CS 203

Assignment 2

Closest Pair Algorithm

By:

Colin Quinn

**Preview:**

To execute the programs connected to this assignment, load each java file into a preferred IDE, compile, then execute. In order to execute DivideAndConquer.java, Point.java must also be available as the Point class is used in the other java file. Within the console or command prompt, instructions will be given upon execution, but all that is needed from the user is a number larger than zero then pressing the ‘enter’ key. The goal of both algorithms included is to find the closest pair of points from a randomly generated list of ‘x’ and ‘y’ coordinates. BruteForce.java compares every point with all of its successors while keeping track of the minimum distance as well as the pair that computes that distance, while DivideAndConquer.java uses a more strategic divide and conquer method to lessen the amount of work required. The data used was collected on an AMD Ryzen 7 1800x clocked at 3.6GHz.

**Brute Force Theoretical:**

Due to the specifications of this algorithm, it does not rely on the information within the list of points. In order to find a minimum distance, it compares every possible pair of points. As a result of this, there is no best or worst case for this algorithm and a closed-form formula can be found. In choosing a basic operation, there are two options that do not alter the efficiency analysis, in the calculation of distance between points and the comparison of current distance and current minimum distance. This assumes that the time taken to calculate distance is constant between iterations, otherwise the comparison of values is the option to select. The basic operation is performed once per iteration of the inner loop, while the outer loop executes *n – 2* iterations. Meaning the classification of the algorithm is C(n) = = = (n – 2)(n – 1). Therefore C(n) ϵ θ(n2 ) where *n* the number of points that must be compared.

**Brute Force Empirical:**

While a brute force approach is acceptable for small values of *n*, it begins to struggle in larger cases. As previously discussed, this algorithm’s runtime tends to a slightly quadratic trend, however the previous analysis shows that C(n) ϵ θ(n2 ). At an *n* value of 100, the average run time is 0.5997 milliseconds, with a slight quadratic trend through an *n* value of 1000 at 11.8518 milliseconds, and 24.5746 milliseconds at *n* = 2000. Times calculated are an average of 3 executions in an attempt to adjust for any background processes that could have an impact on runtimes. While the differences in these tests are bearable, this only includes smaller *n* values and the times continue to grow along with the quadratic pattern.

**Divide and Conquer Theoretical:**

This divide and conquer algorithm does not rely on the contents of an array, but instead relies only on the size of the array. However, the only case that would rely on contents of the array is the initial quicksort algorithm, which is time complexity T(n) ϵ O(n2 ) when the array is already sorted. It will be assumed that the random number generation does not generate a sorted array due to this sort being the potential bottleneck for the algorithm making it T(n) ϵ O(n2 ), Ω(n \* Log(n)). The best case for this algorithm comes from the repetitive calls to a merge sort function which is T(n) ϵ θ(n \* Log(n)). As the main algorithm utilizes a brute force function to solve the problem for sizes of *n <= 3* which runs in T(n) ϵ θ(n). It then copies the left and right sides of the array which utilizes a java function that is T(n) ϵ θ(1). The recursive calls then split the current problem size in half, and then solves each half, which is the first part of the master theorem in that T(n) = 2 \* T(n/2) + F(n). Where F(n) is the work that is required by the algorithm after the recursive calls. The function to find minimum of two numbers runs in constant time. That distance is then used to create a list of points within that distance which is F(n) ϵ θ(n). This list of points is then sorted using mergesort which is F(n) ϵ θ(n \* Log(n)). Lastly, the algorithm must find the absolute minimum, which is done in constant time F(n) ϵ θ(1). Therefore in combining all of these parts, F(n) = n + (n \* Log(n))+ n = 2n + (n \* Log(n)) ϵ θ(n \* Log(n)). This overall algorithm then runs in T(n) = 2 \* T(n/2) + (n \* Log(n)), therefore being T(n) ϵ θ(n \* Log(n)) assuming that the initial quicksort does not run in the worst case efficiency.

**Divide and Conquer Theoretical:**

The divide and conquer algorithm is a much more complex solution that yields worthy results. The algorithm’s reliance on sorting is where it gets the main bottlenecks in performance as there is only certain ways to sort an array. Run times were calculated based on an average time in milliseconds over 3 separate runs. For smaller *n* values, the algorithm quickly increases in run time as it starts at *n = 100* at 0.4252 milliseconds, and at *n = 200* is 0.8698 milliseconds. The growth in time becomes more consistent at *n = 700* where the increases become more linear than logarithmic. Such as between *n = 900* and *n = 1700*, the run time was all between 3 and 4 milliseconds, showing a slowly increasing near linear trend.