Computer Science Foundation Exam

December 16, 2016

Section I A

DATA STRUCTURES

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

SOLUTION

Question #	Max Pts	Category	Passing	Score
1	10	DSN	7	
2	5	ALG	3	
3	10	ALG	7	
TOTAL	25		17	

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)

Consider the following struct, which contains a string and its length in one nice, neat package:

```
typedef struct smart_string {
  char *word;
  int length;
} smart_string;
```

Write a function that takes a string as its input, creates a new *smart_string* struct, and stores a **new copy of that string** in the *word* field of the struct and the length of that string in the *length* member of the struct. The function should then return a pointer to that new *smart_string* struct. Use dynamic memory management as necessary. The function signature is:

```
smart_string *create_smart_string(char *str) {

   smart_string *s = malloc(sizeof(smart_string));

   s->length = strlen(str);

   s->word = malloc(sizeof(char) * (s->length + 1));

   strcpy(s->word, str);

   return s;

}
```

Grading (7 pts): 1 pt for smart_string declaration, 1 pt for first malloc, 1 pt for setting length correctly, 1 pt for sizeof(char) in second malloc, 1 pt for (s->length + 1) in second malloc, 1 pt for using strcpy, and 1 pt for correct return statement. They can also use calloc(). Please deduct a point for other large, obvious errors (such as using the . operator instead of the -> operator).

Now write a function that takes a *smart_string* pointer (which might be NULL) as its only argument, frees all dynamically allocated memory associated with that struct, and returns NULL when it's finished.

```
smart_string *erase_smart_string(smart_string *s) {

   if (s != NULL)
   {
      free(s->word); // This is safe, even if word is NULL.
      free(s);
   }
   return NULL;
}
```

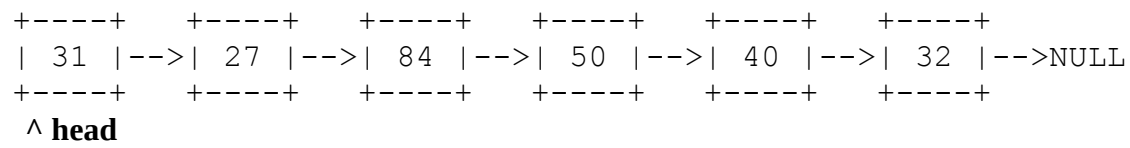
Grading (3 pts): 1 pt for checking s != NULL, 1 pt for free(s->word), and 1 pt for free(s).

2) (5 pts) DSN (Linked Lists)

Consider the following function, which takes the head of a linked list as its only input parameter:

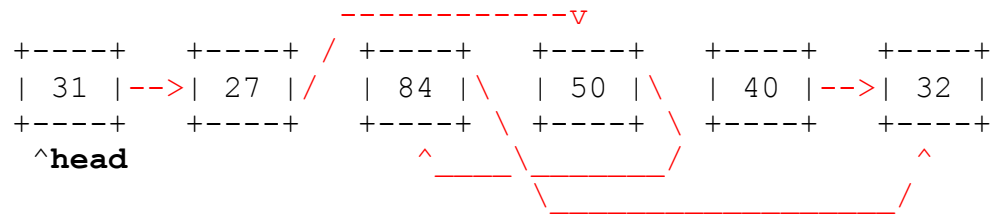
```
node *funky(node *head) {
  if (head == NULL)
    return head;
  if (head->next != NULL && (head->next->data % 2) == 0) {
    head->next = yknuf(head->next->next, head->next);
    head = funky(head->next->next);
  }
  else if (head->next != NULL)
    head->next = funky(head->next);
  return head;
}

node *yknuf(node *n1, node *n2) {
  n2->next = n1->next->next;
  n1->next = n2;
  return n1;
}
```

Suppose someone passes the head of the following linked list to the *funky()* function:



The function call is: `funky(head);`

This program is going to crash spectacularly, but before it does, it will change the structure of the linked list a bit. Trace through the function call(s) and draw a new diagram that shows how the links in this linked list will be arranged at the moment when the program crashes. (In particular, show where the next pointer for each node except the one storing 32 will point.)



