Programming Assignment 1

# Overview

(a)

```text

CalculateClosestPoints(P, m)

(1)  ClosestPoints = []

(2)  i = 1

(3)  while (i < A.Length)

(4)     j = i + 1

(5)     while (j < A.Length)

(6)         CurrentManhattanDistance = CalculateManhattanDistance(P[i], P[j])

(7)         InsertedIntoArray = false

(8)         k = 1

(9)         while (k < ClosestPoints.Length and !InsertedIntoArray)

(10)            if (CurrentManhattanDistance < ClosestPoints[k].ManhattanDistance)

(11)                ClosestPoints.Insert(CurrentManhattanDistance)

(12)                InsertedIntoArray = true

(13)                if (ClosestPoints.Length > m)

(14)                    ClosestPoints.Pop()

(15)            k = k + 1

(16)        if (ClosestPoints.Length < m)

(17)            ClosestPoints.Append(CurrentManhattanDistance)

(18)        j = j + 1

(19)    i = i + 1

(20) return ClosestPoints

```

<br>

As one can see by looking the pseudocode for the CalculateClosestPoints method, a nested while-loop iterates over $P$ to calculate the manhattan distance for every combination of data points. A key aspect to note here is that the inner loop of this sequence (the *\*j-loop\**) starts each iteration at $j = i + 1$, this is because all combinations previous to this index have already been calculated and handled on previous iterations of the *\*i-loop\**. For this portion alone, an exact running time of $O(\frac{n(n-1)}{2})$ can be found, which is asymptotically considered $O(n^2)$.

On each iteration of this nested while-loop, thus, on every calculation of a manhattan distance data point, one last loop (the *\*k-loop\**) iterates over the return array to properly insert, append, or discard the current data point. As this array will at most be of size $m$, but starts with a size of $0$, the worst case running time of the *\*k-loop\** is $\Theta(\frac{1}{2}m)$, which is asymptotically considered $\Theta(m)$.

Combining the time of both portions yields a *\*worst-case\** asymptotic running time for the algorithm of $O(n^2) \cdot \Theta(m) = \Theta(mn^2)$. After some continued development, I realized that the max value of $m = {n\choose2}$ is equivalent to the total number of point combinations made in the *\*i-loop\** and the *\*k-loop\**, this basically means returning every combination as a sorted list in this case. From this realization, the worst-case running time for the *\*k-loop\** can be seen as $\Theta(\frac{n(n-1)}{2})$, which is asymptotically considered $\Theta(n^2)$.

This brings the asymptotic worst-case running time for the algorithm to $O(n^2) \* \Theta(n^2) = \Theta(n^4)$.

For sake of clarity, I want to note the exactly derived worst-case running time as $O(\frac{n(n-1)}{2}) \cdot \Theta(\frac{1}{2}m) = \Theta(\frac{1}{2}m \cdot \frac{n(n-1)}{2}) = \Theta(\frac{n(n-1)}{4} \cdot \frac{n(n-1)}{2}) = \Theta(\frac{n^2(n-1)^2}{8})$. This will be displayed on the test run graph as a reference point to test points and asymptotic worst-case running time.

(b) See Source/Python/ManhattanDistance.py -> CalculateClosestPoints()

(c) See TraceRuns.txt

(d) See TestRuns.txt

(e) At first glance, I did not think that the runtime complexity of this problem was going to become so immense. I think this was partially due to an oversight that the size of the return array, $m$, was going to be as large as it got in it's worst cases. Consider it, there is a massive difference between generating every point combination for say $n=100$ points and comparing to an return array of size ${n\choose2}$, rather than to comparing it to a return array of size $m = 10$.

In implementing this algorithm, I went through several different renditions in attempts to reduce the runtime complexity. One for example was to collect all data points in an unsorted array, then loop through again to place them in a sorted array, but this happened to give the same complexity, as it is turning an array of size $n$ into an array of data of size $\frac{n(n-1)}{2}$, then performing an insertion sort into an array of size $\frac{n(n-1)}{2}$.

Simply put, collecting $\frac{n(n-1)}{2}$ data points alone begins to have a significant running time, not even considering any sort of algorithm for inserting into a sorted array.

![TestRunsGraph-2.png](attachment:TestRunsGraph-2.png)

The graph above shows actual number of keypoint operations counted as number of loop iterations compared to the theoretical worst case running time, the algorithm $\Theta(\frac{n^2(n-1)^2}{4})$. The test points used were $n = 10, 25, 50, 100, 250$, using random generation, an input array of size $n$ was populated with random points such that $-1000 \leq X \leq 1000$ and $-1000 \leq Y \leq 1000$. By looking at the graph, one can see that all test points are less than the theoretical worst case running time, this is because with random points, it cannot be determined when a point will be inserted and the *\*k-loop\** will be terminated early, this is where the worst-case running time deviates from excremental running time.

**# 3. Retrospection**

From my analysis in Section 2 (e), there is no avoiding generating the combination of all input points into data points to be sorted, this will always require O($\frac{n(n-1)}{2}$). Because of this, the only place I see where we are able to reduce worst-case runtime is inserting into the return array. Right now the current algorithm for inserting is done using an insertion sort, which is $\Theta(n^2)$, where $n$ is generic size of array to sort. If instead something like a binary tree were used for storing sorted values, the worst-case running time is $\Theta(lg(n))$, where $n$ is generic size of array to sort. Even with a small enhancement like this, the overall worst-case asymptotic running time would decrease from $\Theta(n^4)$ to $\Theta(n^3lg(n))$.