UNIVERSITY OF CENTRAL FLORIDA (UCF)

Department of Computer Science

COP 3502 Computer Science I

Common Midterm Examination

Total points: 100 Total Time: 2 hours

COP 3502 Sections *(Circle your section)*	**Instructor**
1	Arup Guha
2	Awrad Mohammed Ali
3	Tanvir Ahmed
4	Mahfuzur Rahaman

Last Name in Upper Case Letter: _____

First Name in Upper Case Letter: _____

UCF ID: _____

Did you circle your section above? _____

Instructions:

1. This is a **closed-book** and **closed-neighbor** exam. No notes, textbooks, or outside assistance are permitted.
2. **Calculators, smartwatches, and electronic devices** (phones, tablets, laptops, earbuds, etc.) are strictly prohibited.
3. **No scratch paper** is allowed. However, you may use the margins or back side of the exam paper for rough work.
4. Write **clearly and legibly**. If the instructor cannot read your handwriting, the answer may not receive credit.
5. Manage your time wisely

Part A - Dynamic Memory Allocation [Total: (2 + 2) + (12 + 4) = 20 pts] Solution

1. a) Write a single line of code that allocates an array of type int storing 500 values, each set to 0.
Answer:

```
int* array = calloc(500, sizeof(int));
```

Grading: 1 pt LHS, 1 pt RHS, has to be completely correct to get 2 pts.

b) Assume *scores* is a pointer to a dynamically allocated array of int with size N. The following line of code attempts to double the size of the array:

```
scores = realloc(scores, N * 2 * sizeof(int));
```

However, if realloc fails, this code will cause a memory leak. Rewrite the code using a few additional lines so that: i) If *realloc* fails, the original memory is properly freed, and *NULL* will be assigned to scores and print a message "failed". ii)If it succeeds, scores points to the new, resized array and prints "success"

Answer:

```
int* tmp = realloc(scores, N*2*sizeof(int));

if (tmp == NULL) {
    printf("failed\n");
    free(scores);
    scores = NULL;
}
else {
    scores = tmp;
    printf("success\n");
}
```

Grading: 1 pt realloc in other var, 1 pt rest

2. This problem relies on the following struct definition. A book can have multiple authors.

```
typedef struct Book {
    char title[50]; // Book's title
    char **authors; // For author names. There can be multiple authors
    float price; // price
    int nAuth; //number of authors
 } Book;
```

a) Write a function that takes the title of a book, an array of author names, and the length n of the authNames array. The function dynamically allocates **ONE Book** and stores the data passed to the function into the dynamically allocated book. During this process, you may need to perform multiple dynamic memory allocations. Finally, the function returns the dynamically allocated book. The function must allocate the exact space needed to store the strings when possible.

```
Book* createBook(char *title, char authNames[][50], float price, int n){

    Book* res = malloc(sizeof(Book));                          // 2 pts
    strcpy( res->title, title);                                // 1 pts
    res->price = price;                                        // 1 pt
    res->authors = calloc(n, sizeof(char*));                   // 2 pts
    res->nAuth = n;                                            // 1 pt

    for (int i=0; i<n; i++) {                                  // 1 pt
        int sz = strlen(authNames[i]);
        res->authors[i] = malloc((sz+1)*sizeof(char));         // 2 pts
        strcpy(res->authors[i], authNames[i]);                 // 1 pt
    }

    return res;                                                // 1 pt
}
```

b) Complete the function that takes a pointer to the book created by the createBook() function and frees memory without creating any memory leak.

```
void freeBook(Book *book) {
     for(int i=0; i< book->nAuth; i++) {

         free(book->authors[i];                    // 2 pts
     }

     free(book->authors);                          // 1 pt
     free(book);                                   // 1 pt
}
```

Part B - Recursion [Total: (3 + 5 + 12 = 20 pts)] Solution

1. Consider the following recursive function:

```
int f(int n) {
    if (n <=1)
       return 1;
    if (n == 2)
        return 3;
    return 2 * f(n - 1) + f(n - 2) - 1;
}
```

