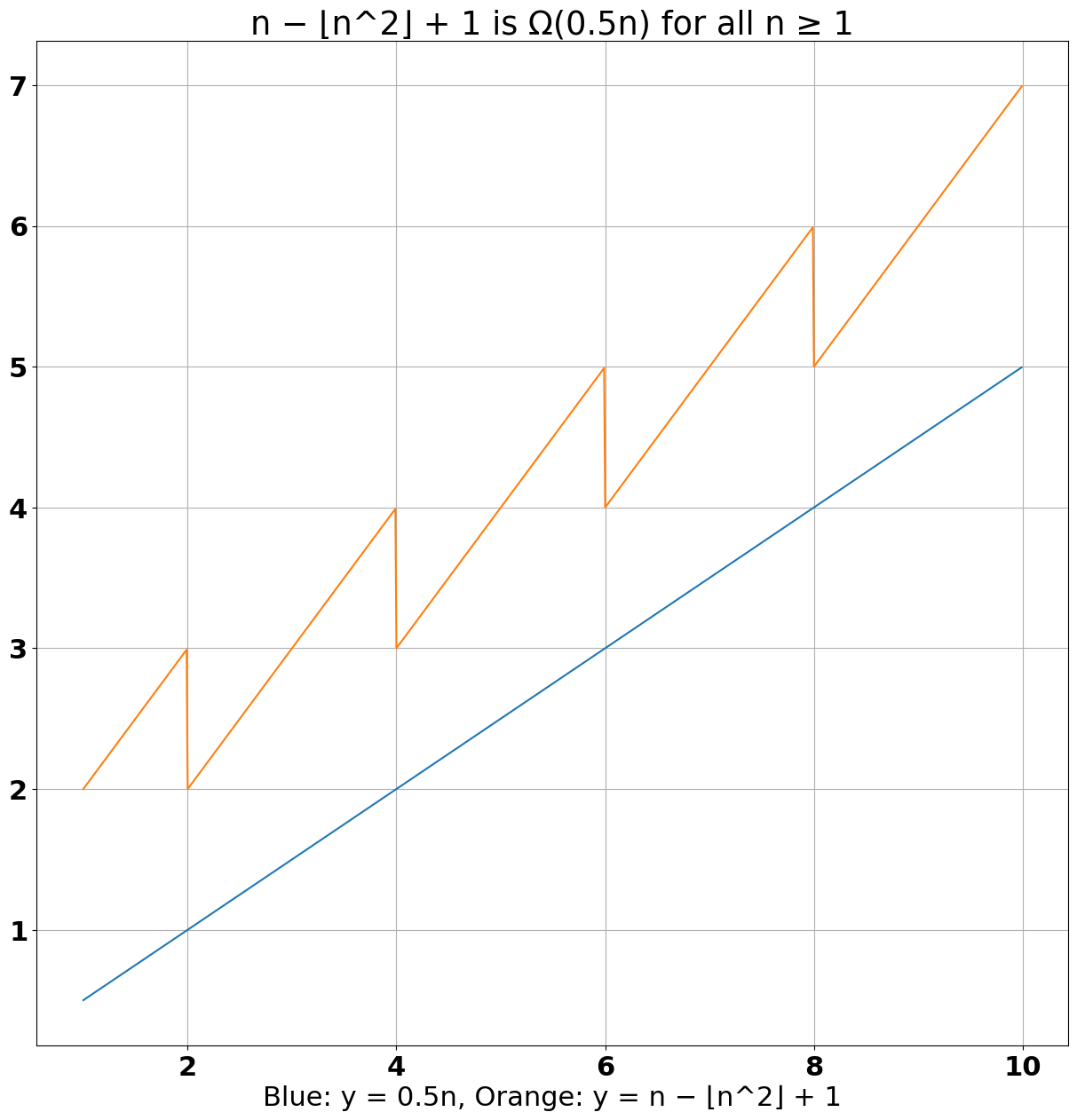
Project 2 – Technical Report

Math 4328-W01: Discrete Mathematics II

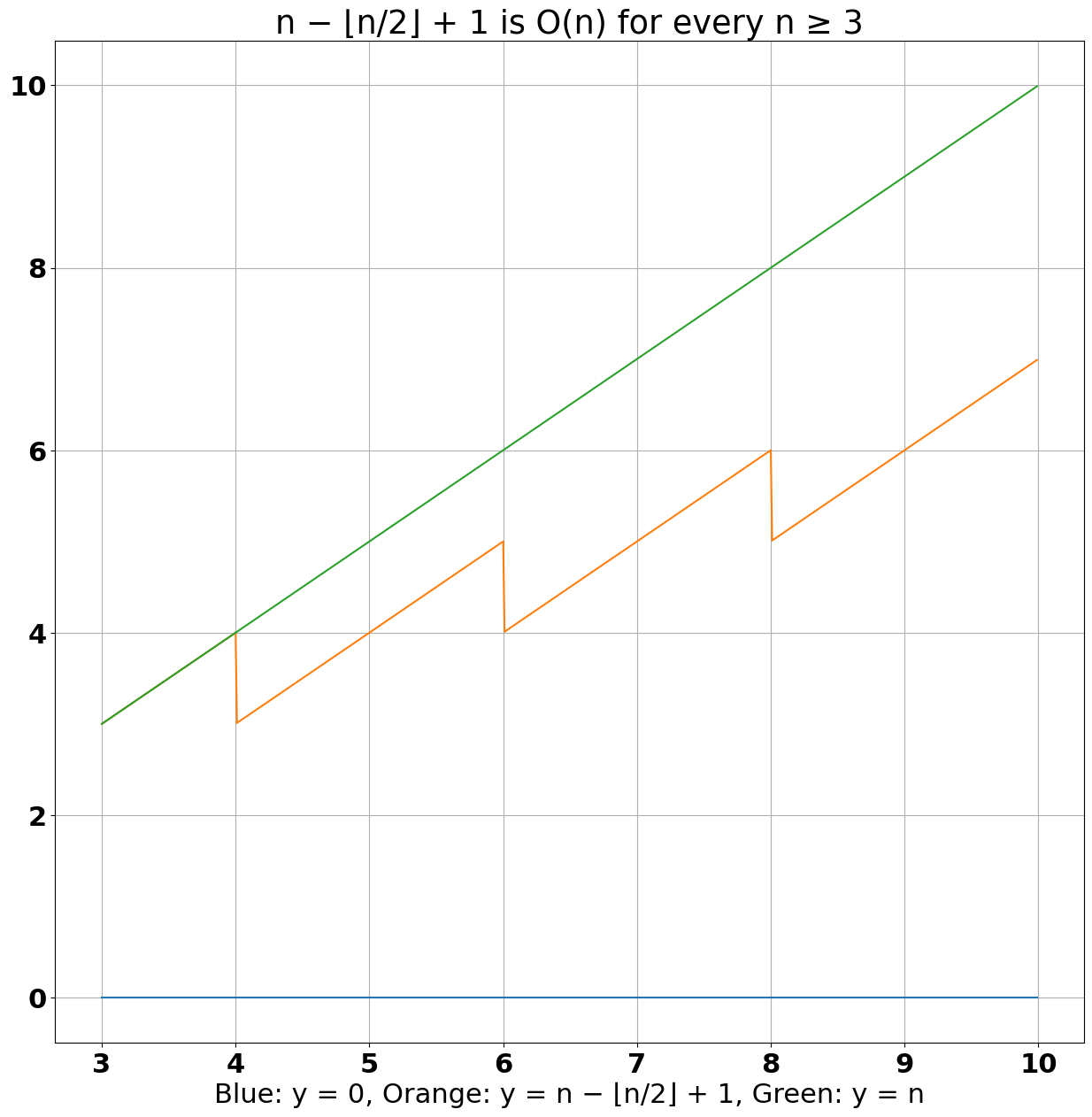
Team: Abrar Ahmad, Aidan Becker

