COP 3502 Recitation Sheet: Review Questions for Quiz #2
(taken from Past Exams)

1) Consider the following recursive function:

```
int compute(int array[],int low, int high) {

    if (low == high) return array[low]%3 + 1;

    int mid = (low+high)/2;
    int left = compute(array, low, mid);
    int right = compute(array, mid+1, high);
    return left*right;
}
```

Consider the function call `compute(array, 0, 6)` where array is shown below:

index	0	1	2	3	4	5	6
array	17	4	19	30	47	999	13

Determine the result of this recursive call, as well as each other recursive call that gets made as a result of this original one and its return value. Please fill in the recursive calls in the order that they *start*. (Note: This order is different than the order in which they finish and a significant hint has been given to you below.)

Recursive Call	Return Value
`compute(array, 0, 6)`	
`compute(array, 0, ___ )`	
`compute(array, 0, ___ )`	
`compute(array, 0, ___ )`	
`compute(array, 1, ___ )`	
`compute(array, 2, ___ )`	
`compute(array, 2, ___ )`	
`compute(array, 3, ___ )`	
`compute(array, 4, ___ )`	
`compute(array, 4, ___ )`	
`compute(array, 4, ___ )`	
`compute(array, 5, ___ )`	
`compute(array, 6, ___ )`	

2) Complete the program below so that it prints out all the permutations of 0,1,2, ...,SIZE-1 such that the absolute value of the difference between each pair of adjacent numbers in the permutations is 2 or greater. For example, when SIZE = 4, the code would print out:

```
1 3 0 2
2 0 3 1
```

the only 2 permutations such that the absolute value of the difference between each pair of adjacent terms is 2 or greater.

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

#define SIZE 4

void printPerms(int perm[], int used[], int k, int n);
void print(int perm[], int n) ;

int main() {
    int perm[SIZE], used[SIZE], i;
    for (i=0; i<SIZE; i++) used[i] = 0;
    printPerms(perm, used, 0, SIZE);
    return 0;
}

void printPerms(int perm[], int used[], int k, int n) {

    if ( ___________ ) print(perm, n);

    int i;
    for (i=0; i<n; i++) {
        if (!used[i]) {
            if ( _________ || ___________________________________ ) {

                used[i] = ______ ;

                perm[k] = ______ ;

                printPerms(______ , ______ , ______ , ______ );

                used[i] = ______ ;
            }
        }
    }
}
```

3) Write a ***recursive*** function, `equal`, that takes in pointers to two linked lists and returns 1 if the two lists are equal and 0 otherwise. For two lists to be equal, they have to have the same number of elements, and each corresponding element must be equal. (For example, the lists 3, 9, 5 and 3, 9, 5 are equal, but the lists 3, 4, 7 and 3, 7, 4 are not equal and the lists 2, 9, 1 and 2, 9, 1, 4 are not equal.)

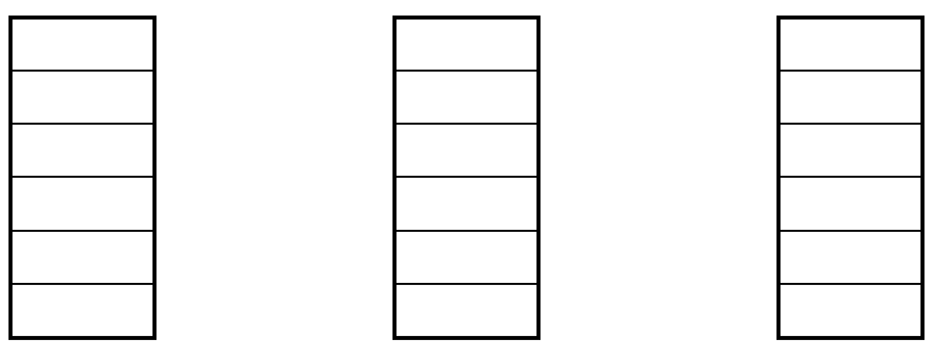
```
typedef struct node {
    int data;
    struct node* next;
} node;

int equal(node* listA, node* listB) {

}
```

4) Utilize an operand stack to evaluate the following postfix expression. Show the contents of the stack at each of the indicated points:

```
                          1                 2                    3
3   6   10   2   3   +      /   +   *   4    8   2   /   1   +      *   -
```



1 2 3

Final Expression Value: ___________

5) An incomplete version of a linked list implementation of a stack is below. Fill in the code for each function that is empty.

```
struct stack {
  int data;
  struct stack *next;
};

void init(struct stack *front) {
  front = NULL;
}

// Returns a pointer to the new front of the stack. Assume that allocating
// space for the new node is always successful.
struct stack* push(struct stack *front, int num) {









}

// Returns the new front of the stack and sets the variable pointed to
// by ptrTop to the value popped from the top of the stack. If no such
// value exists, NULL is returned and the the variable ptrTop is pointing
// to is set to 0.
struct stack* pop(struct stack *front, int* ptrTop) {

  struct stack *temp;
  *ptrTop = 0;
  if (front != NULL) {
    temp = front;
    front = front->next;
    *ptrTop = temp->data;
    free(temp);
    return front;
  }
  else
    return NULL;

}

// Returns 1 if the stack is empty, 0 otherwise.
int empty(struct stack *front) {







}
```