CS325 Group Assignment 1

Shane Barrantes, Griffin Gonsalves, John Miller, Steven Powers

October, 11th 2016

Overview of Algorithm

Our algorithm uses merge sort as the basis for a solution to achieve $O(n\ logn)$ run time. By using merge sort, the amount of compare operations is reduced, and the form of $O(n^2)$ obtained by a recursive solution is then changed to $O(n\ log\ n)$.

The algorithm intakes an input file structured like:

```
point_count
sorted_q_values
unordered_p_values
```

The algorithm then performs a merge sort on the unordered_p_values. Merge sort halves the input array until it breaks it down into its smallest subproblems.

Example:

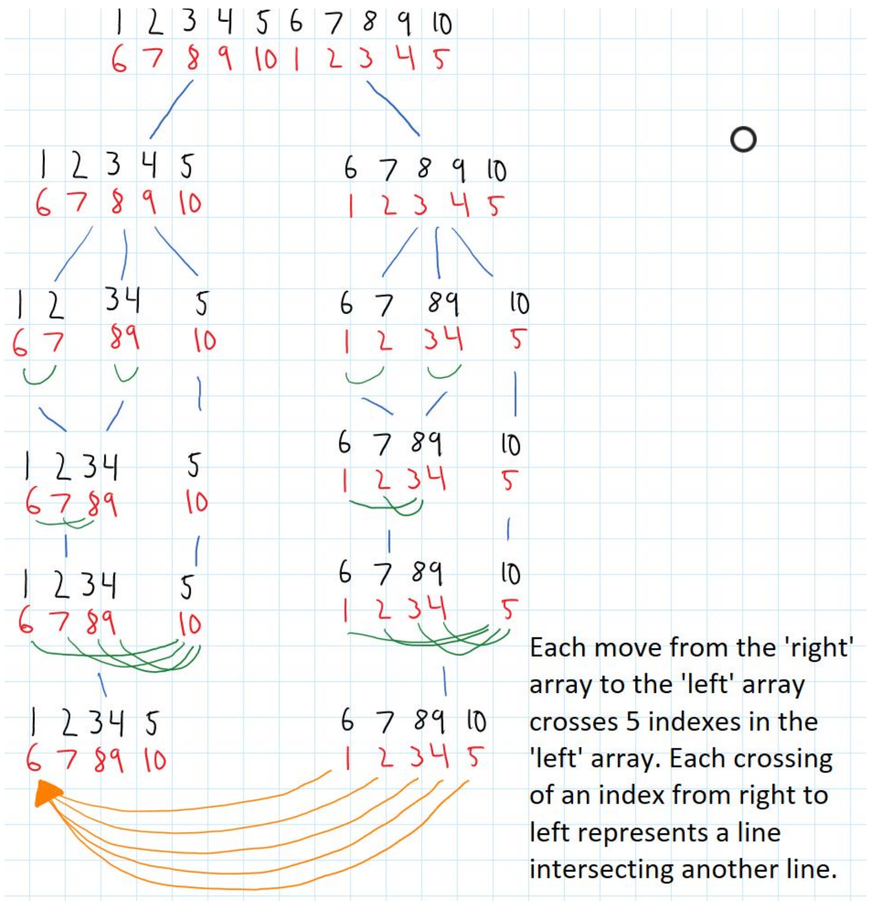
```
          1,4,2,3,5
       1,4,2      3,5
     1,4  2        3  5
  1    4    2       3    5
```

The algorithm then merges these subproblems back together, comparing the values in the "left" subproblem to the values in the "right" array (e.g. 1 vs 4 and 1,4 vs 2). If the value in the "right" array is smaller than that in the left array, we know that at some point the lines connecting those points intersect. If the value in the "right" array must cross over multiple values in the "left" array then we know that the lines connecting those points all intersect as well. By comparing the indexes of those array values we can determine how many lines intersect when merging values in a "left" and "right" array.