**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 from part a, where n ≥ 1.

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

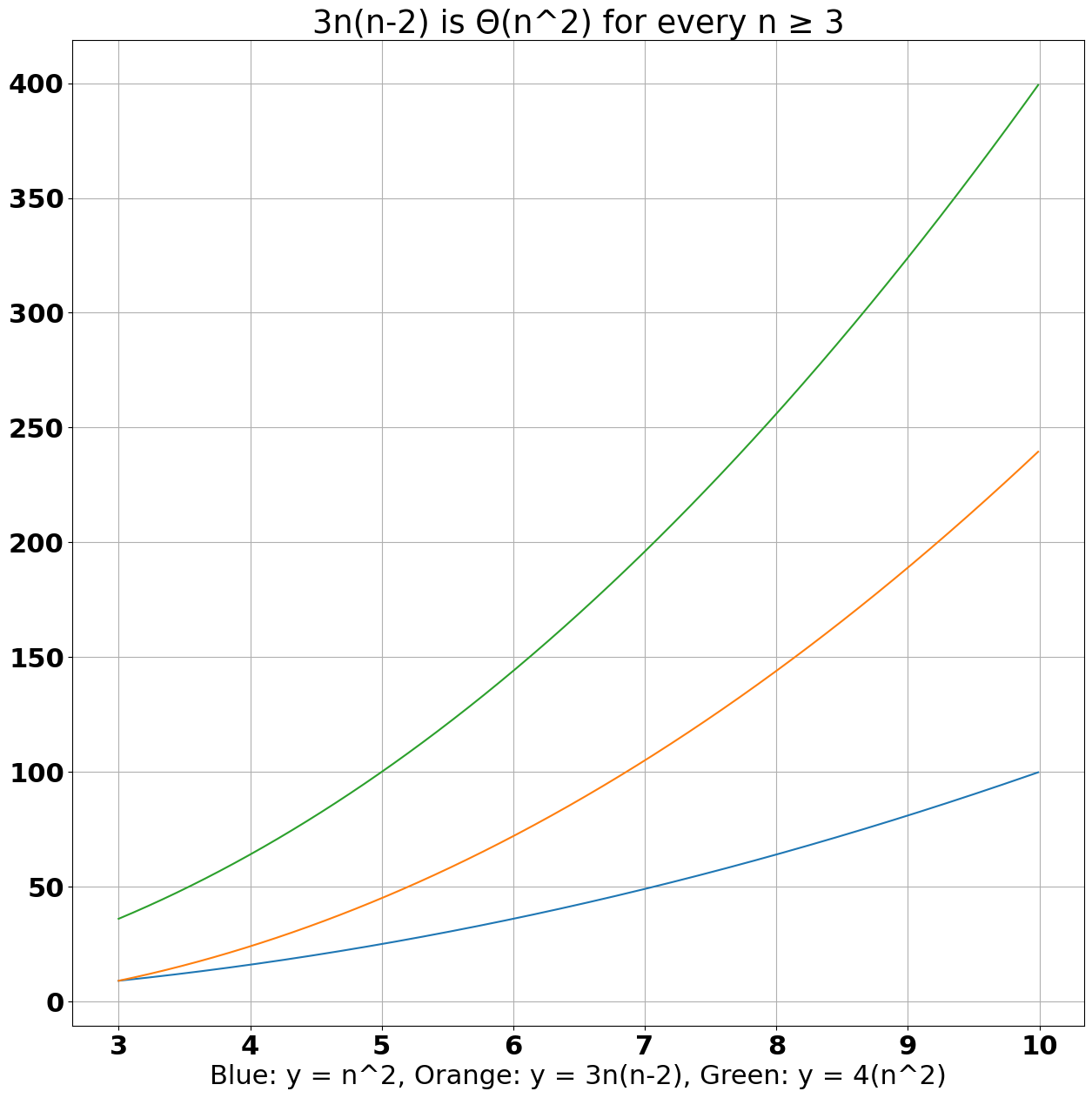


**Part 1 B:** The following graphs are shown for using