Jackie Diep

Assignment 1

CSC 413 G001

3/1/21

Chapter 1

1. **a) Find gcd(31415, 14142) by applying Euclid's algorithm.**

   31415 = 2 * 14142 + 3131

   14142 = 4 * 3131 + 1618

   3131 = 1 * 1618 + 1513

   1618 = 1 * 1513 + 105

   1513 = 14 * 105 + 43

   105 = 2 * 43 + 19

   43 = 2 * 19 + 5

   19 = 3 * 5 + 4

   5 = 1 * 4 + 1

   1 = 1 * 1 + 0

   gcd(31415,14142) = 1

   b) Estimate how many times faster it will be to find gcd(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from min(m, n) down to gcd(m, n).

   Euclid's algorithm requires 10 operations whereas the consecutive integer checking algorithm uses an amount equal to and up to double the smallest integer, in this case 14142.
   So, Consecutive min: 14142, and Consecutive max: 28284, and Euclid's algorithm is between 1414.2 and 2828.4, inclusive, times faster.

Chapter 2

2. **Using Figure 2.2 as a model, illustrate the operation of Insertion-Sort on the array A = {31, 41, 59, 26, 41, 58}**

{31, 41, 59, 26, 41, 58} → {26, 31, 41, 59, 41, 58} → {26, 31, 41, 41, 59, 58} →

{26, 31, 41, 41, 58, 59}

3. **Using Figure 2.4 as a model, illustrate the operation of merge sort on the array A = {3, 41, 52, 26, 38, 57, 9, 49}**

{3, 9, 26, 38, 41, 49, 52, 57}

{3, 26, 41, 52} {9, 38, 49, 57}

{3, 41} {26, 52} {38, 57} {9, 49}

{3} {41} {52} {26} {38} {57} {9} {49}

4. **Write pseudocode for *linear search*, which scans through the sequence, looking for *v*. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.**

```
int aLength = sizeof<A>/sizeof<A[0]>// find the size of the array
for (int i = 0; i < aLength; i++)
{
        if A[i] == v { return i; }                  // if v is found
}
return NULL;                                        // if v is not found in the array (aka “nil”)
```

Loop Invariant: v is not found in A[i-1]

Initialization: A[i-1] is a random memory address not part of the array, so it does not contain the value we are looking for

Maintenance: If the value we are looking for is found in the current iteration of the loop, the loop ends. If the value we are looking for is not found in the current iteration, the loop continues. For whatever current value of i, v will not be found in A[0] through A[i-1].

Termination: If the value v is found, the loop terminates by returning i. If the value is not found by the end of the array, the loop terminates and returns NULL (or nil). Either result fulfills the purpose of the loop.

5. **2.2-2) Write pseudocode for *selection sort*. What loop invariant does this algorithm maintain? Why does it need to run for only the first n - 1 elements rather than for all n elements? Give the best-case and worst-case running times of selection sort in θ-notation.**

```
int aLength = sizeof<A>/sizeof<A[0]>// find the size of the array
double temp = A[0];
for (int i = 0; i < aLength - 1; i++)
{
int tempHolder;
for(int j = i; j < aLength; j++) {
        if (temp > A[j]) {
                temp = A[j];
                tempHolder = j;
                }
        }
A[tempHolder] = A[i];
A[i] = temp;
}
```

Loop invariant: A[temp] is <= A[i...j-1]
Initialization: A[temp] is = to A[0], the only element, so it is the smallest element in the subarray.
Maintenance: If temp > A[j], temp is set = to A[j] and it is once again the smallest element currently in the subarray. If temp is <= A[j], no action is taken for that iteration.
Termination: After the inner loop, A[j] is exchanged with A[i] to such that the elements currently in the subarray are in ascending order.

Best-case:$\theta(n^2)$
Worst-case: $\theta(n^2)$
Both are the same because for every value in the array, each selection must be checked to ensure correctness.

The final entry of the array does not need to be sorted because it should naturally already be the largest selection remaining.

2.2-3) Linear Search: How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in θ-notation? Justify your answers.

Elements checked:

Average-case: (n+1)/2

Because if it is equally likely, then the average would be the best + the worst divided by two.

Worst-case: n

If it is the worst case, then all elements of the array would have to be checked.

Running time:

Average-case: θ(n)

Worse-case: θ(n)

If constant terms ½ and 1, respectively, are removed, then the remainder is θ(n) for both outcomes.

6. **2.3-3) Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence … is T(n) = *n**lg(*n*).**

T(2) = 2log2

Assume that when $n = 2^k$ and $k > 0$, that T(n) = n*lg(n)

If $n = 2^{k+1}$,

$T(2^{k+1}) = 2T*(2^{k+1}/2) + 2^{k+1}$

$T(2^{k+1}) = 2*(2^k*lg(2^k)) + 2^{k+1}$

$T(2^{k+1}) = 2^{k+1}*(lg(2^k) + 1)$

$T(2^{k+1}) = 2^{k+1}*lg(2^{k+1})$

so, T(n) = n*lg(n)

2.3-4) We can express insertion sort as a recursive procedure as follows. In order to sort A[1..n], we recursively sort A[1..n - 1] and then insert A[n] into the sorted array A[1..n - 1]. Write a recurrence for the running time of this recursive version of insertion sort.

```
void insert(int i, int arraySize, double toSort[])
{
int j = i;          // Iterate through subarray using integer j without altering integer i

while (toSort[j - 1] > toSort[i] && j >= 1)              // iterate to the destination of index i
{
        toSort[j] = toSort[j - 1];
        j--;
}

toSort[j] = toSort[i]                                    // enter the value of index i at the destination

                        // increment index i by 1 and recursively call the insert function
if (i + 1 <= arraySize) { insert(i + 1, arraySize, toSort[]); }
}
```