The example shown below illustrates an approximation of how the algorithm determines the number of intersections.

Example

Here is a manual demonstration of how the algorithm finds the number of intersections between lines. The black numbers represent the *q* values and red the *p* values. The data set used for this manual example is the input.txt file from directory 6 in the test files provided.


1 2 3 4 5 6 7 8 9 10
6 7 8 9 10 1 2 3 4 5
Each move from the 'right' array to the 'left' array crosses 5 indexes in the 'left' array. Each crossing of an index from right to left represents a line intersecting another line.

We can see that the process is virtually identical to mergesort, the only difference being that at each merge step, when comparing two values, we count the number of indexes a value from the right array must cross when being moved to it's new sorted position.

Pseudocode

```
countIntersections(p[1..m], p[m+1..n]):
        a ← 0; //Left
        b ← 0; //Right
        c ← 0; //count
        m ← 0; //Midpoint

        if p[1..m] < p[m+1..n]:
            m ← (p[1..m] + p[m+1..n]) / 2;
            a ← countIntersections(a, m);
            b ← countIntersections(m + 1, p[m+1..n]);
            c ← merge(p[1..m], m, p[m+1..n]); // Returns number of intersections

        return a + b + c;
```

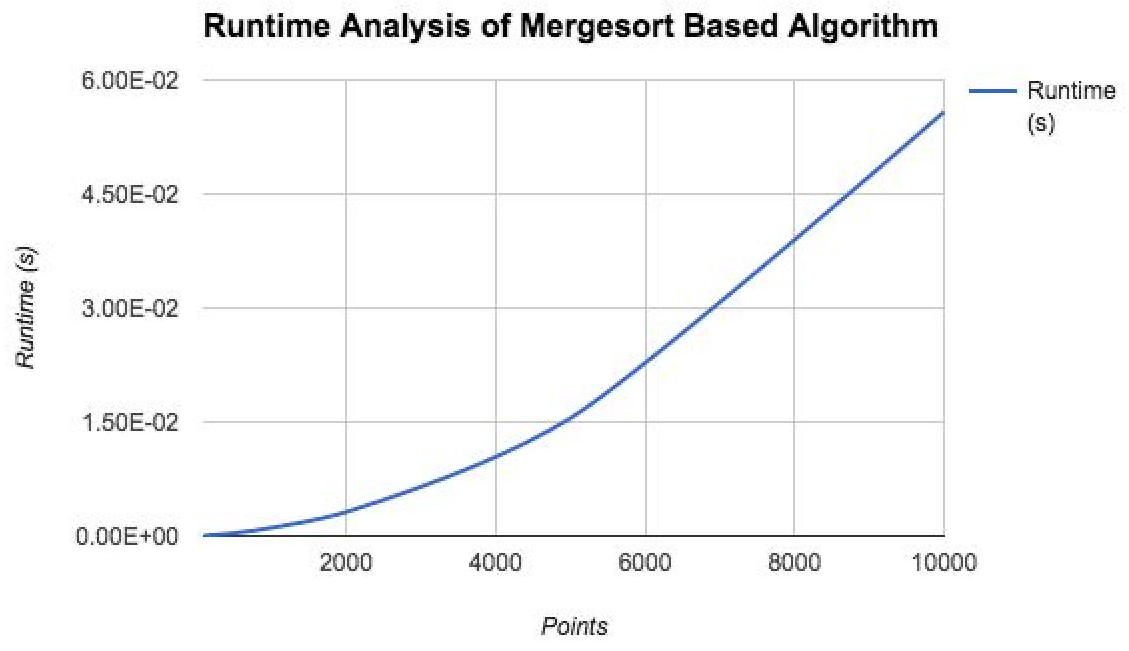
The pseudocode above contains conditional statement which is used to provide the recursion for our count intersections function if the points to the left of our pivot point are less than that of the points to the right of our pivot point. If this is true, we determine the midpoint m, by adding up the points and then dividing by two. Following this we recursively call from our left point a to the midpoint m. We also call from our midpoint plus one to the end of the array. Following this we also use a black box merge function that calculates the number of intersections when merging from right to left.