What value will be returned by **f(5)?** 33

Grading: 3 pts correct answer, 2 pts if one arithmetic error or if answer they give is 14 or 79 or within 3 of the correct answer, Give 0 pts otherwise

2. Consider the following node struct for a singly linked list.
   typedef struct node { int data; struct node* next; } node;
   Write a recursive function **byeList(node *head)**, that receives the head of a singly linked list and frees all the nodes of the linked list without creating any segmentation fault or memory leak.

```
void byeList(node *head) {
    if (head == NULL) return;     // 2 pts
    byeList(head->next);          // 2 pts
    free(head);                   // 1 pt
}
```

3. Complete the code on the next page so that it prints out all permutations of the integers in between 0 and 9 where there is at least one pair of adjacent values that have a difference of +1 or -1. For example, during the execution of the code, the permutation [0, **3, 2**, 9, 7, 5, 1, 8, 4, 6] should be printed because 3 and 2 are adjacent in the permutation and have a difference of 3 – 2 = 1. (If the 2 and 3 are swapped, that different permutation should also be printed since 2 – 3 = -1.) (Note: This code will take long to run! So, don't try running this at home).

Please fill in the body of the code in the **makePerm()** function and the entire **check()** function.

```
#include <stdlib.h> //in case you want to use abs function for absolute value
#define SIZE 10

void makePerm(int* perm, int* used, int k, int n);
void print(int* perm, int n);
int check(int* perm, int n);

int main() {
    int perm[SIZE];
    int used[SIZE];
    for (int i=0; i<SIZE; i++)
        used[i] = 0;
    makePerm(perm, used, 0, SIZE);
    return 0;
}
void makePerm(int* perm, int* used, int k, int n) {
    if (k == n) {
        if (check(perm,n))
            print(perm, n);
        return;
    }
/*** FILL IN FOR LOOP ***/
    for (int i=0; i<n; i++) {

         if (used[i]) continue;                 // 2 pts
         used[i] = 1;                           // 1 pt
         perm[k] = i;                           // 1 pts
         makePerm(perm, used, k+1, n);          // 1 pt
         used[i] = 0;                           // 1 pt
    }
}

void print(int* perm, int n) {
    for (int i=0; i<n; i++)
        printf("%d ", perm[i]);
    printf("\n");
}
// This function should be written iteratively.
int check(int* perm, int n) {

     for (int i=0; i<n-1; i++)                  // 1 pt
         if (abs(perm[i]-perm[i+1])==1)         // 3 pts
             return 1;                          // 1 pt

     return 0;                                  // 1 pt


}
```

Part C – Linked Lists [Total: 12 + 6 = 18 pts] Solution

1. Consider a typical linked list node struct

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

Write a function that takes the head of a linked list, an integer x, and a position p (p>1). The function inserts x into the position p in the linked list. If the position p is more than the total number of nodes, the function inserts the item at the end of the linked list. Also, note that the first item in the linked list is at position 1, and you can assume that the list has at least one item. **Finally,** the function should return the total number of nodes (including the new one) in the linked list.

Example: if there is a linked list: 10 ->20 ->30-> 7 -> 4-> 8, x = 9, and p =3, the function should result in the following linked list: 10 ->20 -> **9** ->30-> 7 -> 4-> 8. The function should also return 7.

```
int insertAt(node* head, int x, int p) { //Write this function

    node* prev = head;

    for (int i=0; i<p-2 && prev->next != NULL; i++)
        prev=prev->next;

    node* tmp = malloc(sizeof(node));

    tmp->data = x;
    tmp->next = prev->next;
    prev->next = tmp;

    int res = 0;
    while (head != NULL) {
        res++;
        head = head->next;
    }
    return res;
}
```

Grading: Iterating to the point of insertion without going too far – 4 pts
Allocating Node and filling it – 2 pts
Patching Node in – 2 pts
Counting Nodes – 3 pts
Returning Count – 1 pt

2. Consider passing a pointer to a linked list storing the values $ptr \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 1 \rightarrow 8 \rightarrow 4$ to the function shown below. After the function is completed, show the contents of the list in order. Please list the items as shown above, with an arrow representing the next node for each struct.