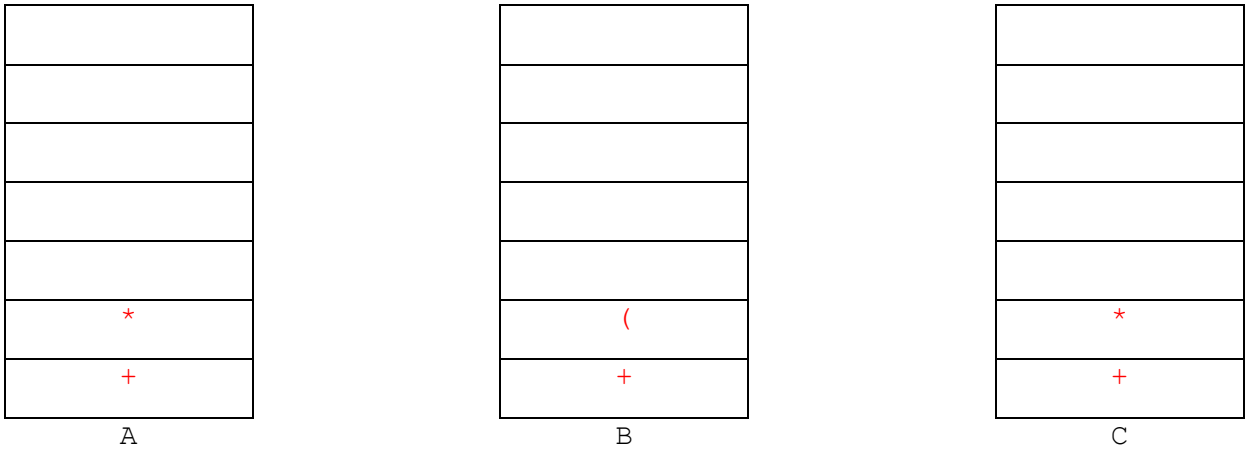
In other words, node 31 points to node 27, 27 points to 50, 50 points back to 84, 84 points to 32, and 40 also points to 32.

Grading (5 pts): 1 pt for each correct 'next' pointer.

3) (10 pts) DSN (Stacks: Infix to Postfix Conversion)

Convert the following from an infix expression to a postfix expression. Show the state of the operator stack at each of the indicated points (A, B, and C):

```
14  +  18  *   9  -  3  +  ( 4   -  8 )  *  ( 9  -  6 )  /  2
           ^              ^                          ^
           A              B                          C
```



Give the final postfix expression here:

14 18 9 * + 3 - 4 8 - 9 6 - * 2 / +

What is the final value of this postfix expression?

167

Grading (10 pts): 5 pts for the correct stack contents (give 1 pt for each symbol, but subtract a pt for each incorrect symbol, but then award a max of 5 and min of 0 pts for this part – no negative scores). 4 pts for correct postfix expression (-0.5 pts for each error, but take the floor of the number of points they earn for this part), and 1 pt for final value (167).

Computer Science Foundation Exam

December 16, 2016

Section I B

DATA STRUCTURES

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

SOLUTION

Question #	Max Pts	Category	Passing	Score
1	10	DSN	7	
2	5	ALG	3	
3	10	DSN	7	
TOTAL	25		17	

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)

A binary search tree is considered "lopsided" if the root's left subtree height and right subtree height differ by more than one (i.e., the left subtree is more than one level deeper or shallower than the right subtree). This is different from the definition of "balanced" that comes up in relation to AVL trees, because the "lopsided" property only applies to the root of the tree – not every single node in the tree.

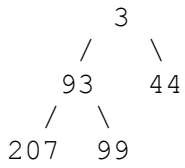
Write a function, *isLopsided()*, that takes the root of a binary search tree and returns 1 if the tree is lopsided, and 0 otherwise. You may write helper functions as you see fit. The node struct and function signature are as follows:

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

int isLopsided(node *root)
{
   int lHeight, rHeight, diff;

   if (root == NULL) return 0;

   lHeight = height(root->left);
   rHeight = height(root->right);
   diff = lHeight - rHeight;

