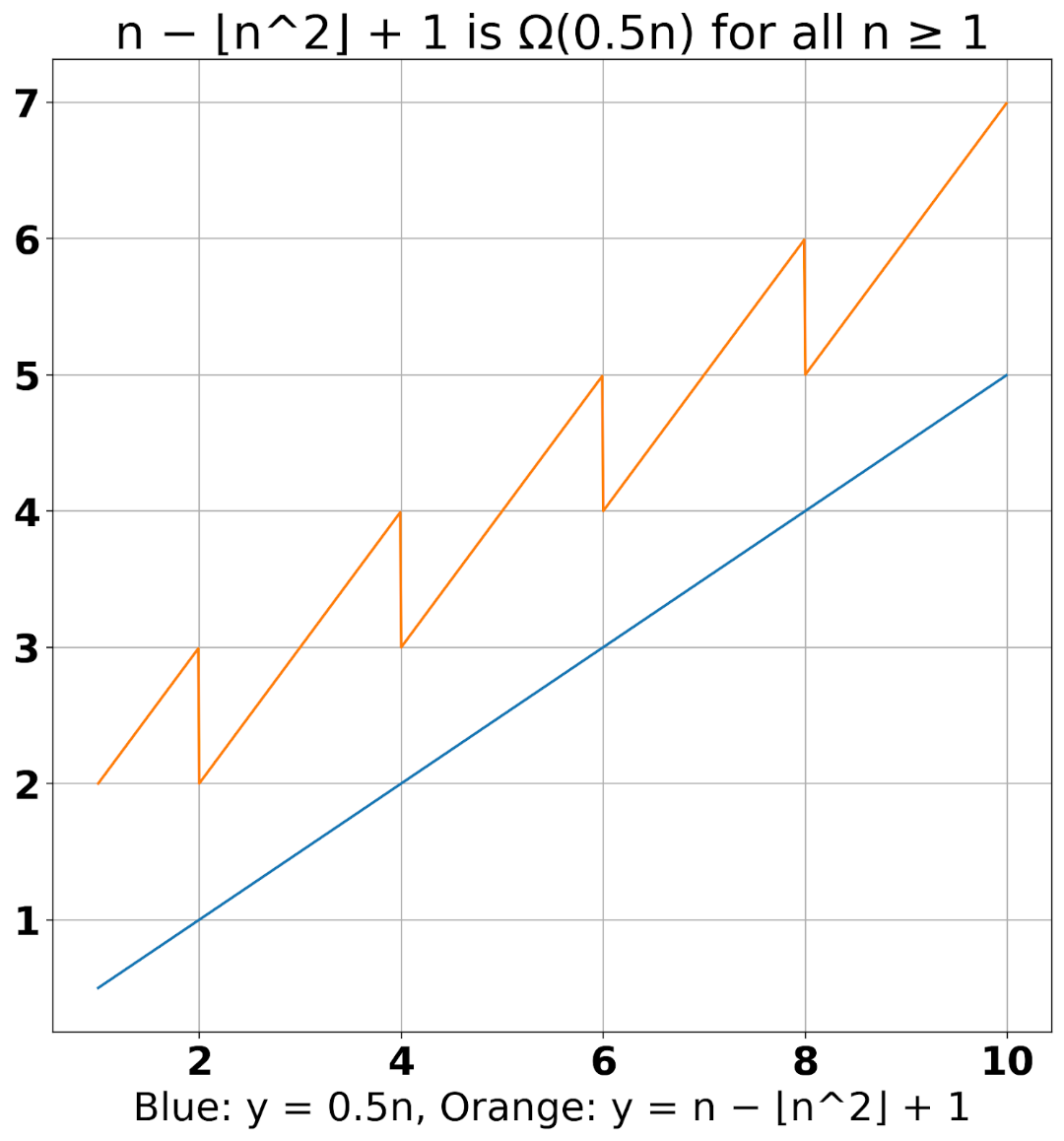
Project 2 – Technical Report

Math 4328-W01: Discrete Mathematics II

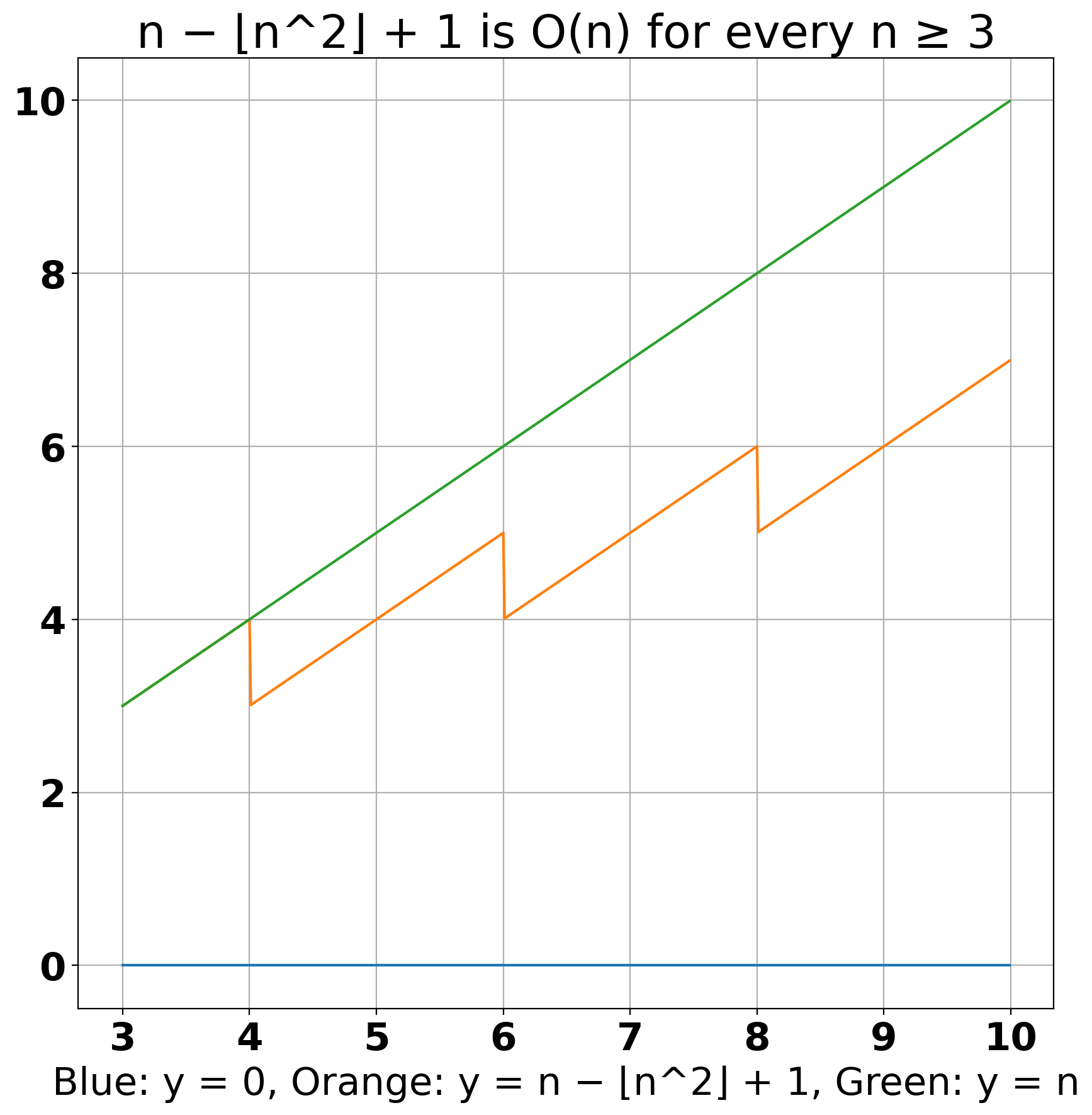
Team: Abrar Ahmad, Aidan Becker

**Analysis Part 1**: For this task, we were assigned to utilize Python (specifically Spyder) to draw a graph for each of the following functions on both sides of the inequalities on the domain of n = 1, …, 10. Then, with the context of the order notations (Big Omega, O, and Theta), we had to explain why parts b-e are different than part a, where n ≥ 1.

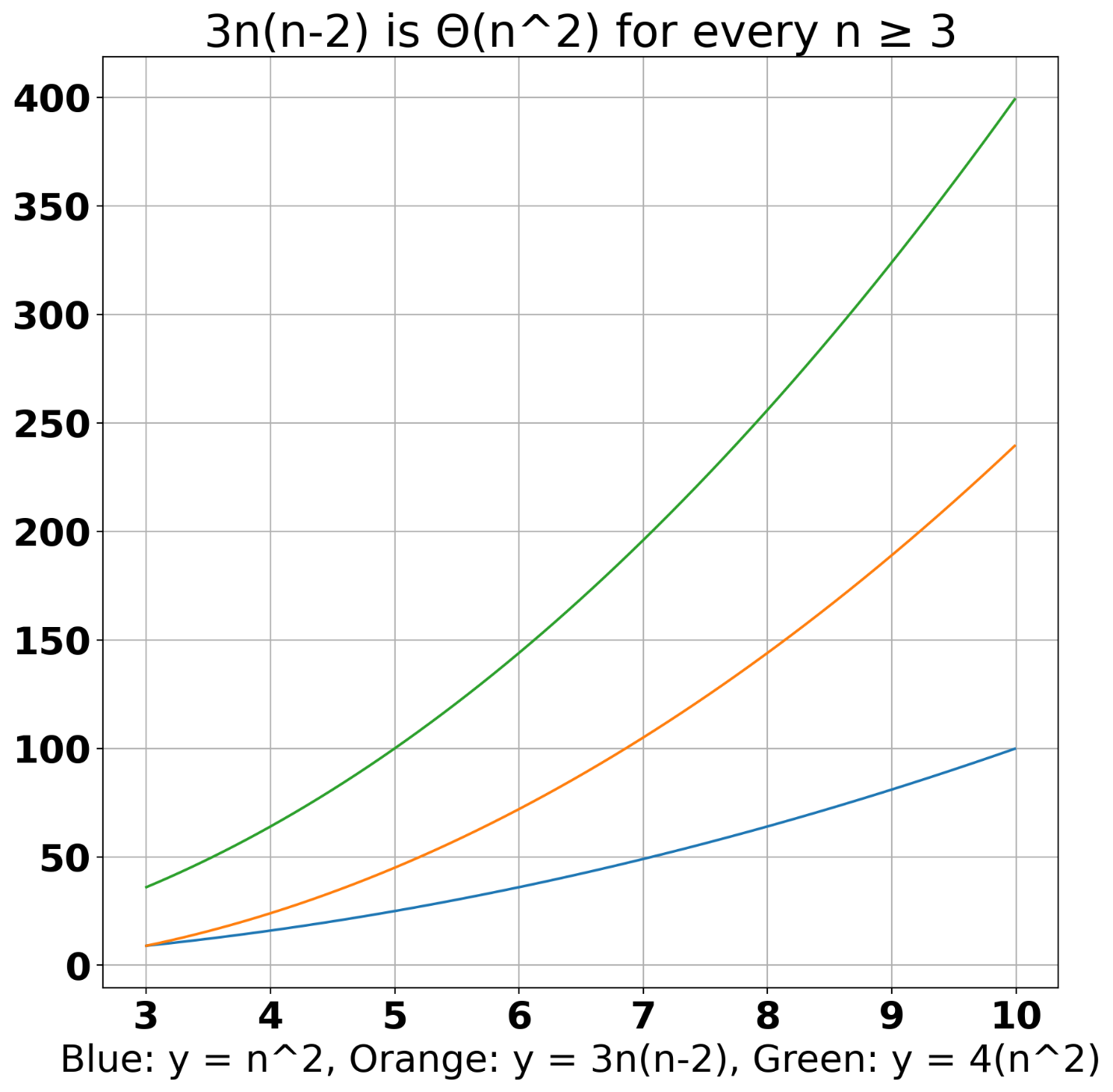
**Part 1 A:** The following graphs are shown for using Ω-notation:



**Part 1 B:** The following graphs are shown for using O-notation:

In the context order notions, the domain of n in part B is different than the domain of n in part A. This is because our functions need to fulfill the inequality 0 ≤ f(n) ≤ Bg(n) for every integer n ≥ b. At values of n < 3, f(n) is ≥ our Bg(n). Therefore, the domains of our functions need to be limited to values of n ≥ 3 or put in context of order notations, b = 3.

