

CSC 112 – Test 2

Name: _____

Show all your work and reasoning for your answer in the space provided. Be certain you clearly **indicate** your final answer. This exam is to be done individually according to the honor code at Wake Forest University. *You are allowed to leave off one of either 1c or 4b (marked with an *), but must do one or the other.*

Problem 1. Here is an implementation of a recursive algorithm that can find the k^{th} smallest element of an array of size n (if $k == 1$, returning the minimum; if $k == 2$, returning the 2nd smallest, ..., if $k == n$, returning the maximum). You can assume $n \geq 1$.

```
int minimumIndex(int* array, int first, int last)
{
    int j;
    int minIndex = first;
    for (j = first + 1; j <= last; j = j + 1)
    {
        if (array[j] < array[minIndex]) minIndex = j;
    }
    return minIndex;
}
```

```
void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}
```

```
int kthSmallestHelper (int* array, int first, int last, int k)
{
    int minIndex = minimumIndex(array, first, last);
    swap(array, first, minIndex);
    if (first == (k-1))
    {
        return array[first];
    }
    else
    {
        return kthSmallestHelper (array, first+1, last, k);
    }
}
```

```
int kthSmallest(int* array, int size, int k)
{ return kthSmallestHelper(array, 0, size-1, k); }
```

Note that the *minimumIndex* and *swap* functions are the same as previously written for *selection sort*.

a) Trace the algorithm on the following call, with your trace being *the current ordering of the data in the array* and the values for the parameters (*first, last, and k*) when each call to *kthSmallestHelper* is made, as well as *the final ordering of the data in the array* when the function returns:

```
int* array = {9, 2, 1, 5, 4, 3};  
cout << kthSmallest(array, 6, 3) << endl;
```

```
kthSmallestHelper([9|2|1|5|4|3], 0, 5, 3); // find min and swap, check if at right  
point, if not make call below  
kthSmallestHelper([1|2|9|5|4|3], 1, 5, 3); // find min and swap, check if at right  
point, if not make call below  
kthSmallestHelper([1|2|9|5|4|3], 2, 5, 3); // find min and swap, check if at right  
point, we are
```

final array: [1|2|3|5|4|9] return array[first] which at this point is array[2]

b) What are the worst case inputs for the algorithm as *written*, and what is the computational cost, in terms of number of comparisons, of that worst case?

Worst case inputs: Any ordering of the data, looking for the largest item overall ($k == \text{size}$). This algorithm is essentially using the selection sort process to put items into place. Selection sort's cost is not dependent on the ordering of the array. When we look for the largest item overall, as written, we essentially have to do a full selection sort, which is $O(n^2)$.

*c) Argue, using what you have learned over the past month in class, that there exist algorithms that find the k^{th} smallest element whose worst case performance is always better than the worst case performance of the algorithm I've written. You don't have to write such an algorithm, just argue that there must be a better one that exists.

One way to approach this problem is to say: I can just sort the array and find the k^{th} element in the array. We know there are sorting algorithms that are cheaper than selection sort (merge sort is always $O(n * \log n)$), so if we used one of those, we know in the worst case we can always do better than the written approach.

Problem 2. Here is a recursive function that correctly computes the greatest common divisor of two integers, m and n . Assume that this function, when called the first time, will be provided **two different positive integers**. Here are some examples of GCD at work: $GCD(12,18)$ is 6, $GCD(12,20)$ is 4, $GCD(2,5)$ is 1.

```
int gcd(int m, int n)
{
    if ((m % n) == 0)
        return n;
    else
        return gcd(n, m % n);
}
```

a) Choose **any one of the three** steps used to verify a recursive algorithm is correct and demonstrate that step here (and on the back of this page if needed) for the `gcd` function.

a) Are base cases correct?

Given two different positive integers m and n , the only way to have remainder 0 for m/n is if n evenly divides m and, thus, n is a divisor of m . Since n is a divisor, n is $>$ than any numbers below it, and there can't be any common divisors m and n share greater n (since those wouldn't divide n), then n is the greatest of all divisors of m that n could share with m and is the correct $\text{gcd}(m,n)$.

b) Are we driving to base cases (no infinite recursion)?

Every two integers have, at the minimum, a gcd of 1, and gcd (anything, 1) is guaranteed to equal 0. So, we need to show we will eventually call gcd(anything, 1) in that extreme case. Alternatively, if $m \% n \neq 0$, then the gcd of m and n is guaranteed less than n . Assuming, the gcd of m and n is > 1 , then we'll call need to call gcd(something, actual_gcd). In either case, we need to make sure that the n parameter is shrinking in every recursive call.