   return (diff >= 2 || diff <= -2);
}

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

int height(node *root)
{
   if (root == NULL) return -1;
   return 1 + max(height(root->left), height(root->right));
}
```

Grading:
2 pts for checking root == NULL in isLopsided()
2 pts for realizing they need to write a separate height() function
3 pts for the rest of the logic in isLopsided, such as the diff check (award partial credit as you deem fit)
3 pts for the logic in height() (award partial credit as you deem fit)
Note - for this particular question, if height returns 0 for null tree, that is fine since all that matters for correctness here is relative height.

2) (5 pts) ALG (Advanced Data Structures: Binary Heaps)

(a) (2 pts) Is the following tree a valid minheap? If so, give an array representation of this minheap. If not, explain why it's not a minheap.



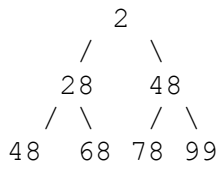
Yes, this is a valid minheap. Both of the following array representations are acceptable:

3	93	44	207	99
0	1	2	3	4

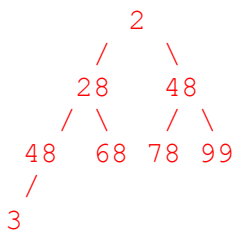
(?)	3	93	44	207	99
0	1	2	3	4	5

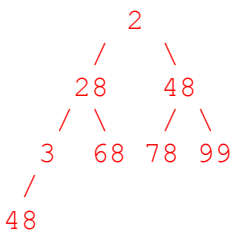
Grading (2 pts): -1 for a very minor error. Otherwise, it's 2 pts if correct, 0 if not.

(b) (3 pts) Insert the value 3 into the following minheap. Clearly show each step of the process.

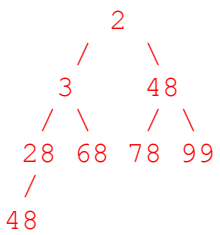


Step 1: Insert 3 into the next available position (left child of 48)





Step 2: The 3 percolates up (twice).



Grading (5 pts):
2 pts: The 3 is inserted into the correct position to begin with.
2 pts: The 3 ends up in the correct position after percolating up.
1 pt: Nothing else ended up in the wrong place after percolating.

3) (10 pts) DSN (Advanced Tree Structures: Tries)

Write a recursive function that takes the root of a trie and counts how many odd-lengthed strings there are in the trie. For example, if the trie contains the empty string (“”), “bananas”, “avocados”, and “randomness”, the function should return 1, because only one of those strings has a length that is odd (“bananas”).

We will make our initial call to your function like so: `countOddStrings(root, 0);`

Part of the fun in this problem is figuring out what to do with that second parameter.

Please do **NOT** write any helper functions. Restrict yourself to the function whose signature is given below.

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

int countOddStrings(TrieNode *root, int k) {

  int total = 0;

  if (root == NULL)
    return 0;

  if (k % 2 == 1 && root->flag == 1) total++;

  for (i = 0; i < 26; i++)
    total += countOddStrings(root->children[i], k + 1);

  return total;
}
```

Note: An alternate solution is to pass (k + 1) % 2 in the recursive call, so the k keeps toggling between 0 (even) and 1 (odd).

Grading (10 pts):
1 pts for checking root == NULL as a base case.
2 pts the line that increments total if k % 2 == 1 and root->flag == 1 (award partial credit)
2 pts for the for loop (double check that the range of ‘i’ is valid)
2 pts for the recursive call (make sure they use k + 1 and not k++ or ++k, as the latter two cause issues)
2 pts for capturing the return values of the recursive calls and using them to increment total
1 pt for returning total

Computer Science Foundation Exam

December 16, 2016

Section II A

ALGORITHMS AND ANALYSIS TOOLS

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

SOLUTION

Question #	Max Pts	Category	Passing	Score
1	10	ANL	7	
2	5	ANL	3	
3	10	ANL	7	
TOTAL	25		17	

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) ALG (Algorithm Analysis)

Consider the following function:

```
int* makeArray(int n) {
    int* array = calloc(n, sizeof(int));
    int i, j;
    for (i=0; i<n; i++)
        for (j=i; j<n; j = j+i+1)
            array[j]++;
    return array;
}
```

(a) (1 pt) Assuming that the function is called with a value of n = 12 or greater, what will array[11] store when the array is returned from the function?

6 (**Grading:** 1 pt all or nothing)

(b) (3 pts) In general, what will array[k] store when the function completes, assuming the function was called with an input value of k+1 or greater?

array[k] will store the number of divisors of k+1.
(**Grading**: 3 pts mostly all or nothing, perhaps partial credit if there is some mention of divisibility but if the answer isn't correct.)