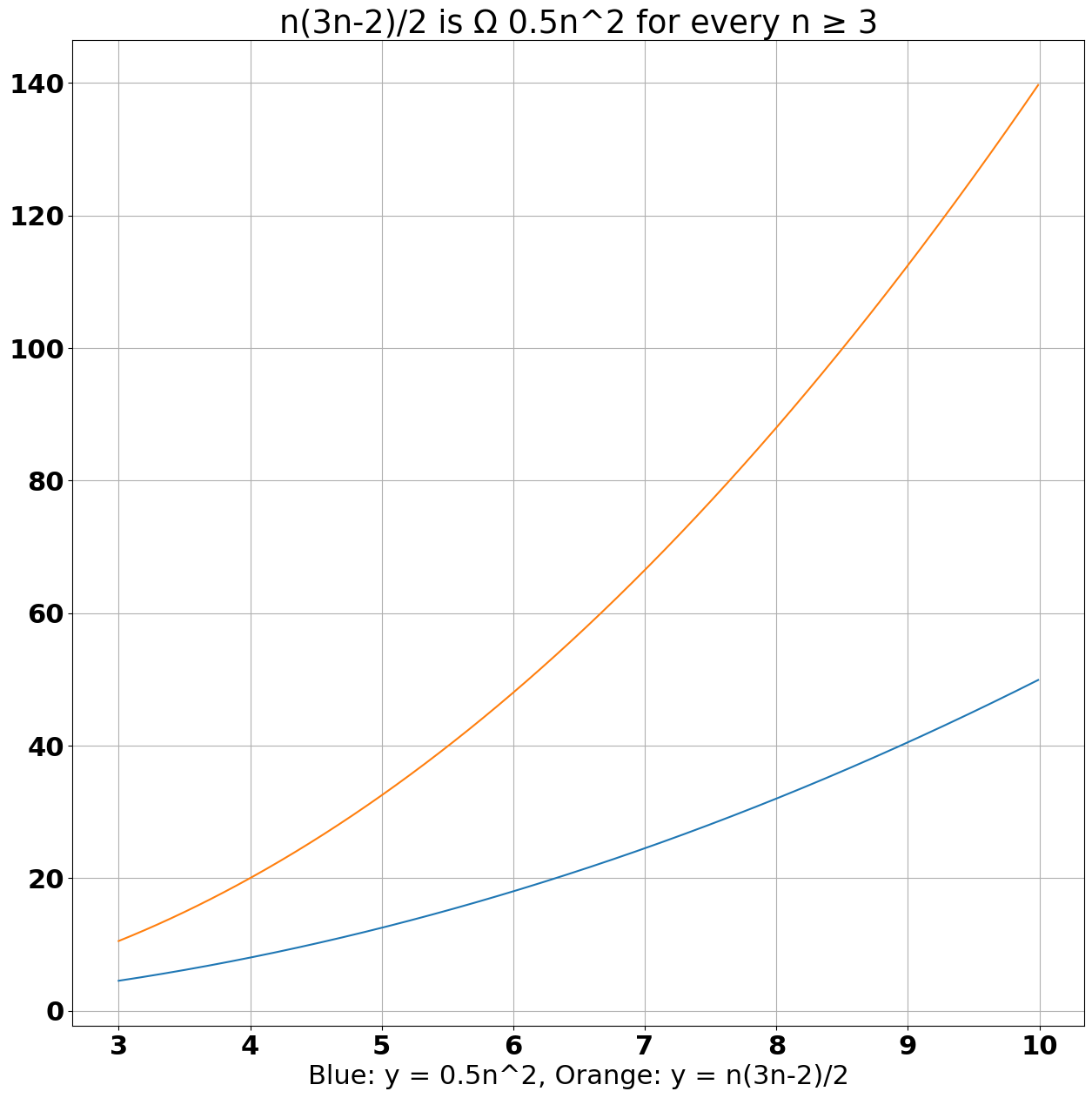
**Part 1 C:** The following graphs are shown for using Θ-notation:



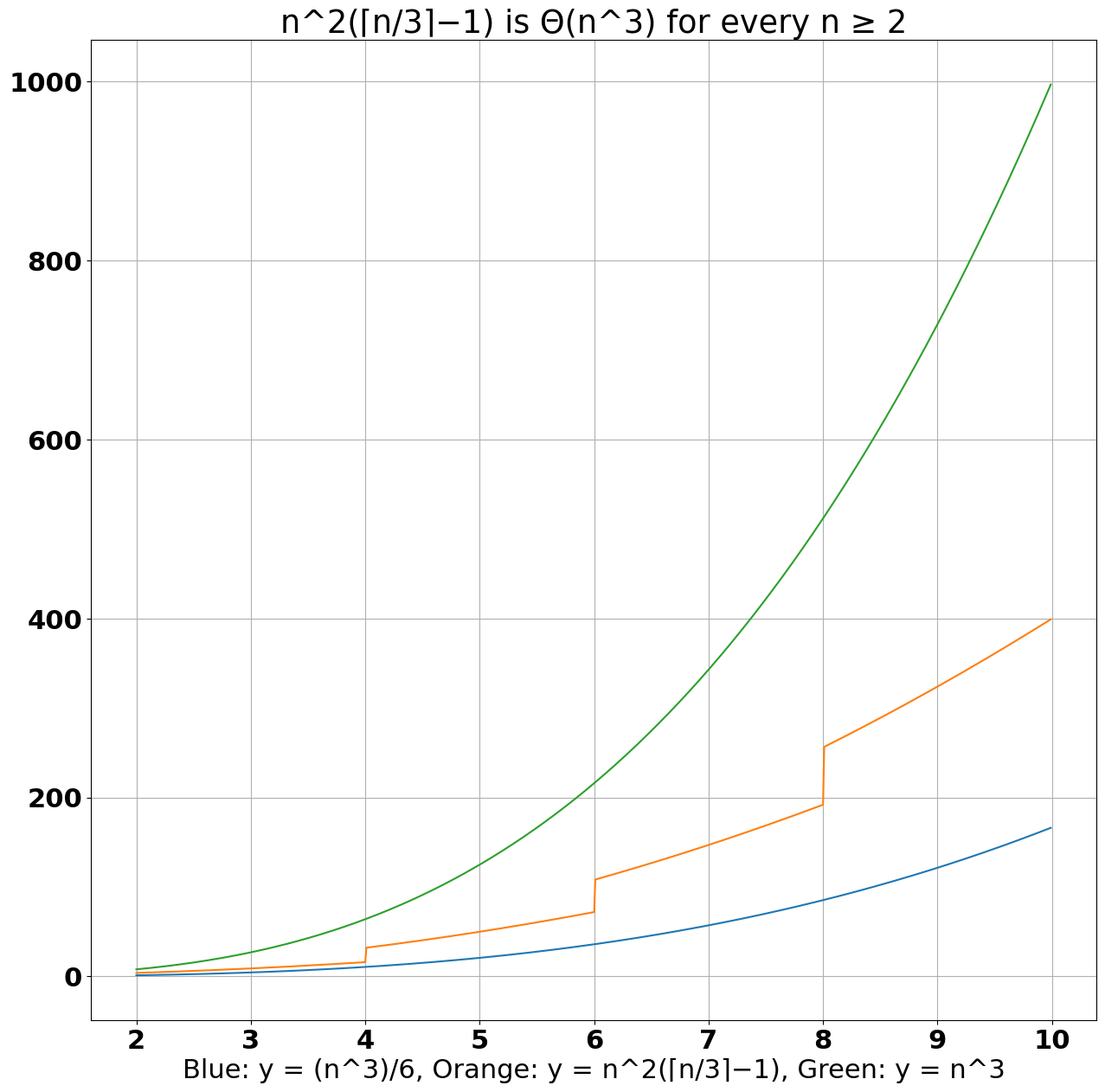
Following the definition of Θ-notation, there exist real numbers A and B such that Ag(n) ≤ f(n) ≤ Bg(n) for every integer n ≥ b. These conditions are fulfilled in this case, where g(n) = n^2, f(n) = , A = 1, and B = 4. Therefore, f(n) is Θ(g(n)).

The domain is restricted to b = 3 (or n ≥ 3) because f(n) would be less than 4\*g(n) at values of n < 3.

**Part 1 D:** The following graphs are shown for using Ω-notation:

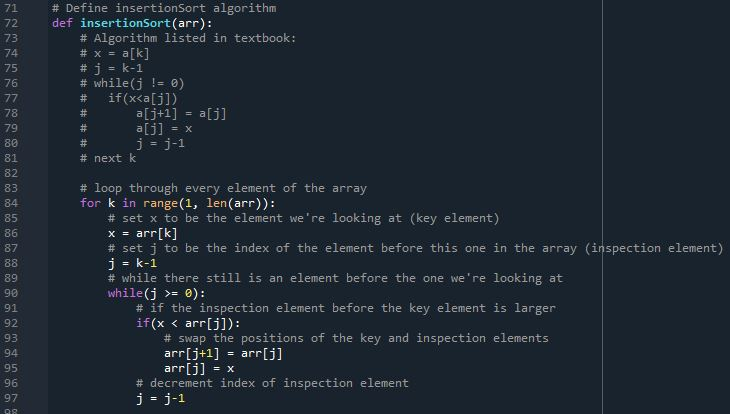
The inequality Ag(n) ≤ f(n) for every integer n ≥ a, needs to be fulfilled for big-Omega notation. In this case, a is equal to 3, because at values of n < 3, f(n) becomes > Ag(n).

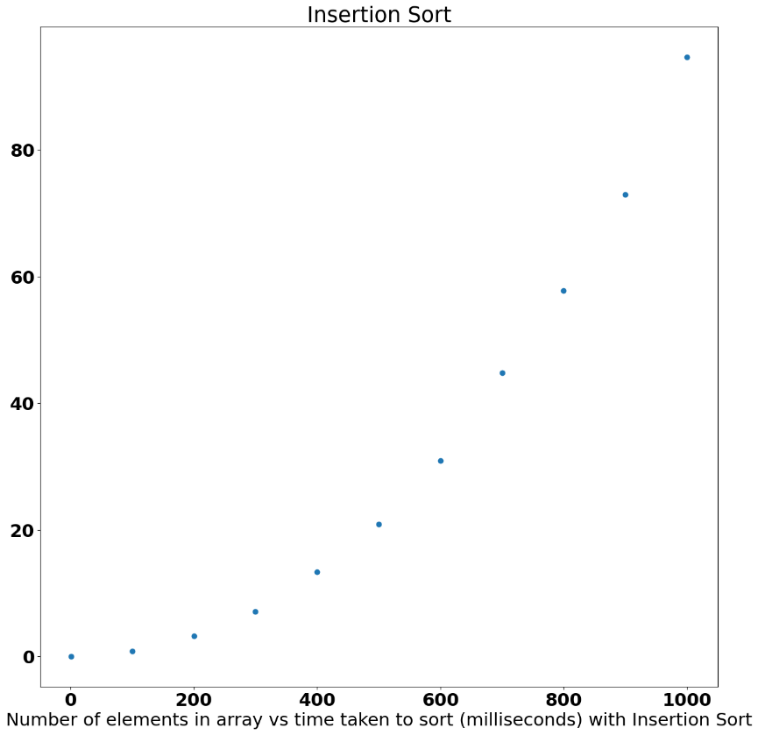
**Part 1 E:** The following graphs are shown for using Θ-notation:



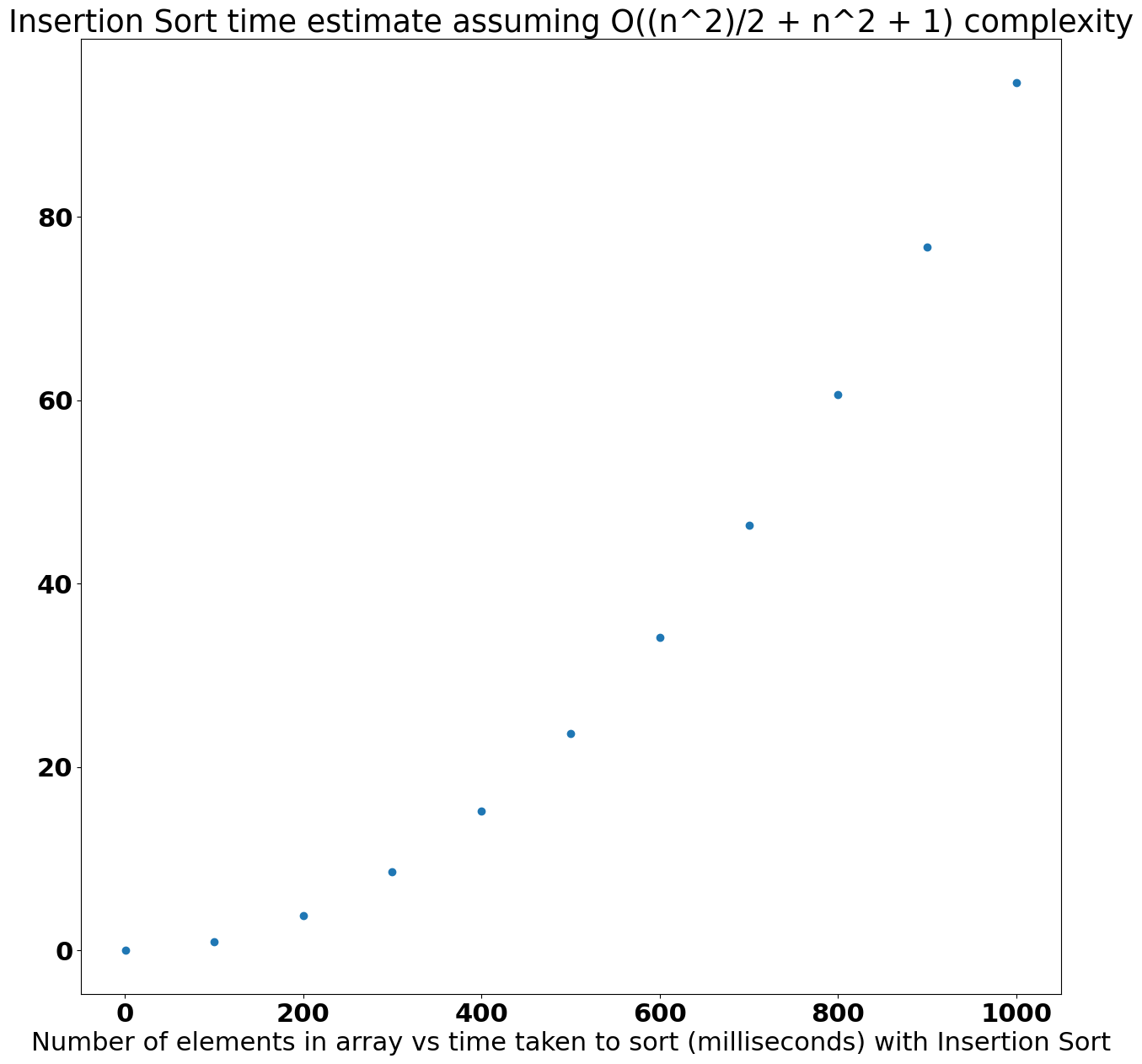
Similar to the previous part on Θ-notation, we start by assuming that g(n) = , and f(n) = . If A = , and B = 1, A\*g(n) ≤ f(n) ≤ B\*g(n) for every integer n ≥ b. Therefore, f(n) is Θ(g(n)). In this case, the domain is restricted by b = 3 (or n ≥ 3) because the inequality A\*g(n) ≤ f(n) ≤ B\*g(n) would no longer be fulfilled.

**Analysis Part 2**: For this part, we had to code Insertion Sort in Python for any given array of size n, which had to sort into ascending order. The code snippet for the algorithm is shown below:

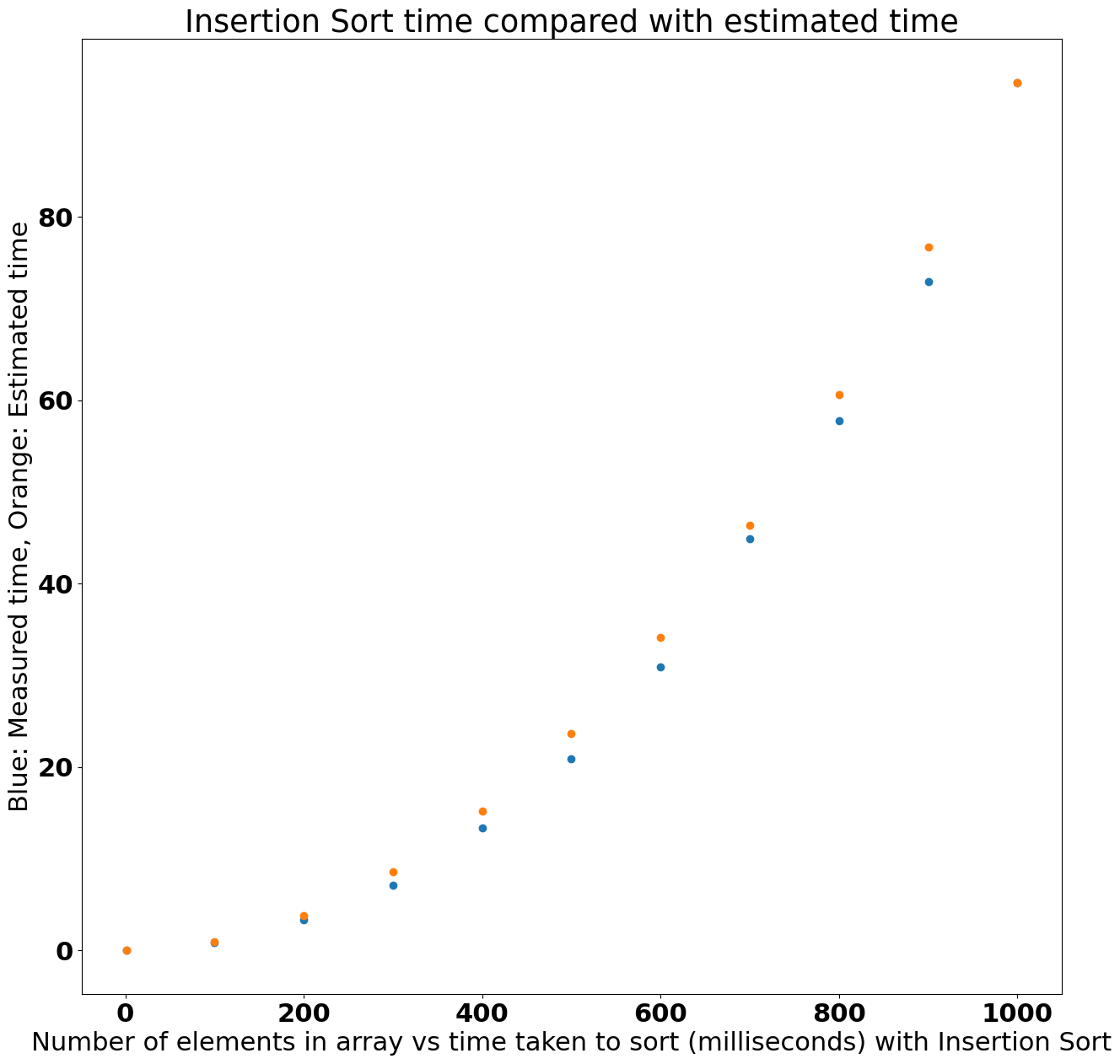
**Part C**: We had to analyze the computational time of the Insertion Sort algorithm in the worst-case scenario. The scenario is that the algorithm must sort a given array of size n that is in descending order to ascending order. For each different size n, where n=1, 100, 200, ..., 1000, we had to generate 11 different arrays in descending order and apply the Insertion Sort algorithm to them. We compute the time using Python’s function . The results are shown below:

****

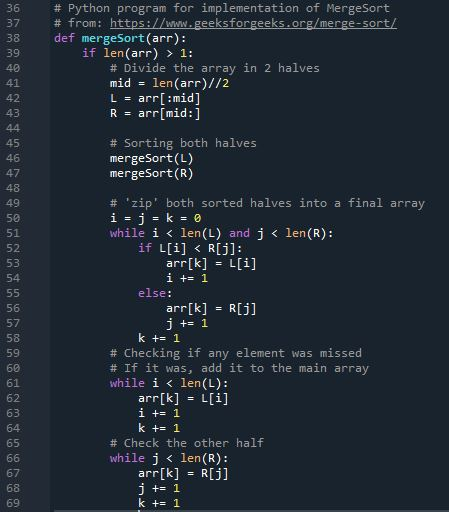
**Part D**: We are given the following formula: , where is the number of comparisons. The whole formula is the calculation of the maximum number of comparisons in the worst-case scenario. We had to graph a plot of this equation and contrast it with n=1, 100, 200, ..., 1000.