O-notation:

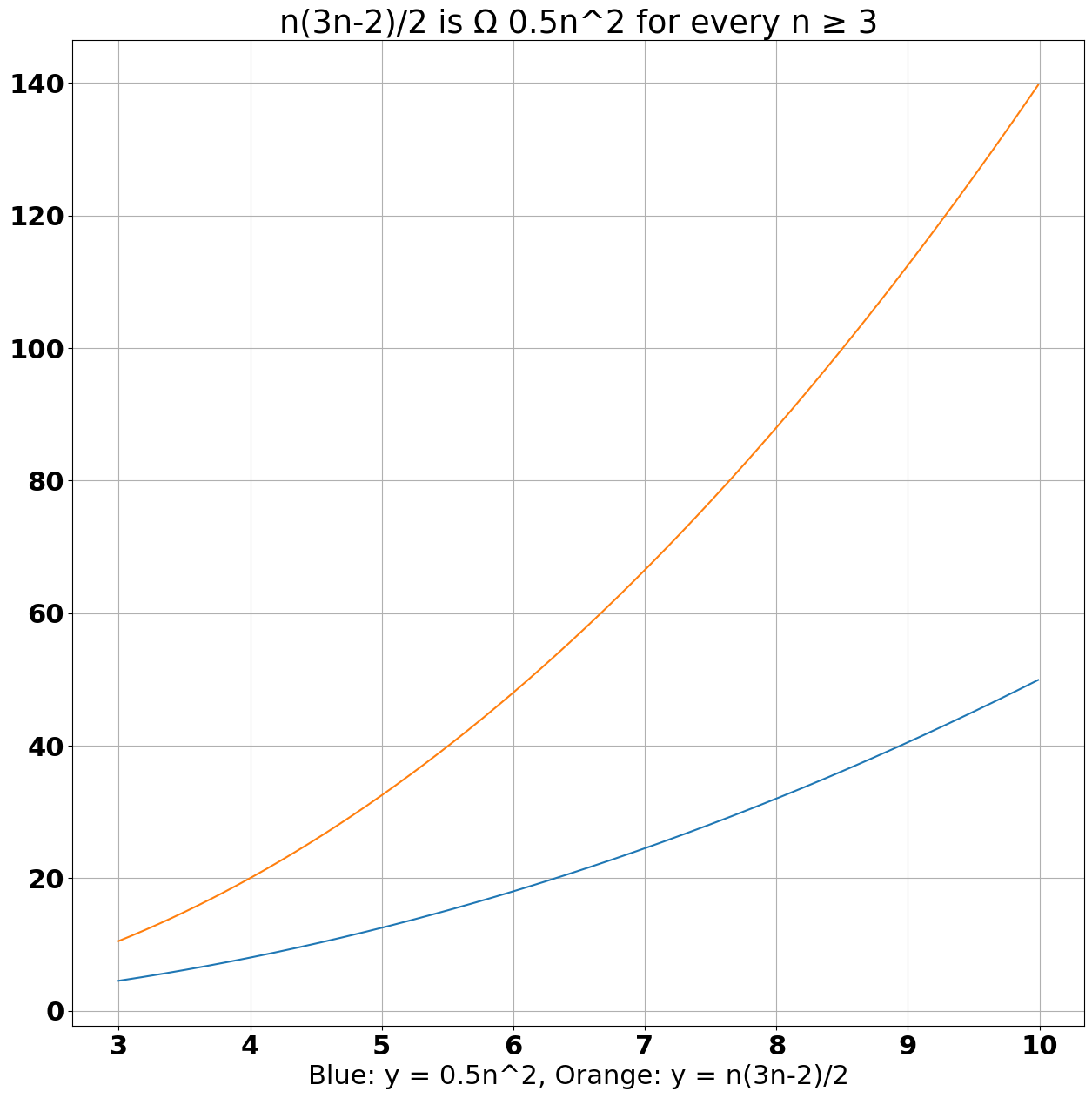


Following the definition of Big O-notation,

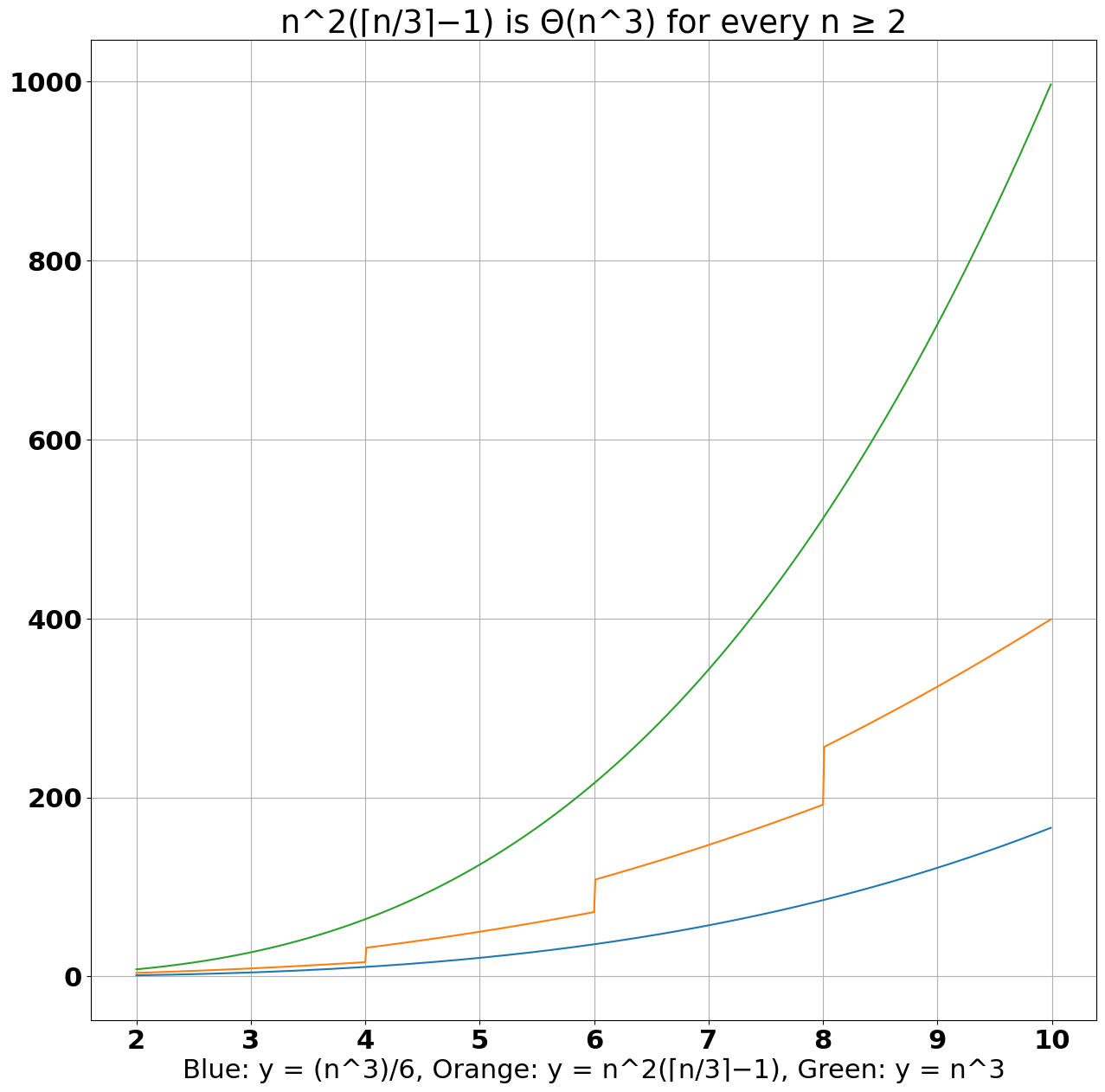
**Part 1 C:** The following graphs are shown for using Θ-notation:



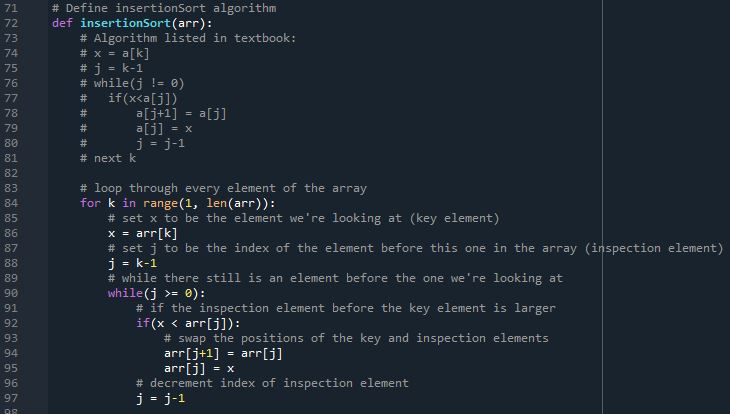
**Part 1 D:** The following graphs are shown for using Ω-notation:



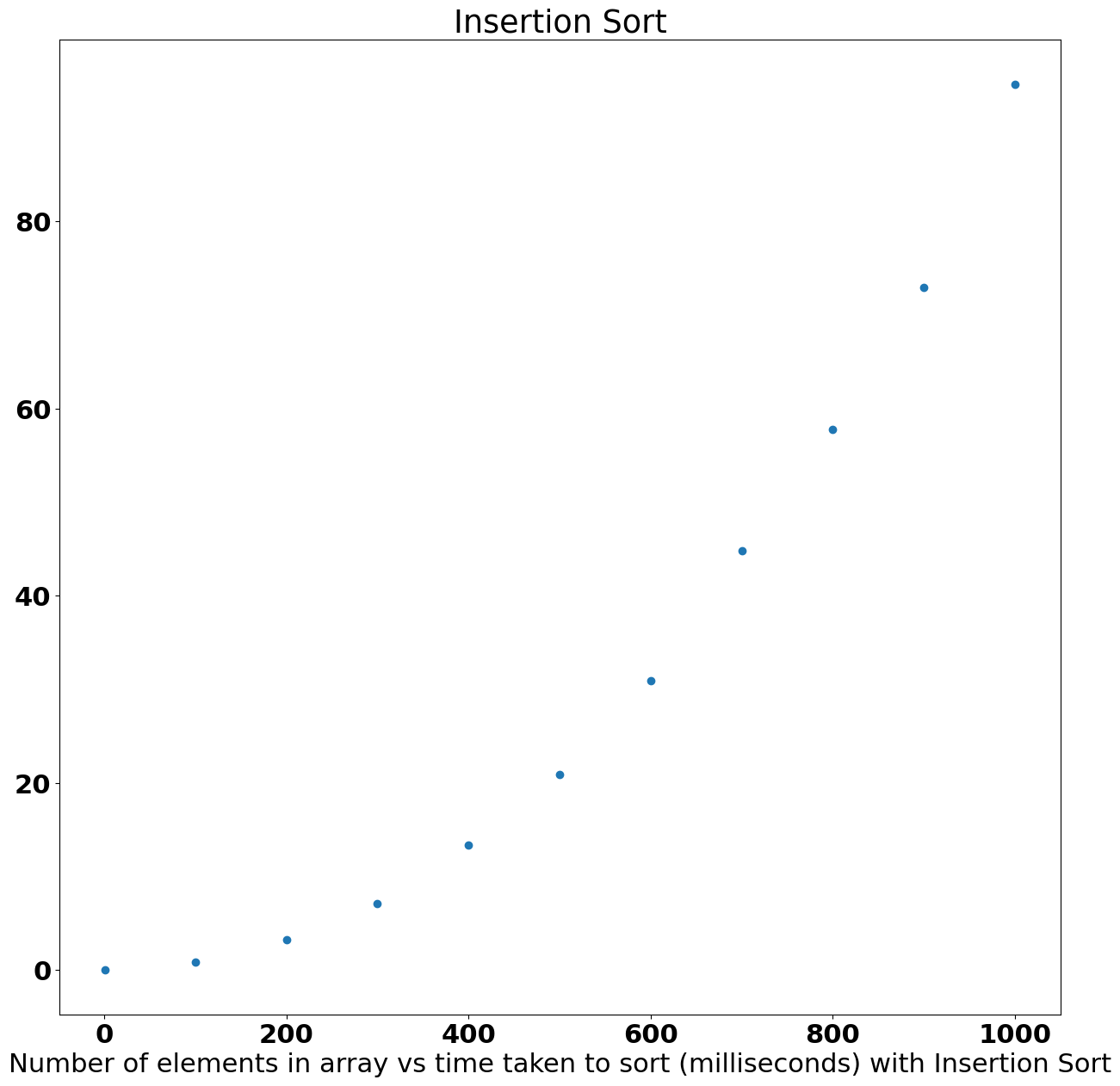
**Part 1 E:** The following graphs are shown forusing Θ-notation:



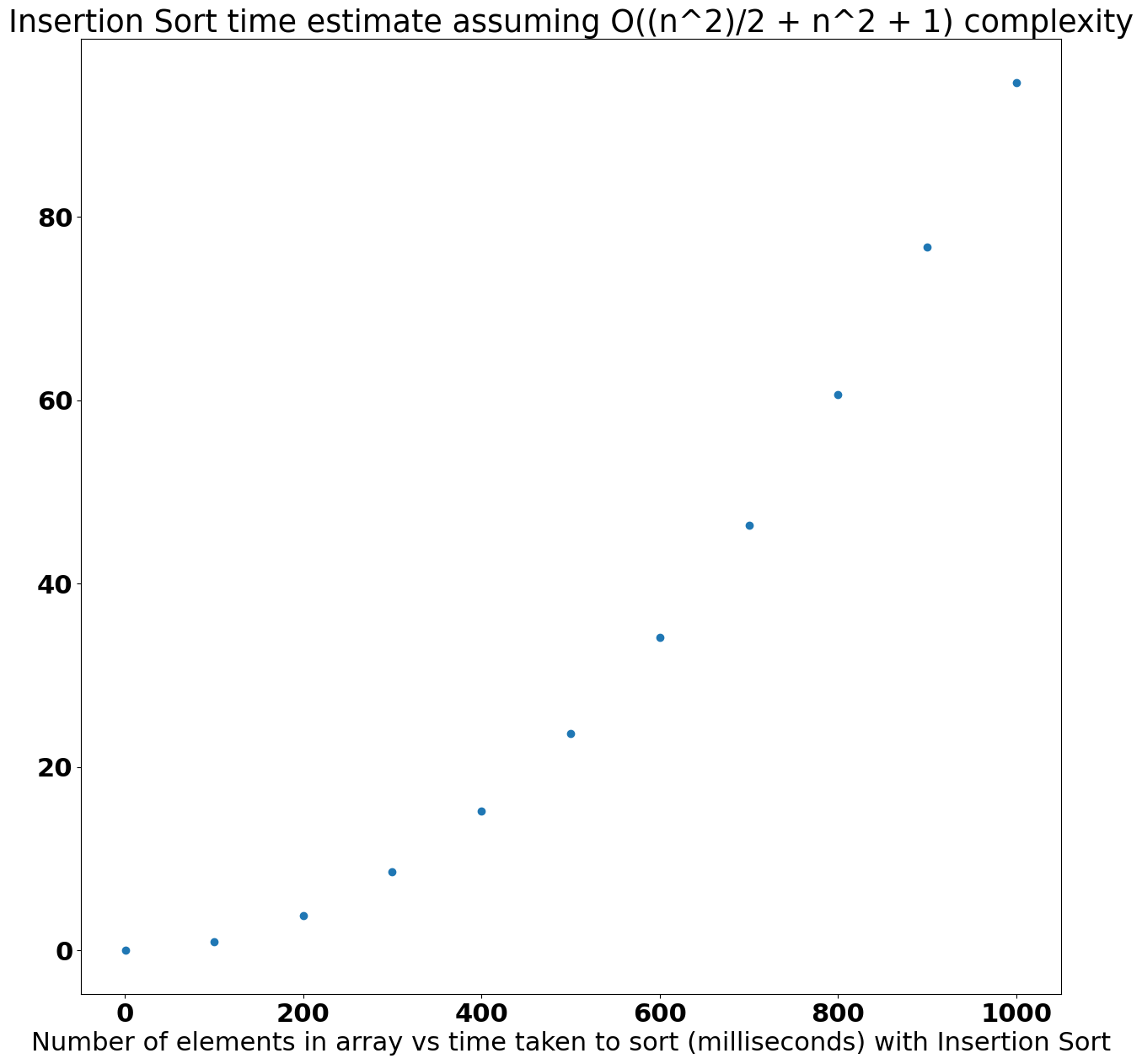
**Analysis Part 2 A and B**: For this part, we had to code Insertion Sort in Python for any given array of size n, which had to sort into ascending order. The code snippet for the algorithm is shown below:



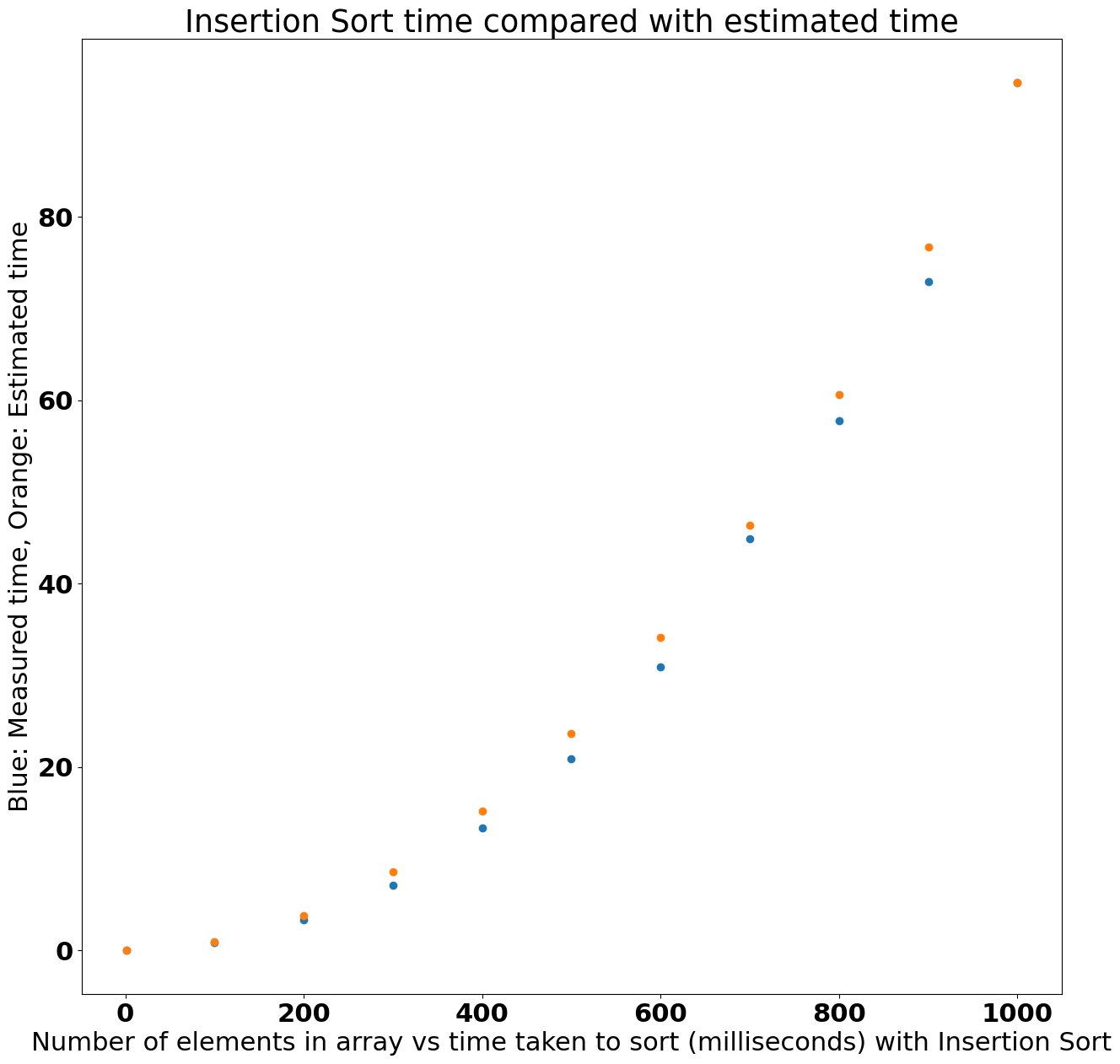
**Part 2 C:** we had to analyze the computational time of the Insertion Sort algorithm in the worst-case scenario. The scenario is that the algorithm must sort a given array of size n that is in descending order to ascending order. For each different size n, where , we had to generate 11 different arrays in descending order and apply the Insertion Sort algorithm to them. We compute the time using the timeit library’s default\_timer, because Python’s time.time() function produces varying levels of support for sub-second level accuracy depending on the system’s implementation. The results are shown below:



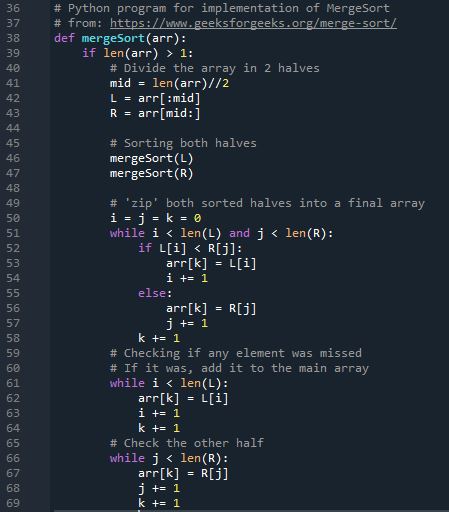
**Part 2 D**: We are given the following formula: , where is the number of comparisons. The whole formula is the calculation of the maximum number of comparisons in the worst-case scenario. We had to graph a plot of this equation and contrast it with .