(c) (2 pts) Write a summation that provides a tight upper bound on the number of times the line of code array[j]++ runs when the function is called with the input value n.

$$\sum_{i=1}^{n} \frac{n}{i}$$

(**Grading:** 1 pt bounds, 1 pt function inside sum)

(d) (4 pts) Utilizing the fact that $\sum_{i=1}^{n} \frac{1}{i} = O(lgn)$, determine the run time of the function makeArray for an input of size n. (Note: This run time is equal to the summation from part c.)

$$\sum_{i=1}^{n} \frac{n}{i} = n \sum_{i=1}^{n} \frac{1}{i} = nO(lgn) = O(nlgn)$$

(**Grading:** 2 pts pulling out n, 1 pt plugging in given info, 1 pt final answer)

2) (5 pts) ANL (Algorithm Analysis)

An image processing algorithm takes $O(n^3)$ time to run to filter an n x n pixel picture. If it takes 8 seconds to process a 1024 x 1024 pixel picture, how long will it take to process a 1536 x 1536 pixel picture?

Let T(n) be the run time of the algorithm. $T(n) = cn^3$ for some constant c. Use the first piece information to set up an equation to solve for c:

$$T(1024) = c(1024^3) = 8sec$$
$$c = \frac{8}{1024^3} sec$$

Now, solve for T(1536):

$$T(1536) = c(1536^3) = \frac{8sec}{1024^3} \times 1536^3 = (8\text{sec}) \times \left(\frac{1536}{1024}\right)^3 = (8\text{sec}) \times \left(\frac{3}{2}\right)^3 = 27sec$$

Grading: 2 pts solving for c, 2 pts plugging in 1536, 1 pt simplifying to 27 sec.

3) (10 pts) ANL (Summations and Recurrence Relations)

(a) (5 pts) Determine the following sum in terms of n: $\sum_{i=1}^{2n-1}(3i-2)$.

$$\sum_{i=1}^{2n-1}(3i-2) = 3(\sum_{i}^{2n-1} i) - 2\sum_{i=1}^{2n-1} 1$$
$$= \frac{3(2n-1)(2n)}{2} - 2(2n-1)$$
$$= (2n-1)(3n-2)$$
$$= 6n^2 - 7n + 2$$

(**Grading:** 1 pt split, 2 pts formula for i, 1 pt const formula, 1 pt final answer - can leave in either factored for polynomial form.)

(b) (5 pts) Let $T(n) = 3T\left(\frac{n}{2}\right) + n^2$. In using the iteration technique (3 steps) to solve the recurrence, we arrive at an equation of the form: $T(n) = AT\left(\frac{n}{8}\right) + Bn^2$. Find A and B.

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$
$$= 3(3T\left(\frac{n}{4}\right) + (\frac{n}{2})^2) + n^2$$
$$= 3(3T\left(\frac{n}{4}\right) + \frac{n^2}{4}) + n^2$$
$$= 9T\left(\frac{n}{4}\right) + \frac{3n^2}{4} + n^2$$
$$= 9(3T\left(\frac{n}{8}\right) + (\frac{n}{4})^2) + \frac{3n^2}{4} + n^2$$
$$= 9(3T\left(\frac{n}{8}\right) + \frac{n^2}{16}) + \frac{3n^2}{4} + n^2$$
$$= 27T\left(\frac{n}{8}\right) + \frac{9n^2}{16} + \frac{3n^2}{4} + n^2$$

$$= 27T\left(\frac{n}{8}\right) + \frac{37n^2}{16}$$

In follows that A = 27 and B = $\frac{37}{16}$.

(**Grading:** 2 pts to get to second iteration, 3 pts to get to third iteration)

Computer Science Foundation Exam

December 16, 2016

Section II B

ALGORITHMS AND ANALYSIS TOOLS

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

SOLUTION

Question #	Max Pts	Category	Passing	Score
1	10	DSN	7	
2	10	ALG	7	
3	5	ALG	3	
TOTAL	25		17	

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>.

1) (10 pts) DSN (Recursive Coding)

A derangement is a permutation of the integers 1, 2, 3, ..., n such that for all i, $1 \leq i \leq n$, the value in the i^{th} location isn't i. For example, (2, 1, 4, 3) is a derangement of 4 items since the first item isn't 1, the second item isn't 2, the third item isn't 3 and the fourth item isn't 4. But (3, 1, 5, 4, 2) is NOT a derangement of 5 items since the 4th item on this list is 4. Complete the code below so that it prints out all derangements of size n ($2 \leq n \leq 10$), where n is entered by the user.