Proof of Correctness

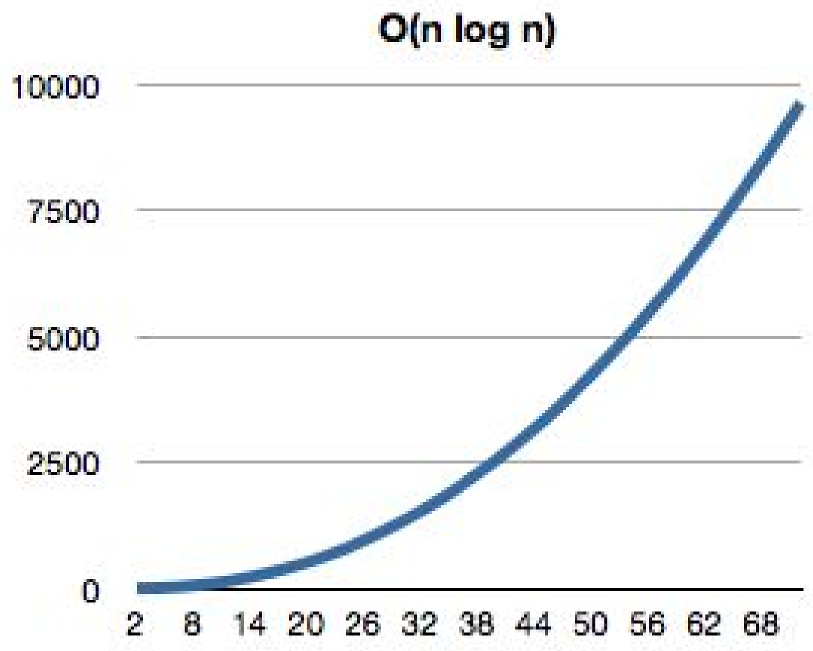
```
Base Case:
       Let arrays p and q have one point.
       q=1
       p=1

       Since the array p has length 1 and the list q has length 1, therefore
       the lists are sorted and merge() gives zero intersections.

Inductive Hypothesis:
       Suppose that our algorithm counts all intersections for p ≥ 1, correctly using the
       merge() function within a working MergeSort().

Induction Step:
       MergeSort() can sort an array of points A[] of size n.

       By calling MergeSort() we split A[] using the midpoint m of size n/2 and  between

       The indices p and q are temporary endpoints such that A[p,m] and A[m+1,q].

       Next we call Merge(), which counts the number of
       Intersections and returns a sorted array of points p.

       By the Induction Hypothesis, this operation will succeed because our new array A[] is
       sorted correctly by MergeSort() and the intersections are counted using Merge().
```

Runtime Analysis
To analyze the runtime of our algorithm we created test data sequences of sizes 100, 500, 1000, 2000, 5000, 10000, ran each sequence against our algorithm 10 times and average the runtime of each set. The graph below is the result of that data.
Data:

Data Points	Average Runtime
100	$4.838e^{-5}$
500	$3.897e^{-4}$
1000	$3.084e^{-3}$
2000	$1.093e^{-3}$
5000	$1.547e^{-2}$
10000	$5.585e^{-2}$



Looking at the graph the runtime appears to be that of $n\ log(n)$ which is show in the image below


O(n log n)
10000
7500
5000
2500
0
2
8
14
20
26
32
38
44
50
56
62
68

Bibliography:

[1] "Merge Sort Powered by HackerRack," in *HackerRank.com*. [Online]. Available: https://www.bing.com/search?q=merge%20sort%20python&qs=n&form=QBRE&pq=merge%20sort%20python&sc=6-16&sp=-1&sk=&ghc=1&cvid=2C49E03FF38849479CBC385A3BFB55B7. Accessed: Oct. 5, 2016.