Homework 2
Emily Costa

Problem 1
My approach to solving the programming problem was laying out the steps by comments and filling it out based on the pseudo-code provided in this course. To do the addition of checking for an already sorted array, I simply created an additional function that iterated through the subarray and did the check and added a conditional statement into the merge() function. The pseudo-code was sometimes confusing when it came to the index starting at 0 in C++, not 1. A challenge I found was really just dealing with C++ syntax. A few of these issues caused segmentation faults and I had to spend a good amount of time addressing them. The concept of Merge Sort was clear and easy to me. Here is a sample output of my program:

```
Input the number of integers in A, n: 15
Array before mergesort:  83 86 77 15 93 35 86 92 49 21 62 27 90 59 63
Already sorted subarray: 83 86
Already sorted subarray: 86 92
Array after mergesort:   15 21 27 35 49 59 62 63 77 83 86 86 90 92 93
```

Problem 2
In order to no longer need the sentinel value of infinity, one could simply check if i or j reached n_1 or n_2, respectively, then fill out the resulting array with the leftover values from i or j, as those are already sorted.

Instead of:
for k=p to r {...}

Use:
while i < n_1 and j < n_2 {...}
for i to n_1
  A[k] = L[i]
  i = i + 1
  k = k + 1
for j to n_2
  A[k] = R[j]
  j = j + 1
  k = k + 1

Problem 3
a. This function returns 2 to the power of n.
b. F2 is significantly faster as n->inf
c. The time complexity of F1 is O(2^n) and the time complexity is O(n), or linear, for F2.

Problem 4
a. The purpose of ProcedureX is the sort the integers in ascending order. It achieves this by placing the least integer from i to j at position i, shifting the i pointer every time the first loop occurs. Once the i pointer passes every position in the array, the array is sorted.

b. The time complexity of the worst-case scenario would be O(n^2). This is because, for n elements, each element needs to loop through an array up to a size of n. So the time complexity becomes n*n, or n^2.

Problem 5

```
insertion_sort_recursive(A, i):
    if i=1: return
    insertion_sort_recursive(A, i-1)
    do_insert_task(A, i-1) # no need to give algorithm for this
```

This algorithm has a time complexity of O(n^2). This can be found by analyzing the running time equation:

T(n)

= T(n-1)+n for n>1 (we ignore n=1, as it becomes irrelevant as n->inf)

= T(n-2)+(n-1)+n

= T(n-3)+(n-2)+(n-1)+n (it does continues this pattern n times)

= O(n*n)

= O(n^2)