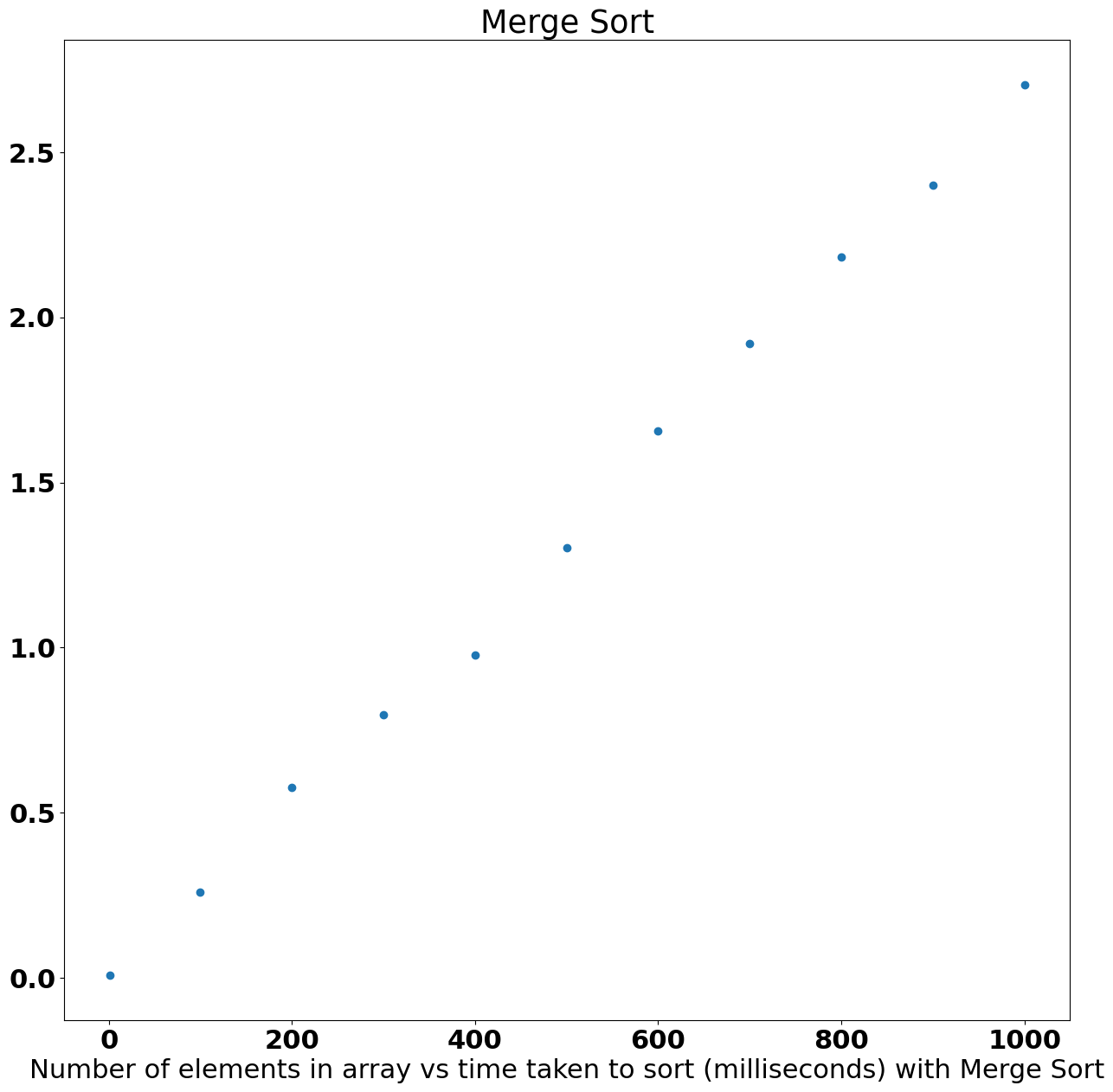
**Part E**: Here, we simply compare and contrast our results from parts 2c and 2d. Though it was not necessary, we thought it might be interesting to show our estimated time on the same chart as the measured time, so we scaled all estimates by a factor of (measured time for 1000 elements)/(estimated time for 1000 elements). Both charts should overlap almost perfectly at both n = 1000.



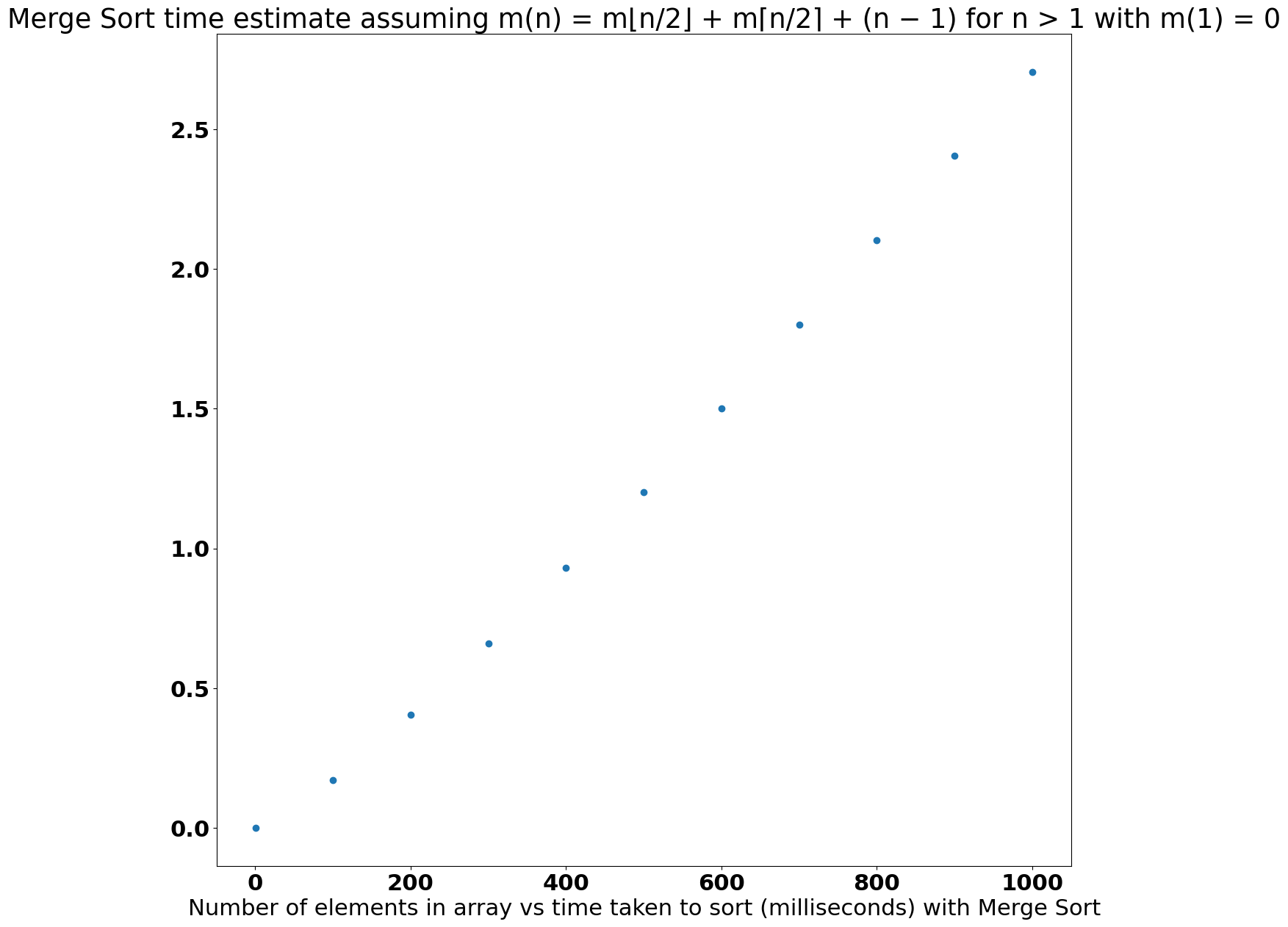
We can see the expected outcome, which is that our measured results accurately follow our estimation curve and were bounded above by our estimated time. In O-notation, we can say that Insertion Sort is O().

