value, and k represents the number of items in the permutation that have already been filled.

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

void printExtremeWrapper(int n);
void printExtreme(int* perm, int n, int low, int high, int k);
void printPerm(int* perm, int n);

void printExtremeWrapper(int n) {
    int* perm = malloc(sizeof(int)*n);
    printExtreme(perm, n, 0, n-1, 0);
    free(perm);
}

void printPerm(int* perm, int n) {
    for (int i=0; i<n; i++) printf("%d, ", perm[i]);
    printf("\n");
}
void printExtreme(int* perm, int n, int low, int high, int k) {

    if (low > high) {
        printPerm(perm, n);
        return;
    }


    perm[k] = low;                              // Grading 2 pts
    printExtreme(perm, n, low+1, high, k+1);    // Grading 2 pts

    if (low == high) return;                    // Grading 2 pts

    perm[k] = high;                             // Grading 2 pts
    printExtreme(perm, n, low, high-1, k+1);    // Grading 2 pts

}
```

Grading notes: It's possible students use a loop to try all possible values in slot k. If they write regular permutation code (so print all perms), then award 4/10 points. If students write the loop but only recurse if the value they put into the slot is low or high, give full credit (this works). Note that no order is imposed by the question so students can try high before low. Give 8 out of 10 if they repeat that recursive call twice (ie. always call with both low and high.)

2) (10 pts) DSN (Sorting)

Complete the following merge function that is used as part of the merge sort process. The function performs merge operation from left to mid and mid+1 to right index of the array.

```
void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1; //size of the left array
    int n2 =  right - mid; //size of the right array

    /* create temp arrays */
    int *L = (int*) malloc(n1*sizeof(int)); //left array
    int *R = (int*) malloc(n2*sizeof(int)); //right array

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of left subarray
    j = 0; // Initial index of right subarray
    k = left; // Initial index of merged subarray
    // Complete the remaining part of the code that will
    // merge L and R array into arr


    while(i<n1 || j< n2) {                         // Grading: 2 pts
        if( j==n2 || (i<n1 && L[i] < R[j])){ // Grading: 3 pts

            arr[k] = L[i];                         // Grading: 1 pt
            i++;                                   // Grading: 1 pt
        }
        else {
            arr[k] = R[j];                         // Grading: 1 pt
            j++;                                   // Grading: 1 pt
        }
        k++;                                       // Grading: 1 pt
    }
}
```

Grading Notes: Longer ways to do this…typical errors will involve AOOB or not copying the last items after one list is exhausted. (Stops before end && in loop = 8/10, Copy back all items into arr but wrong order, for example alternating, – 4/10, any AOOB but otherwise correct – 8/10)

3) (5 pts) ALG (Base Conversion)

Convert 375 in base 10 to binary. **Please show your work and put a box around your final answer.**

```
2 | 375
2 | 187  R 1
2 |   93  R 1
2 |   46  R 1
2 |   23  R 0
2 |   11  R 1
2 |     5  R 1
2 |     2  R 1
2 |     1  R 0
```

375 in base 10 converts to 101110111_2 (in binary).

Alternate solution is to go through the list 256, 128, 64, 32, 16, 8, 4, 2, 1 and greedily subtract out the largest value possible from 375 until arriving at 0. (Though this method isn't taught in the class, it will be accepted.)

Grading: 5 pts for all correct solutions with reasonable work.
4 pts for any solution using a valid method with exactly 1 arithmetic error.
3 pts for any solution using a valid method with exactly 2 arithmetic errors.
2 pts for any solution that indicates repeated division by 2 but has other errors
1 pt for writing down some powers of 2 or any sort of valid scratch work.