Jenna Ellis
Project 2 Report

1. Tilde Approximation:

Brute Implementation:

To obtain the tilde approximation for this implementation, I walked through the major loop and comparison structures used in my code in this class. The terms for the equation in terms of N, where N is the number of points being investigated for a given execution, are calculated as follows:

Structure Pseudo-Code	Equation in terms of N	Comments
For I = 0 to N{ read from input }	N	Reads N points from input, a constant need for the implementation

For I = 0 to N
For j = i+1 to N
For k = j+1 to N
For l = k+1 to N
sort(collinear)
end
end
end
end
	$16 * (N)(N-1)(N-2)(N-3)$
	$= 16(N^4 - 6N^3 + 11N^2 - 6N)$
	$= 16N^4 - 96N^3 + 176N^2 - 96N$
		Represents the four nested for loops used in brute to find all combinations of 4 points. The sort method called always only operates on arrays of size 4, with an N^2 sorting algorithm. This incorporates the 16 coefficient into the multiplication of complexities of each nested for loop.

The above table highlights the large loop structures that the majority of computation stems from. By summing the terms of these large structures, the complete approximation equation for Brute implementation is: $16N^4 - 96N^3 + 176N^2 - 95N$. This is tilde approximated as: $\sim 16N^4$.

Fast Implementation:

To obtain the tilde approximation for the Fast implementation, I did the same process as listed above for Brute with also investigating the complexity of the Arrays.Sort algorithm

Structure Pseudo-Code	Equation in terms of N	Comments
For I = 0 to N {read from input} For I = 0 to N {copy points for a secondary reference array}	2N	Reads N points from input, a constant need for the implementation. Copy for later loop control in main algorithm.

for I = 0 to N
  let current = points[i]
    for j = 0 to N
      //define angles b/w current and points[j]
    end
arrays.sort(points)
print(points, compare)
end
	$Outer\ loop\ complexity: N$
	$set\ current\ to\ point, constant$
	$Inner\ loop, N\ complexity$
	$Term\ in\ outer\ loop, sort$
	$\rightarrow N\log(N)$
	$print\ complexity$
	$\rightarrow N + Constant$
	Therefore: N * (N*log(N)+ N+(N+N/2)) = $N^2\log(N) + \frac{5}{2}N^2$
		Iterates through N points to make each the origin. For each origin, the angles are calculated for all N points. After the angles are calculated, the points are sorted as N Log N complexity sorting algorithm. Print complexity requires all N points, and then a constant dependent upon the number of points that are collinear, estimated as $\frac{N}{2}$. All of these multiplied due to nesting within the initial for loop.

From the above analysis of my main algorithm for fast, it can be determined that the approximate equation representing complexity with relation to N, the number of points read in, is: $N^2\log(N) + \frac{5}{2}N^2 + 2N$, which corresponded to the tilde approximation $\sim N^2\log(N)$.

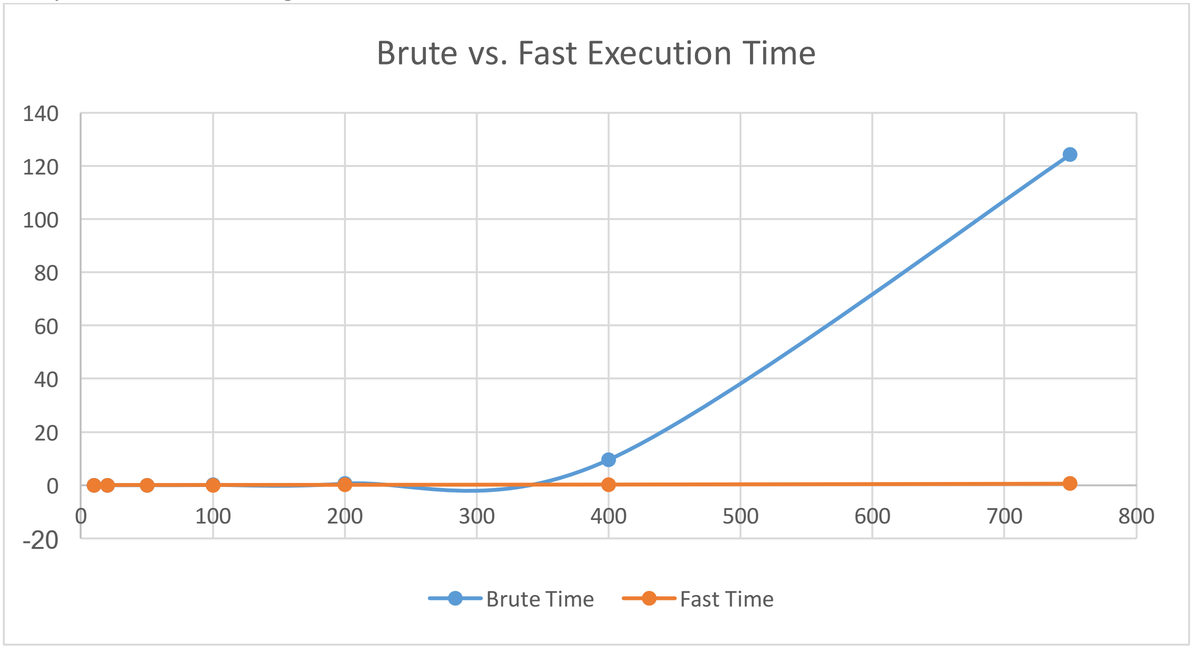
2. Time Data:

The following table includes raw times obtained from running each program under various values for N:

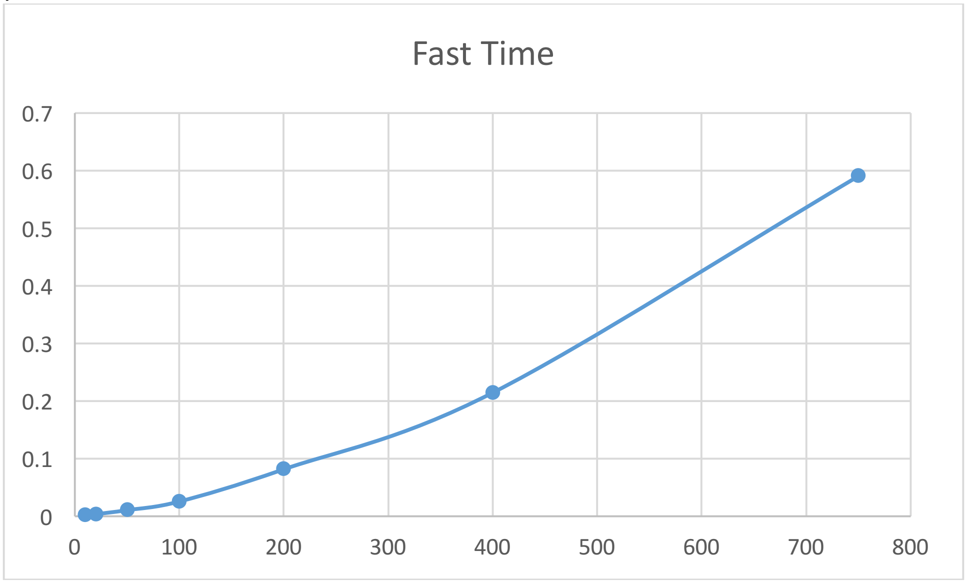
N	Brute-Force Time (seconds)	Fast Time (seconds)
10	0.001	0.002
20	0.004	0.004
50	0.017	0.011
100	0.085	0.026
200	0.612	0.082
400	9.577	0.215
750	124.233	0.591

When N reached 1000, the runtime for Brute Force surpassed 200 seconds, so its data was not measured in the analysis.

Plot of the above data, in which the Brute Times are plotted with blue, and the fast times are plotted with orange, is as follows:



The above plot does not efficiently represent the curvature of the fast time line, and it is presented below:



While it is normally anticipated that the fast time curve would be logarithmic time, it is altered slightly because it is multiplied with an N^3term.

3. Time estimation for N = 1,000,000:

In order to adjust for conversion between raw complexity and time, a line of best fit was calculated based off of the N and time data recorded for the Brute-Force Execution (using *Mathematica*). The quartic polynomial estimate for T(N), where the time (seconds) of execution is a function of N, is as follows:

$$T(N) = 4.22225 * 10^{-10}N^4 \text{-} 2.2806 * 10^{-8}N^3 \text{-} 2.96933 * 10^{-7}N^2 + 0.000711038\, N \text{-} 0.00948691$$

By this, an N=1,000,000 input would take:
$T(1{,}000{,}000) = 4.22225 * 10^{-10}(1{,}000{,}000)^4 \text{-} 2.2806 * 10^{-8}(1{,}000{,}000)^3 \text{-} 2.96933 * 10^{-7}(1{,}000{,}000)^2 + 0.000711038\,(1{,}000{,}000) \text{-} 0.00948691 = 4.22202 * 10^{14} seconds$

For fast, the same was done, but by using a logarithmic form. The following function T(N) gives the time in seconds the program with execute with N points as input:

$$T(N) = 1.00507 * 10^{-6}N^2\log(N) + 0.000422619\, N\, Log(N) + \text{-}7.39313 * 10^{-6}N^2 \text{-} 0.00146668\, N + 0.00862049$$

By this, an N= 1,000,000 input would take:

$$T(1{,}000{,}000) = 6.49678 * 10^6\ seconds$$

Although both programs would take a significant amount of time to execute, the fast implementation is still extremely more efficient in comparison to the Brute-Force implementation, as expected.