

## CSC 112 Homework 2 - Recursion & Search

Name \_\_\_\_\_

Due Date: Start of class, 10/14/2009

1. Define a tail-recursive function equivalent to the *count* function below. This function counts the number of times an element exists in an array. Feel free to turn this into a helper function or add parameters as necessary.

```
int count(int* data, int x, int arraySize)
{
    if (arraySize == 0) return 0;
    else
    {
        int countInRest = count(data+1, x, arraySize-1);
        if (x == *data) return 1 + countInRest;
        else return countInRest;
    }
}
```

Answer:

```
int count2(int* data, int x, int arraySize, int countSoFar)
{
    if (arraySize == 0) return countSoFar;
    else
    {
        if (x == *data)
            countSoFar = countSoFar + 1;
    }
    return count2(data + 1, x, arraySize-1, countSoFar);
}

int count2(int* data, int x, int arraySize)
{
    return count2(data, x, arraySize, 0);
}
```

2. Complete the function below. Your answer should be a **tail-recursive** function that finds the minimum of an array using the following algorithm: on any recursive call, compare the minimum value seen so far (*minSoFar*) against the first entry of the array as passed into the current recursive call. Whichever value is smaller becomes the new minimum value seen so far, and the recursion continues with this *minSoFar* and the part of the array not yet seen passed in as parameters to the next recursive call.

Here's an example setup:

Array: [ 2 | 4 | 1 | 5 | 0 ]

Initially, 2 is the minimum so far, and a test is made looking at 2 against the rest of the array [ 4 | 1 | 5 | 0 ]

2 is compared against 4 (the first entry of [ 4 | 1 | 5 | 0 ]).

2 is smaller than 4, so 2 wins and remains the minSoFar.

Now a test against the rest of the array needs to be executed – 2 against [ 1 | 5 | 0 ]

2 will be compared against 1 ....

```
int minimumHelper(int* data, int arraySize, int minSoFar)
{
    if (arraySize == 0) return minSoFar;
    else
    {
        if (minSoFar > *data) minSoFar = *data;
        return minimumHelper(data+1, arraySize-1,
                               minSoFar);
    }
}

int minimum(int* data, int arraySize)
{
    return minimumHelper(data+1, arraySize-1, *data);
}
```

3. Convert the following tail recursive *logicalOR* function into a while loop. You should be heavily influenced by the nature of the tail-recursion in coming up with your while loop and indicate how the recursive function led you to the particular while loop implementation you used.

```
bool logicalOR(bool* array, int size)
{
    if (size == 1) return *array; // line 1
    else if (*array == true) return true; // line 2
    else return logicalOR(array+1, size-1); // line 3
}
```

```
bool logicalOR(bool* array, int size)
{
    bool answer = false;
    bool *location = array;

    // the loop runs until the base cases are hit
    while ((size > 1) && (*location == false))
    {
        // if looping, update array position by 1
        // subtract 1 from size

        location = location + 1;
        size = size - 1;
    }
    // if hit base cases, return what's at current
    // point in array
    // either value of last spot, or a true (which was one
    // of the base cases and which got you out of loop
    return *location;
}
```

4. Below you will find a sorted array of 5 integers. You need to search for each of the 5 items in the array. With any search algorithm, finding each of the five items will take varying lengths of time depending on the placement of the items in the array and the search technique used. Compute the **average length of time to search** (the sum, over all items, of the number of comparisons required to find each item, divided by five) if you use our standard **front-to-back search** and then **if you use our binary search algorithm**. Code for each search algorithm is on the next page. You should trace the search process using each algorithm to come up with the average times.

data = [ 0 | 7 | 21 | 42 | 95]  
Search for: 0, 7, 21, 42, 95

Front-to-back search requires 1 comparison to find 0, 2 to find 7, 3 to find 21, 4 to find 42, and 5 to find 95. This is a total of  $(1 + 2 + 3 + 4 + 5)$  which is 15, so the average is 3.

binarySearch(data, 0, 4, 0) → compare to position 2, value 21 (less than)  
binarySearch(data, 0, 1, 0) → compare to position 0, value 0 (equal) → 2 steps

binarySearch(data, 0, 4, 7) → compare to position 2, value 21 (less than)  
binarySearch(data, 0, 1, 7) → compare to position 0, value 0 (greater than)  
binarySearch(data, 1, 1, 7) → compare to position 1, value 7 (equal) → 3 steps

binarySearch(data, 0, 4, 21) → compare to position 2, value 21 (equal) → 1 step

binarySearch(data, 0, 4, 42) → compare to position 2, value 21 (greater than)  
binarySearch(data, 3, 4, 42) → compare to position 3, value 42 (equal) → 2 steps

binarySearch(data, 0, 4, 95) → compare to position 2, value 21 (greater than)  
binarySearch(data, 3, 4, 95) → compare to position 3, value 42 (greater than)  
binarySearch(data, 4, 4, 95) → compare to position 4, value 95 (equal) → 3 steps

$(2 + 3 + 1 + 2 + 3)$  which is 11 steps total, or an average of **2.2** over the 5 searches.

Front-to-back-search code:

```
int findInArray(int* data, int size, int position, int
valueToSearchFor)
{
    if (position == size) return -1;
    else if (valueToSearchFor == data[position]) return
position;
    else return findInArray(data, size, position+1,
valueToSearchFor);
}
```

```
// this function, called by the user, just passes the work
// of to the other findInArray function
int findInArray(int* data, int size, int valueToSearchFor)
{
    return findInArray(data, size, 0, valueToSearchFor);
}
```

Binary search code:

```
int binarySearch(int* array, int first, int last, int
valueToSearchFor)
{
    if (first > last)
        return -1;
    else
    {
        int middle = (first + last) / 2;
        if (valueToSearchFor == array[middle])
            return middle;
        else if (valueToSearchFor < array[middle])
            return binarySearch(array, first, middle-1,
valueToSearchFor);
        else return binarySearch(array, middle+1, last,
valueToSearchFor);
    }
}
```

```
// this function, called by the user, just passes the work
// of to the other binarySearch function
int binarySearch(int* array, int size, int valueToSearchFor)
{
    return binarySearch(array, 0, size-1, valueToSearchFor);
}
```