We know by definition of $m \% n$, that the 2nd parameter will shrink every time ($m \% n$ has to be $< n$). Thus, we are moving in the right direction (towards 1 or towards the actual_gcd which is $<$ the original n)

c) Is your solution correct, assuming recursive solution correct?

Since we are not making changes to our recursive solution, we need to prove: $\text{gcd}(m,n) == \text{gcd}(n, m\%n)$. We will prove: the set of common divisors of m and n is the same as the set of common divisors of n and $m \bmod n$

Let m, n be integers with $m, n > 0$. Let $x \mid y$ indicate x evenly divides y .

Suppose $d \mid m$ and $d \mid n$. We have $d \mid n$. It remains to show that $d \mid (m \bmod n)$. By our assumption, there are integers k_0, k_1 with $m = k_0 \cdot d$ and $n = k_1 \cdot d$. By the division theorem, we can write $m = (m \operatorname{div} n) \cdot n + (m \bmod n)$. Hence $(m \bmod n) = m - (m \operatorname{div} n) \cdot n = k_0 \cdot d - (m \operatorname{div} n) \cdot k_1 \cdot d = [k_0 - (m \operatorname{div} n) \cdot k_1] \cdot d$. So $d \mid (m \bmod n)$.

Assume separately, $d \mid n$ and $d \mid (m \bmod n)$. We have $d \mid n$. It remains to show $d \mid m$. By our assumption, there are integers k_2, k_3 with $n = k_2 \cdot d$ and $(m \bmod n) = k_3 \cdot d$. By the division theorem, we can write $m = (m \operatorname{div} n) \cdot n + (m \bmod n)$. So $m = (m \operatorname{div} n) \cdot k_2 \cdot d + k_3 \cdot d = [(m \operatorname{div} n) \cdot k_2 + k_3] \cdot d$. So $d \mid m$.

Since the set of common divisors of m and n is the same as the set of common divisors of n and $m \bmod n$, the greatest element of these sets must also be the same. Hence $\gcd(m, n) = \gcd(n, m \bmod n)$.

b) Is this function tail-recursive? Why or why not?

Yes, this function is tail recursive as there is no additional work done to the answer returned from the recursive call. The current functions solution is just the solution to the smaller problem.

c) It is possible to translate this function to a while-loop based function. Remember that every while loop has *initialization*, *test*, and *update* instructions that control the execution of the loop. What would be correct *update* instruction(s) for this algorithm if translated to a while loop?

One needs to make sure m and n are set up appropriately for each time through the loop. We need to give them the values they would get through the recursive call. However, since we are not passing them as parameters, updating either one sequentially is problematic. We need a temp variable to hold one of the values while doing the updates. I decided to let temp hold m .

```
temp = m;  
m = n;  
n = temp % n;
```

Problem 3. Indicate below appropriate base case values for the following problems and explain why.

a) Problem: Computing the remainder of m divided by n using the following recursive approach:

```
int remainder(int m, int n)
{
    // base case(s)
    //...
    // recursion:
    else return remainder(m - n, n);
}
```

Appropriate Base Case(s): **if ($m < n$) return m**

Explanation: **By repeatedly subtracting m from n in the recursion, you are removing any multiples of n that are components of m . Once m is less than n , there are no more multiples of n as part of m . Thus, this remaining value (which is guaranteed $< n$, leading to the base case check) is the integer remainder.**

b) Problem: Computing the *exclusiveOR* over an array of boolean values of size n , under a definition of exclusiveOR for arrays of: at least one value in the array must be true, but they can't all be true.

Appropriate Base Case(s): **if ($position == size$) [hit end of array] return false OR if have seen one true and one false return true**

Explanation: **if you see at least one true and one false, then you know the extremes which lead to an overall false value (*all falses, all trues*) can't be happening – thus you know the answer – true.**

The other base case is if you hit the end of the array (have looked at every item). If you get there, you do know you are in the extremes case (you have either seen all true or all false) and thus your answer must be false.

Problem 4. a) On the last pages of the test, you will find the code for insertion sort. Trace, showing the array after each of the first four “put-in-order” steps of the algorithm, insertion sort on the following array:

```
[ 9 | 7 | 2 | 5 | 10 | 12 | 1 ] // 9 is already in place (get that one for free)
[ 7 | 9 | 2 | 5 | 10 | 12 | 1 ] // push 7 forwards as far as possible, it moves past 9
[ 2 | 7 | 9 | 5 | 10 | 12 | 1 ] // push 2 forwards as far as possible, it moves past 9
and 7
[ 2 | 5 | 7 | 9 | 10 | 12 | 1 ] // push 5 forwards as far as possible, it moves past 7
and 9
[ 2 | 5 | 7 | 9 | 10 | 12 | 1 ] // push 10 forwards as far as possible, it doesn't move
anywhere
```

*b) On the last pages of the test, you will find the code for *insertion sort*, *selection sort*, and *merge sort*. Assume you have been asked to write an algorithm to do what I'll call a “partial sort” – that after execution, the first m sorted values of the original size n array are available and the ordering of the rest is unimportant. Note this isn't “sort the first m values in the current array” but instead “make available the first m sorted values (those that would reside at the front of the array if the data was sorted)”. *Which of the three algorithms we have studied would be most easily converted to this algorithm and why?* Make sure you explain your choice.

