COP 3502 Quiz #2 Version D (Recursion, Linked Lists, Stacks, Queues) Solutions

1) (5 pts) Write a **recursive function** that takes in an integer, **start**, an integer **diff**, and an integer **n**, and returns the sum of the values in an arithmetic series starting with the value **start** with a common difference of **diff**, with **n** terms. An arithmetic series is one where the difference between successive values is the same. For example, an arithmetic series starting at start = 7 with a common difference of diff = 4 with n = 5 terms is 7, 11, 15, 19, and 23. This series has a sum of 75. (Thus, the function call sumSeries(7, 4, 5) should return 75.)

```
int sumSeries(int start, int diff, int n) {
    if (n == 0) return 0;
    return start + sumSeries(start+diff, diff, n-1);
}
```

Grading: Base case = 2 pts, can use either n = 1 or n = 0.
Recursive case: Can split off first or last term… 1 pt for term, 2 pts for rec call

2) (8 pts) 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 and provide the corresponding postfix expression as well.

6 * (1 + 18 / (1 [A] + 30 / 6) [B] - 1) + 7 - (8 * (3 [C] - 1))

A	B	C
(		
/	/	(
+	+	*
(	(	(
*	*	-

Resulting postfix expression:

6	1	18	1	30	6	/	+	/	+	1	-	*	7	+	8	3	1	+	*	-

Grading: 1 pt for each stack, 5 pts for the expression, take off 1 pt for each error in the expression (a single error is usually an item out of place, so to fix it, you insert an item in the different place. The fewest # of insertions needed to fix the expression is the number of points to take off, capped at 5)

3) (10 pts) A frog is on a 2D grid and has a rule that he will always jump strictly up in elevation, never down. Consider marking all of the places the frog could go via floodfill from a particular location. For example, if the grid was as follows with a starting location of row 2, column 3 (0-based), where each value represents the elevation of that square in the grid, the highlighted squares represent all of the places the frog could reach. Row 0 is the northern most row of the grid, Column 0 is the western most column of the grid.

12	14	3	8	3
14	13	12	10	4
15	12	14	5	6
12	18	16	3	8
8	6	9	12	9

Complete the code below so that the used array gets appropriately filled with 1's in the locations the frog can reach. (Note: when making recursive calls, you don't have to see if you've been to a square before; mutual recursion is impossible due to the nature of the problem!!!) Note: on a single jump the frog can move north, south, east or west in the grid only. Also, assume the grid size is SIZE by SIZE.

```
#define SIZE 5
#define NUMDIR 4
const int DX[NUMDIR] = {-1,0,0,1};
const int DY[NUMDIR] = {0,-1,1,0};

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

void fill(int grid[][SIZE], int used[][SIZE], int x, int y) {

    used[x][y] = 1;

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

        int nX = x + DX[i];

        int nY = y + DY[i];

        if ( !inbounds(nX, nY) ) continue;

        if ( grid[nX][nY] <= grid[x][y] ) continue;

        fill(grid, used, nX, nY);
    }

}
```

Grading: 2 pts for each slot, give 1 pt if close but there's a mistake

4) (12 pts) Write a function that takes in a pointer to a linked list and deletes every other node in the list, starting with the second node. The function is void since the front of the list will never change. Use the struct given to you below and don't forget to free the memory for each of the deleted nodes.

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

void delEveryOther(node* head) {


    node* cur = head;

    while (cur != NULL && cur->next != NULL) {
        node* forward = cur->next->next;
        free(cur->next);
        cur->next = forward;
        cur = forward;
    }


}
```

Grading: Many ways to do this. Here are the key issues and points for each:

Handling lists of size 0, 1: 2 pts (sometimes separate code, sometimes now)

Reasonable looping structure: 2 pts

Storing next node to connect to in loop: 2 pts

Freeing the appropriate node: 2 pts

Connecting previous node to next node: 2 pts

Updating where you are in the list: 2 pts