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

void whatDoesItDo(node* front) {
    while (front != NULL) {
        if (!(front->data%2 == 1)) {
            node* tmp = malloc(sizeof(node));
            tmp->data = front->data + 5;
            tmp->next = front->next;
            front->next = tmp;
            front = tmp;
        }
        front = front->next;
    }
}
```

Answer:

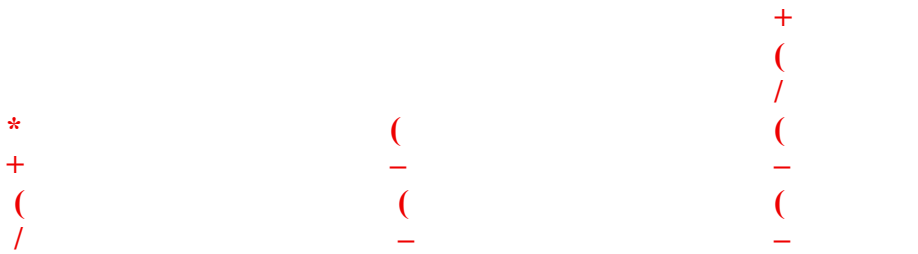
ptr → 3 → 2 → 7 → 6 → 11 → 1 → 8 → 13 → 4 → 9

Grading: 2 pts for keeping all original values in their same relative orderr
1 pt each for each of the 4 insertions being the correct number

Part D – Stacks and Queues [Total: 8 + 4 + 3 + 5= 20 pts] Solution

1. Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (A, B, and C) in the infix expression. It means when you reach point A, show the content of the stack in the first stack out of the following 3 stacks. Similarly, do this for points B and C.

8 / (4 + 2 * 2 **A**) - (9 - (8 / (2 + 2 **B**))) + 5 **C**



Stack Status at A **Stack Status at B** **Stack Status at C**

Final postfix (Note that you may not need to fill out all the boxes):

8	4	2	2	*	+	/	9	8	2	2	+	/	-	-	5	+			

Grading: 1 pt for each stack, 5 pts for the expression, give partial on the expression as you see fit (generally speaking take off 1 pt per error cap at 5)

2. The following function performs deQueue operation on a linked list-based implementation of a queue.

```
int deQueue(struct queue* qPtr) {
    if(isEmpty(qPtr))
        return -1;
    int saveVal = qPtr->front->data;
    struct node* temp = qPtr->front;
    qPtr->front = qPtr->front->next;
    free(temp);
    if(isEmpty(qPtr)) {
        qPtr->back = NULL;
    }
    return saveVal;
}
```

The above function calls the **isEmpty()** function twice. In what **initial** state of the queue will the second isEmpty() call return true while performing a dequeue operation, and why do we need to check this condition again? **Answer in one to three sentences:**

If the queue pointed to by qPtr has 1 element in it when deQueue is called, the second isEmpty call will return 1 (true). We need to check it because both front and back need to be set to NULL in this special case.

Grading: 2 pts for 1 element answer, 2 pts for why to recheck

3. Explain why a queue implemented via a traditional linked list storing only a pointer to the front of the queue is inefficient? Which operation(s) are less efficient than they could be?

Enqueue takes O(n) time for a queue storing n items because with no direct access to the back of the queue, one has to traverse through the entire list of n elements to find the last node, where the next pointer has to be changed.

Grading: 1 pt enqueue, 2 pts explanation why it's slow.

4. Consider the following struct for an array-based circular queue implementation. Answer the following

```
struct queue {
   int* elements;
   int front;
   int noe;
   int queueSize;
};
```

noe= total items in the queue.

queuesize= *total capacity of the elements array.*

Complete the dequeue function. If the queue is empty, return -1. Otherwise, write the necessary code to complete the regular dequeue operation and adjust the queue properties.