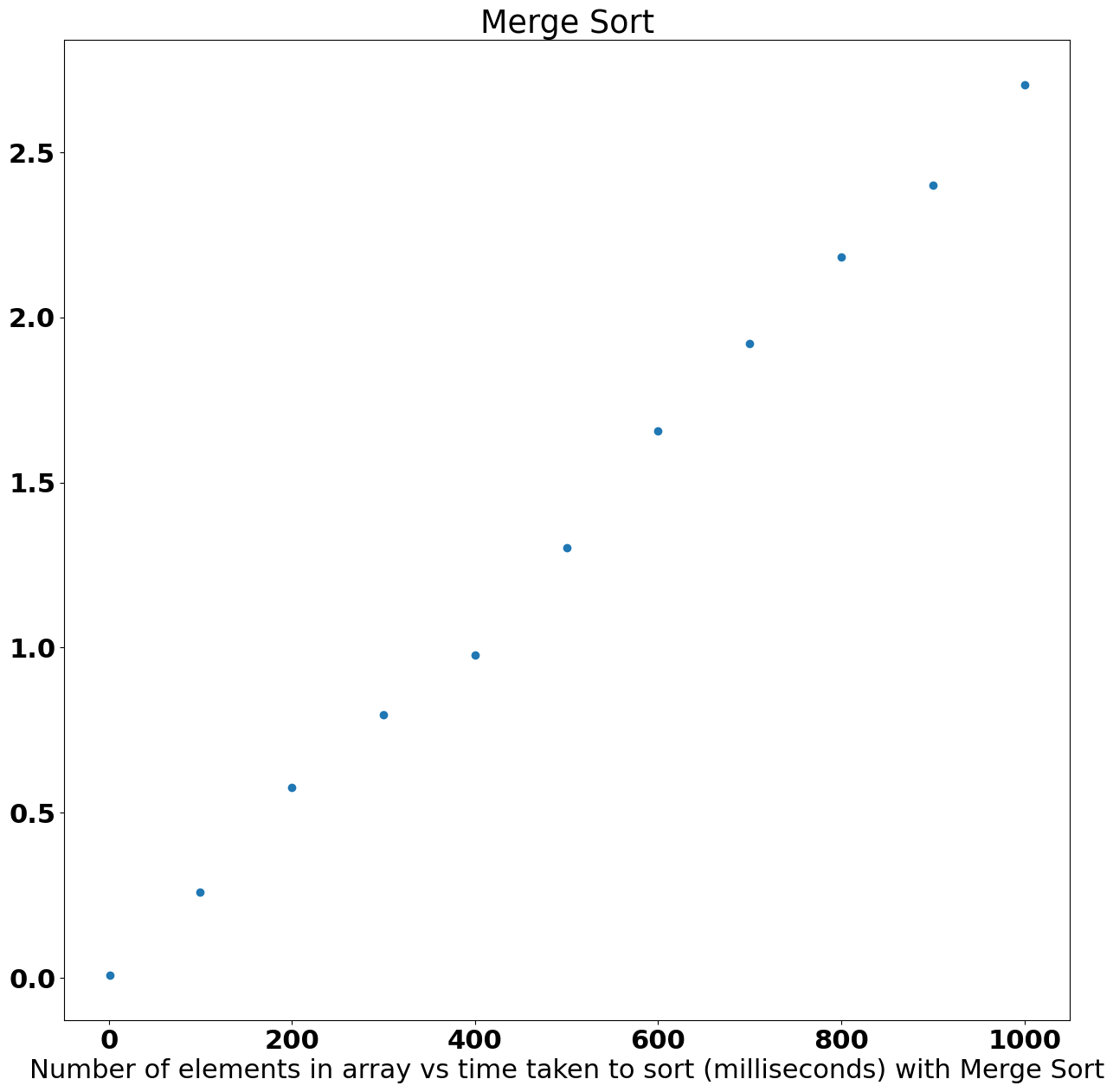
**Part 2 E**: Here, we simply compare and contrast our results from parts 2c and 2d.



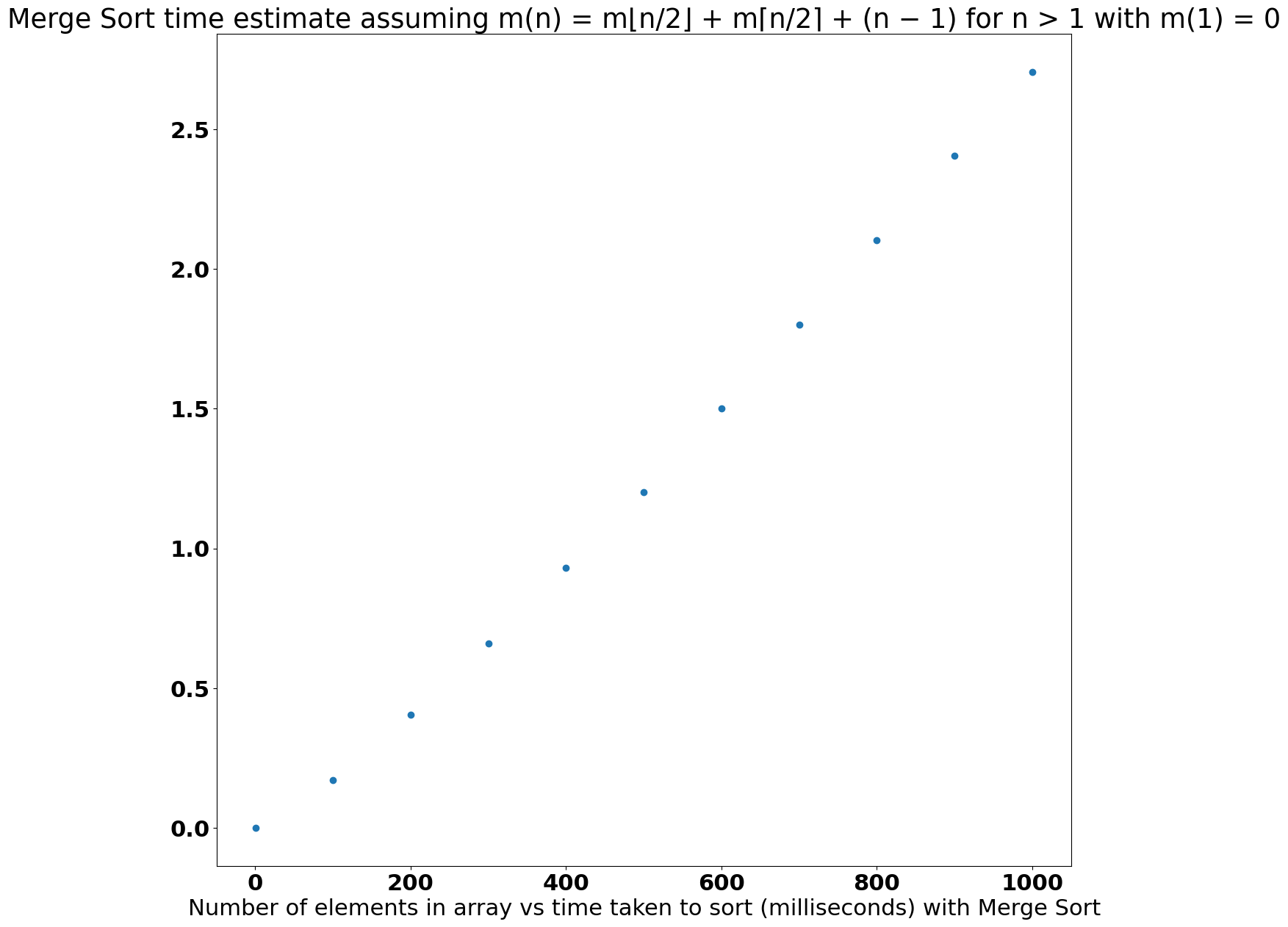
**Analysis Part 3**: For this part of the project, we do exactly all of the things we did in Part 2, except we do it for Merge Sort. The following is the code that was used for merge sort.