You should modify selection sort. Neither insertion sort nor merge sort make any guarantees about the ordering of the data (with regards to the first m sorted values) until those algorithms quit (for example, the smallest value may be put in place **last** when using insertion sort).

Selection sort works by putting the smallest items at the front, and doing this in order – the smallest item is found in the first step and moved to the front, the 2nd smallest item is found in the second step and moved to just behind the front, etc. Thus, on the k th step, the first k values are in the correct overall sorted spot.

The modification to make to selection sort is to just add another parameter m which is used to say when to stop working (after the m th item is put in place).

Problem 5. On the last page of the test, you will find an algorithm called Mystery Sort that has been proposed as a valid sorting algorithm. From reading the algorithm, indicate and argue for whether or not you believe it will correctly sort in ascending order an array of integers. If you do not believe it will sort correctly, your best argument is to show a counter-example where it fails. If you do believe it should sort correctly, you should direct your argument to show that the instructions provided will always put the items in ascending order.

This algorithm does sort correctly. It continually pushes the next largest item to the back, and then updates the back position to be one less in each iteration. The front spot (the smallest item) is put in place for free (it is guaranteed to be the smallest item when we get to that spot).

```
void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}

void mysterySort(int* array, int size)
{
    int j, k;
    for (j = size-1; j > 0; j--) // controls where to put the
        item, starting at last spot and moving forwards to 2nd spot
    {
        for (k = 0; k < j; k++) // for items not yet put in
            place, if item at k is larger than item at k+1, swap them; next
            iteration k equals k+1 so you pick up your larger of the two
            items and keep comparing it to the rest of the array
        {
            if (array[k+1] < array[k])
            {
                swap(array, k, k+1);
            }
        }
    }
}
```

This algorithm is called “bubble-sort”. It’s not looked upon favorably as it is $O(n^2)$ always in terms of number of comparisons and it does more swaps than either insertion or selection sort.

Selection Sort:

```
int minimumIndex(int* array, int first, int last)
{
    int j;
    int minIndex = first;
    for (j = first + 1; j <= last; j = j + 1)
    {
        if (array[j] < array[minIndex]) minIndex = j;
    }
    return minIndex;
}

void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}

void selectionSort(int* array, int size)
{
    int index, k;
    for (k = 0; k < (size-1); k++)
    {
        index = minimumIndex(array, k, size-1);
        swap(array,k,index);
    }
}
```

Insertion Sort:

```
void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}

void insertionSort(int* array, int size)
{
    int k;
    int temp;
    int position;

    for (k = 1; k < size; k++)
    {
        temp = array[k];
        position = k;

        while ((position > 0) && (array[position-1] > temp))
        {
            array[position] = array[position-1];
            position = position - 1;
        }
        array[position] = temp;
    }
}
```

Merge Sort:

```
void merge(int* array, int* tempArray, int low, int middle, int
high)
{
    int i, j, k;

    for (i = low; i <= high; i++)
    {
        tempArray[i] = array[i];
    }

    i = low;
    j = middle+1;
    k = low;

    while ((i <= middle) && (j <= high))
    {
        if (tempArray[i] <= tempArray[j])
            array[k++] = tempArray[i++];
        else
            array[k++] = tempArray[j++];
    }

    while (i <= middle)
        array[k++] = tempArray[i++];
}

void mergeSortHelper(int* array, int* tempArray, int low, int
high)
{
    if (low < high)
    {
        int middle = (low + high) / 2;
        mergeSortHelper(array, tempArray, low, middle);
        mergeSortHelper(array, tempArray, middle+1, high);
        merge(array, tempArray, low, middle, high);
    }
}

void mergeSort(int* array, int size)
{
    int* tempArray = new int[size];
    mergeSortHelper(array, tempArray, 0, size-1);
    delete [] tempArray;
}
```

Mystery Sort:

```
void swap(int* array, int position1, int position2)
{
    int temp = array[position1];
    array[position1] = array[position2];
    array[position2] = temp;
}

void mysterySort(int* array, int size)
{
    int j, k;
    for (j = size-1; j > 0; j--)
    {
        for (k = 0; k < j; k++)
        {
            if (array[k+1] < array[k])
            {
                swap(array, k, k+1);
            }
        }
    }
}
```