Keenan Miller
CSCI-261 HW2

1. Max Rectangle Area Problem
    a. My implementation of the max rectangle area algorithm utilizes assumptions made about the given data to its advantage. This algorithm begins by creating a bar-graph in the exact shape of the given polygon. It iterates over each x-value and fills in an integer array representing the bar-graph (x values are indices and y values are the values in the array).

        Once we are ready to find the largest-area rectangle in the bar-graph, we instantiate a stack (just an integer array in this case, where we keep track of the last-in value's index). The stack represents all the "open" rectangles. These will be closed (and areas checked) as the y-value moves back down. We also create another integer array that keeps track of each index's "left-support". This means, for each bar (being the top of its own rectangle) the left-support will be the maximum left side of the given rectangle (again tracked by its index).

        We begin looping over each bar in the bar-graph. Within this loop, if we have any bars on the stack that happen to be larger than the current bar, we "close" that bar's rectangle and record its area if it is the new max area (here we also remove it from the stack). We can close it because the fact that the current bar is smaller means that the rectangle of the bar on the stack can no longer expand to the right. This "closing" occurs in a nested loop, but it doesn't increase the complexity of the overall algorithm because this nested loop will only loop once per index (total, not per outer-loop), since each bar in the graph can only be added to it once. After that nested loop, we close the left side of the current bar's rectangle with the highest bar still on the stack (since it is lower and will be a left bound for this rectangle). Finally we add the current bar to the stack because it is yet to be closed.

    b. Pseudocode

```
Read in data as points;

barGraph = integer array;

// fill in barGraph representing the polygon
// this is two loops, but is O(n) because it only iterates
// once per x value.
for point in points (from i = 0 to points.length) {
    for j from lastX to point.x {
        graph[j] = point.y;
    }
    lastX = point.x;
}

maxArea = 0;
open = integer array; // this works like a stack first-in-last-out queue
int openIndex = 0; // this is the stack index (where the first out is)

// this represents the 'left side' of the rectangle for the index of bar
```

```
leftSupportOfIndex = integer array;

// O(n)
for each bar in barGraph (from i = 0 to barGraph.length){
        while there is a value on the stack that is greater than bar {
                // bar represents the right edge of the rectangle, and
                // the value on the stack is the bar in the graph that
                // is a left side of the same rectangle
                if this rectangle is bigger than maxArea {
                        maxArea = this rectangle;
                }
                openIndex--; // essentially remove this item from the stack
        }

        // we can 'close' the left side of the current bar with the
        // highest still-open value
        leftSupportOfIndex[i] = open[openIndex];

        // add this bar to the stack
        open[1 + openIndex] = i;
        increment openIndex;
}
return maxArea;
```

c. This algorithm will always produce the correct result because it considers every possible rectangle that has a height equal to its upper bound. Since it is able to consider every possibly-maximum rectangle, it will always be able to compare and find the largest one.
d. This algorithm runs with θ(n) complexity.
e. This algorithm performs a O(n) complexity loop to convert the polygon to a bar graph. Subsequently, it performs another O(n) loop over each x value while adding some of these to the stack. Each x value will be added to the stack at most once, and there is a loop within the O(n) loop which doesn't loop over each point, but only a handful on the stack (while removing them, so this inner-loop will at max run once per x value in total, therefor not increasing complexity). Because of how this main loop is designed, it would run with O(2n) complexity, which is simplified to O(n) as the 2 is irrelevant. Since no operations exist over O(n) complexity, this algorithm runs at θ(n).

2. WhatDoIDo
    a. $T(n) = 2T(\frac{n}{2}) + O(1)$
    b. Since: $T(n) = 2T(\frac{n}{2}) + O(1)$

       Note that:

       $T(\frac{n}{2}) = 2T(\frac{n}{4}) + O(1)$

       $T(\frac{n}{4}) = 2T(\frac{n}{8}) + O(1)$

       $T(\frac{n}{8}) = 2T(\frac{n}{16}) + O(1)$

       Substitute in the $T(\frac{n}{2})$ value from above:

$T(n) = 2(2T(\frac{n}{4}) + O(1)) + O(1)$

Simplify:

$T(n) = 4T(\frac{n}{4}) + 3O(1)$

Substitute in the $T(\frac{n}{4})$ value from above:

$T(n) = 4(2T(\frac{n}{8}) + O(1)) + 3(O(1))$

Simplify:

$T(n) = 8T(\frac{n}{8}) + 7(O(1))$

As can be seen here, where k is any value, and C represents any number of constant-time operations:

$T(n) = 2^k T(\frac{n}{2^k}) + C$

Let $T(\frac{n}{2^k}) = T(1)$ as $k \to \infty$

This means that $\frac{n}{2^k} = 1$ which means $k = log_2(n)$

Plug in $k = log_2(n)$ to $T(n) = 2^k T(\frac{n}{2^k}) + C$ :

$T(n) = 2^{log(n)}(1) + C$

Simplify:

$T(n) = 2^{log(n)}$

Since it is a fact that $2^{log(n)} = n$, then

$T(n) = \theta(n)$

c. The given algorithm finds the sum of all values in the array A between indices "left" and "right".

