UCSC Silicon Valley Extension

C Programming, Advanced

Assignment 1: Recursion and time complexity
Instructor : Radhika Grover

Draw the call stack to show how the following recursive functions are executed. For each callee show the values of the local variables, arguments and the statement where it returns to the caller. Provide the source code and output for all programming problems and test your programs on the test cases provided in the folder *Test cases for Assignment 1. The sample output is also provided for the programming problems.*

1. Draw the call stack for the merge sort algorithm when it is executed on an array with the following elements: 10, 3, 7. Show how the array is updated by each function. The algorithm is invoked as follows:

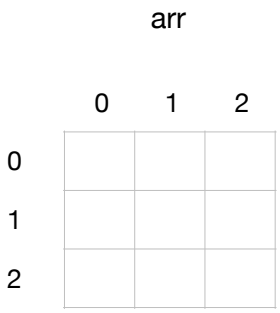
```
mergeSort(array, 0, 2);
```

The pseudocode is provided below:

```
#include <stdio.h>

void merge(int *arr, int p, int q, int r) ;
void mergesort(int *arr, int p, int r);

void mergesort(int *arr, int p, int r) {
        if (p < r) {
                int q = (p + r)/2;
                mergesort(arr, p, q);
                mergesort(arr, q+1, r);
                merge(arr, p, q, r);
        }
}

void merge(int *arr, int p, int q, int r) {
        int p1 = p; // index to first array
        int q1 = q+1; // index to second array
        int l1 = 0, i;
        int length = r - p + 1;
        int temp[length]; // temporary array
        int numMerged = 0;

        // place the smaller of the two elements in arrays with indices p1 and p1
        // into temp
        while (numMerged <= length && p1 <= q && q1 <= r) {
```

```
                if (arr[p1] < arr[q1]) {
                        temp[l1++] = arr[p1];
                        p1++;
                } else if (arr[q1] < arr[p1]) {
                        temp[l1++] = arr[q1];
                        q1++;
                } else if (arr[q1] == arr[p1]) {
                        temp[l1++] = arr[q1];
                        temp[l1++] = arr[p1];
                        q1++;
                        p1++;
                        numMerged++;
                }
                numMerged++;
        }

        // all the elements of the first array have been merged into temp
        if (p1 > q) {
                // copy all the elements from arr[q1] to arr[r] into temp1

                for (i = q1; i <= r; i++) {
                        temp[l1++] = arr[i];
                }

        }
        // all the elements of the second array have been merged into temp
        if (q1 > r) {
                // copy all the elements from = arr[p1] to arr[q] into temp1
                for (i = p1; i <= q; i++) {
                        temp[l1++] = arr[i];
                }

        }

        // copy all the elements from temp into arr
        for (int i = 0; i < length; i++)
                arr[p+i] = temp[i];

}

int main(void) {
        int arr[] = { 10, 300, 15, 10, 10000, 17, 8, 10, 50, 100};

        mergesort(arr, 0, 9);

        for (int i = 0; i < 10; i++)
                printf("%d \n", arr[i]);

        return 0;
}
```

2. The function *move* recursively visits each square to the north, south, east, and west of a given square (*r*, *c*) in the following grid, where *r* is the row number and *c* is the column number:




```
move(int arr[3][3], int r, int c){
      arr[r][c] = 1;
      move (arr, r, c+1);
      move (arr, r, c-1);
      move (arr, r+1, c);
      move (arr, r-1, c);
  }


  int main(){
      initialize arr to 0;
      move(arr, 0, 0);
      print out contents of arr;
  }
```

a. Draw the call stack to show how the functions are called, and the local variables and arguments that are stored in the stack frame for each function call.
b. The program terminates with a segmentation fault - identify the causes by examining the call stack in (a).
c. Correct the program so that all squares in the grid are visited and the program terminates normally.

3. Prove or disprove each of the following statements:
(a) $1/2\ n^2 - 3n = \Theta(n^2)$
(b) $n^3 \mathrel{!}= O(n^2)$
(c) $n \log n - 2n + 13 = \Omega(n \log n)$
(d) $n^{1/2} = \Theta(n^{2/3})$
(e) $n! = \omega(2^n)$

Time Complexity Analysis

4. Find the recurrence relation for the following algorithm. Solve the recurrence relation to find the running time of the algorithm.

```
// assume n is even
    void arraySum(int * A, int n){
    if(n == 1) return;
    for(int i=1; i <= n/2; i++){
                A[i] = A[i] + A[n/2];
        }
             arraySum(A, n/2);
    }
```