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

void printD(int n);
void printDRec(int n, int* perm, int* used, int k);
void print(int* perm, int length);

int main() {
    int n;
    printf("Enter the size of your derangement(2-10).\n");
    scanf("%d", &n);
    printD(n);
    return 0;
}
void printD(int n) {
    int perm[MAX];
    int used[MAX];
    int i;
    for (i=0; i<MAX; i++) used[i] = 0;
    printDRec( n, perm, used, 0); // Grading: 1 pt
}

void printDRec(int n, int* perm, int* used, int k) {
    if (k == n) {
        print(perm, n);
        return;
    }

    int i;
    for (i=0; i<n; i++) {
        if (!used[i] && k != i ) {                  // Grading: 3 pts

            perm[ k ] =  i ;        // Grading: 2 pts, exchange i,k okay.

            used[ i ] =  1 ;                        // Grading: 2 pts
            printDRec(n, perm, used, k+1);

            used[ i ] = 0 ;                         // Grading: 2 pts
        }
    }
}

void print(int* perm, int length) {
    int i;
    for (i=0; i<length; i++)
        printf("%d ", perm[i]+1);
    printf("\n");
}
```

2) (10 pts) ALG (Sorting)

(a) (6 pts) In quick sort, when running the partition function, the first step is to choose a random partition element. In some implementations, instead of just choosing a random element, 3 or 5 random elements are chosen and the median of those elements is then selected as the partition element, as opposed to making the partition element a single randomly selected item. What is the potential benefit of using this strategy (median of 3 or median of 5) versus the default strategy of just choosing a single random element?

The potential benefit of this strategy is that the median of 3 or 5 randomly selected items is very likely to be closer to the actual median of the data and this is precisely when Quick Sort will run faster. Quick Sort achieves its best case run-time if every partition element chosen is the median element and its run-time deteriorates as the partition element choices stray further from the median. In the worst case, the partition puts all elements either to the left or right of it, leaving an array almost as big (n-1 elements) to sort as it started with (n). Thus, by taking a bit of extra time up front to increase the probability of getting an element closer to the actual median of the array, Quick Sort has a better chance of achieving a run time closer to its best case. This trade-off is worth it when the array being sorted is very large. An optimized implementation may only choose to use the median of 3 or 5 strategy if it's sorting more than some number of elements.

Grading: A response does NOT need to be nearly as verbose as the one above. Give them full credit simply if they mention that the chance of getting something closer to the median of the array goes up by doing the median of 3 or 5 strategy. Decide partial credit as you see fit. You may want to read a few papers to get a gauge of the common answers students give before figuring out how much partial credit to give for certain types of responses.

(b) (4 pts) The best case run time of an insertion sort of n elements is O(n) and the worst case run time of an insertion sort is $O(n^2)$. Describe how to (a) construct a list of n distinct integers that, when sorted by insertion sort, gets sorted in the best case run time, and (b) construct a list of n distinct integers that, when sorted by insertion sort, gets sorted in the worst case run time.

We simply keep the list of integers in sorted order to achieve a best case run time. As insertion sort attempts to insert each new item, it will see that it's immediately in place and the inner loop won't ever run, allowing for the best case run time.

To achieve the worst case run time, just put each item in reverse sorted order. When insertion sort attempts to insert each new item, it will be forced to swap the new item with every single other previously inserted item, creating the worst case run-time.

(**Grading:** No need for explanation, 2 pts per case and all that needs to be mentioned is that the list needs to be sorted and reverse sorted, respectively.)

3) (5 pts) ALG (Bitwise Operators)

What is the output of the following C program?

```
#include <stdio.h>

int main() {

    int x = 13, y = 27, z = 74;
    printf("x^y = %d\n", x^y);
    printf("x&z = %d\n", x&z);
    printf("x&(y|z) = %d\n", x&(y|z));
    printf("x|y|z = %d\n", x|y|z);

    int i, sum = 0;
    for (i=0; i<10; i++) {
        if ((x & (1<<i)) != 0) sum++;
        if ((y & (1<<i)) != 0) sum++;
        if ((z & (1<<i)) != 0) sum++;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

x^y = **22**

x&z = **8**

x&(y|z) = **9**

x|y|z = **95**

sum = **10**

(**Grading:** 1 pt each, all or nothing)