3. Weighted Inversions
   a. My algorithm to determine the sum of weighted inversions in a given integer permutation uses a structure identical to that of mergesort, but with a few important distinctions to allow for counting and summation of weighted inversions that exist in the original list.

      When the input is read in, an array of Entry objects is instantiated, each of which includes the value, originalPosition (index in the input-permutation), and the inversionMagnitude (described later). All of which are primitive longs. This array of Entry objects is what is sorted by the implementation of mergeSort that I used.

      When passed into the mergesort, the recursive method functions identically to its counterpart in a standard mergesort implementation. The merge method is where the modifications have been made:

      The merge method first adds the total weighted-inversion sums from both Responses (for the right and left arrays that it will merge). It keeps track of this for later. While the arrays are being merged, when a value from the right array is about to be merged into the left array, inversions exist which have to be accounted for. This can be accounted for using the following formula:

```
formula A = (originalPosition of Entry in the right array) x (length of the left
array - index of Entry in the left array) - (the inversionMagnitude of the Entry in
the left array)
```

The inversionMagnitude is a value which must account for all weighted inversions between the two values being compared. For example, if we have two arrays,

LEFT = {1, 4, 6}
RIGHT = {2, 8, 10}

Say we are at a step where we are comparing LEFT[1] and RIGHT[0], we must not only account for the weight of the position-shift between the '2' value and the '4' value, but also the '6' value from the LEFT array. We can see that the inversionMagnitude for the '6' entry in the LEFT array is simply its originalPosition because any inversions involving it will simply have a weight of the original position of the RIGHT entry minus that originalPosition (6's inversionMagnitude). Thus, the inversion magnitude of the entry to its left (4), will be the sum of 6's inversionMagnitude and 4's originalPosition. Since these inversionMagnitudes recorded for the LEFT array are independent of the Entry objects in the RIGHT array, a helper function is used which iterates over each element in the left array from the last Entry to the first Entry (this happens outside of any other loop in the merge function) and assigns each Entry in LEFT an inversionMagnitude based off the formulae described above.

Each result of 'formula A' above is recorded and added to the total inversion count of each smaller Entry array that was passed into the merge function originally. This is returned along with the now-sorted Entry array (as a Response object, containing said information).

b. Pseudocode

```
array Entry[];

mergeSortAndCount (array) {
        if array length < 2 {
                return Response with count = 0 and array;
        }

        leftArray = first half of array;
        rightArray = second half of array;
        leftResponse = mergeSortAndCount(leftArray);
        rightResponse = mergeSortAndCount(rightArray);
        return mergeAndCount(leftResponse, rightResponse);
}

mergeAndCount (left, right) {
        count = left.count + right.count;

        // O(n)
        augmentInversionMagnitudes(left);
```

```
        sortedArray = [];
        while either left or right has entries that haven't been counted {
                if left has entries that haven't been counted AND current left entry
                is less than current right entry {
                        add left entry to sorted array;
                } else if left has entries that haven't been counted {
                        add right entry to sorted array;
                        // increase count
                        count += (right entry's original position)*(left array length
                        - current left index) - (left entry's inversion magnitude);
                } else {
                        add right entry to sorted array;
                }
        }
        return count and sortedArray;
}

augmentInversionMagnitudes (array) {
        currentMagnitude = 0;
        for entry in array from last to first {
                entry = entry's original position + currentMagnitude;
        }
}
```

c. This algorithm always produces the correct result because it always accounts for the original index of each entry in the input array, and uses it in conjunction with any elements rearranged at any point throughout the process. During the array-merge process, whenever an entry from the right side is moved in front of any entries from the left side, all left-entries greater than it have their original position subtracted from that of the right-entry's original position, which by definition produces the weighted-inversion count when added to the current count and the current-counts of both left and right arrays.
d. This algorithm runs with $\theta$(nlogn) complexity.
e. This algorithm runs with $\theta$(nlogn) complexity since this algorithm is an exact implementation of mergesort, with a few additional constant-time operations (irrelevant to running time complexity), and with a single additional iterative loop which runs with O(n) complexity and is run sequentially, rather than nested, with the existing O(n) operation in the merge function of mergesort.

4. Master Theorem Problems

1. $T(n) = 9T(\frac{n}{3}) + n^3$

   $a = 9$
   $b = 3$
   $k = 3$
   $log_b a = log_3 9 = 2$
   Since $log_b a < k$, we use case 3:

$T(n) = \theta(n^3)$

2. $T(n) = \frac{3}{2}T(\frac{2n}{3}) + 3$

   $a = \frac{3}{2}$

   $b = \frac{3}{2}$

   $k = 0$

   $log_b a = log_{\frac{3}{2}} \frac{3}{2} = 1$

   Since $log_b a > k$, we use case 1:

   $T(n) = \theta(n^{log_{\frac{3}{2}} \frac{3}{2}}) = \theta(n)$

3. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

   $a = 2$

   $b = 4$

   $k = \frac{1}{2}$

   $log_b a = log_4 2 = \frac{1}{2}$

   Since $log_b a = k$, we use case 2:

   $T(n) = \theta(n^{log_4 2} log(n)) = \theta(n^{\frac{1}{2}} log(n)) = \theta(\sqrt{n}\, log(n))$