```
int dequeue(struct queue* qPtr){

    if (qPtr->noe == 0) return -1;

    int retval = qPtr->elements[qPtr->front];

    qPtr->front = (qPtr->front + 1)%qPtr->queueSize;

    qPtr->noe--;

    return retval;

}
```

Grading: 1 pt return -1 when empty
1 pt storing retval
1 pt updating front correctly
1 pt updating number of elements
1 pt returning

Just take off 1 pt total if qPtr-> missing 2 or less times
Take off 2 pts total if qPtr-> missing a lot

Part E – Algorithm Analysis [Total: 4 + 8 + 9 + 1 = 22 pts] Solution

1. A brute force algorithm has a run time of $O(4^n)$. For an input size n = 10, the algorithm takes 100 milliseconds. How long would it be expected for the algorithm to take to complete running on an input size of 12? **Please give your answer in seconds. (1 second = 1000 milliseconds)**

Let T(n) be the run time of the algorithm for an input of size n. Using the given information, we have $T(n) = c(4^n)$, for some constant c. Plug in information about n = 10:

$T(10) = c(4^{10}) = 100$ ms $\rightarrow c = \frac{100\ ms}{4^{10}}$. Now, we must solve for T(12):

$$T(12) = c(4^{12}) = \frac{100\ ms}{4^{10}} \times 4^{12} = (100\ ms) \times 4^2 = 1600ms = 1.6\ sec$$

Grading: 2 pts solve for c, 1 pt get to 1600 ms, 1 to convert to 1.6 seconds

2. What is the **BigO** run-time for the following functions? Just write the BigO. You don't have to explain them.

```
int f1(int A[],int B[],int n) {
  int i, j, sum = 0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
       if (A[i] == B[j])
          sum++;
  return sum;
}
```

Run-time: $O(n^2)$

```
int f3(int m) {
  int i,j;
  for (i=1; i<=m; i++) {
    for (j=1; j<=m; j++)
        if (j == 2)
            break;
   }
   return j;
}
```

Run-time: O(m)

```
void analyzeBigOh(int n) {
    int sum = 0;
    int i = 1;
    while (i < n) {
        sum += i;
        i *= 2;
    }
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            sum += j + k;
        }
    }
    printf("Sum is: %d\n", sum);
}
```

Run-time: $O(n^2)$

```
Assume a binary search function
binSearch() is already implemented.
//in the parameter, n is the length of array
b[], the array on which the search occurs.
void match(int a[], int m, int b[], int n)
{
   for (int i = 0; i < m; i++) {

        if(a[i] == -1) break;
        //searching for a[i] in b array
        if(binSearch(a[i], b, n) ==1)
           printf("%d ", a[i]) ;
    }
}
```

Run-time: O(mlgn)

Grading: 2 pts for each one, no partial credit on any part, each part is right or wrong.

3. **Multiple choice (Circle the correct answer) /fill in the blanks:**

a) In binary search, if searchKey < Array[mid], which operation would you perform?

i) high = mid+1 ii) return Array[mid] iii) low = mid + 1 iv) low = mid – 1 **v) high = mid-1**

b) What would be the Big-O notation for $f(n) = 4n^2 + 500\frac{n}{2} * \log n$

i) $O(n^2)$ (ii) $O(n^3 + \frac{n}{2} * \log n)$ (iii) O(n log n) (iv) $O(\frac{n}{2} * \log n)$ (v) O(log n)

What is the Big-O run-time for the following operations?

c) Getting the largest number from a sorted array of size **n**: i)Best case: **O(1)** ii) Worst case:**O(1)**

d) Printing the elements of a stack with **top** number of elements:

i)Best case:**O(top)** ii) Worst case:**O(top)**

e) Linear search in an array of size **M**: i)Best case: **O(1)** ii) Worst case: **O(M)**

f) Inserting an item to the end of a linked list of **n** nodes without a tail pointer: i)Best case: **O(n)** ii) Worst case: **O(n)**

g) Push operation in a stack with **n** items (No reallocation occurs if the stack becomes full):

i)Best case:**O(1)** ii) Worst case: **O(1)**

h) Worst case runtime of calculating factorial of **n**: **O(n)**

i) Number of disk movement in tower of Hanoi in **n** disks: **$2^n - 1$**

Grading: 1 pt for each part, have to get the whole part to get the point.

4. **For Fun** The UCF Football team is about to start an away game against the Cincinnati Bearcats. In what city is the team currently?

Cincinnati (Grading: Give to All)