**Analysis Part 3**: For this part of the project, we do exactly what we did in Part 2, except we do it for Merge Sort. The following is a code snippet we used for Merge Sort

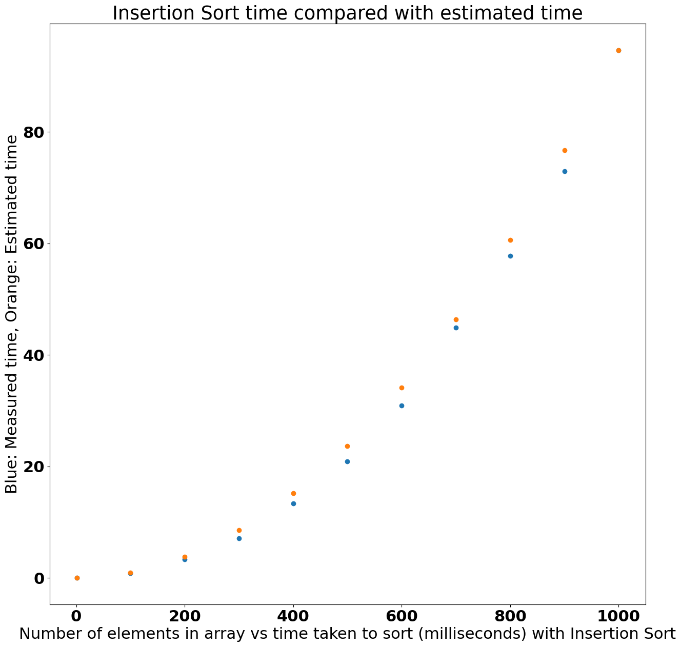
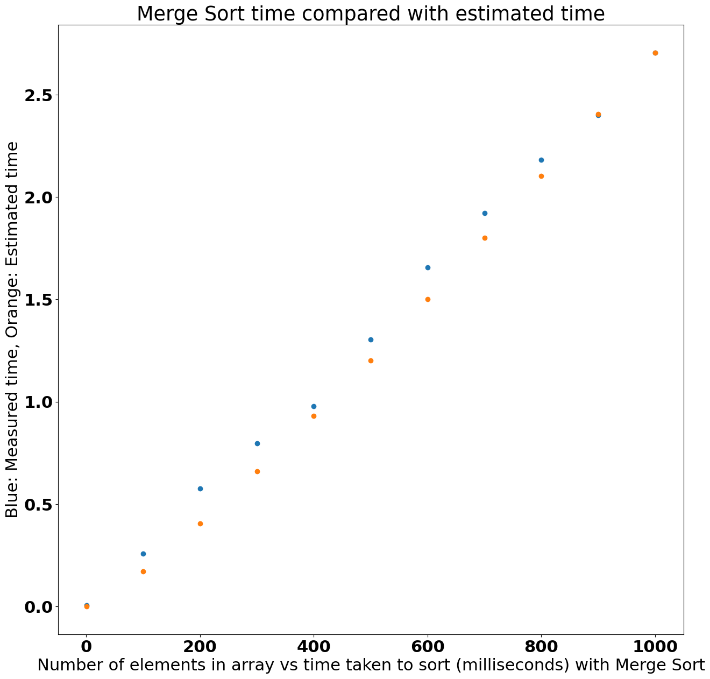
**Part C**: We analyze the computational time for the Merge Sort algorithm, placing it in the worst-case scenario, where it must sort from descending order to ascending. Just like in Part 2, we had to generate 11 different arrays in descending order and use Python’s Time function for computing the total time spent. The following plot contains our results from timing each sorting function.



**Part D**: For the maximum number of comparisons in Merge Sort, the total can be calculated using this recursive formula: .



**Part E**: For this part, we not only compare and contrast the plots in parts 3c and 3d but also the two plots from Part 2. Estimates were scaled in the same way as they were for Insertion Sort’s estimates.

****

We see that Merge Sort follows the same general pattern as our estimation algorithm, though not nearly as closely. Over several trial runs, similar patterns were produced, where the actual code consistently took more time than the algorithm estimated. However, we can still reasonably say that Merge Sort is roughly O(.)

We can also see that Merge Sort appears to be significantly more linear in nature than Insertion Sort, which appears to be roughly exponential in nature.

**Part F**: Here, we provide our opinion on which sorting algorithm is better: Insertion Sort or Merge Sort.

The opinion is that Merge Sort is ultimately better because it’s more efficient than Insertion Sort, and that because it’s more efficient, it will perform better regardless of language.

There was a provided example from a textbook where it compared the two algorithms. It said that the Merge Sort algorithm written in Python is more efficient than the Insertion Sort algorithm written in C, telling the reader to take note of the algorithms each language used.

In practice, we can see that in the worst-case-scenario, Merge Sort is significantly faster than Insertion Sort for sufficiently large array sizes. While Insertion Sort took almost 100 milliseconds to sort an array of 1000 elements (n=1000), Merge Sort only took 2.75 milliseconds to sort the same array. Even at n=1, Insertion Sort took roughly 0.014 milliseconds to sort the array, while Merge Sort took roughly 0.007 milliseconds.

Some of this overall speed increase might be due to the way that the compiler handles these implementations but given how well our estimations fit our measured performance, it is likely safe to say that Merge Sort should be faster than Insertion Sort in almost every scenario.

However, we have not performed any tests on an average-case-scenario, or best-case scenarios. Regardless, for any applications where it is possible that a worst-case-scenario might appear, Merge Sort is definitively faster, at least with these minimal Python implementations.