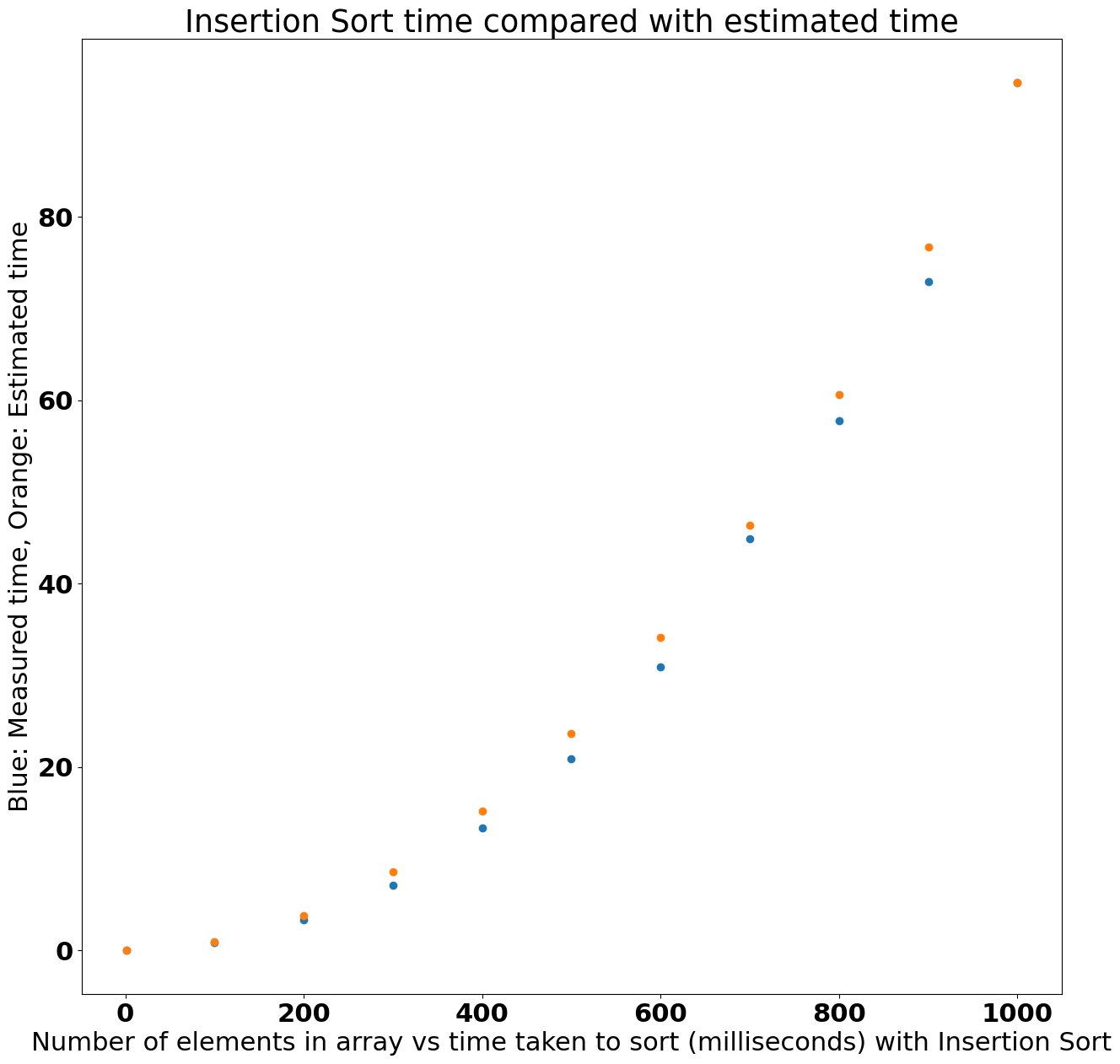
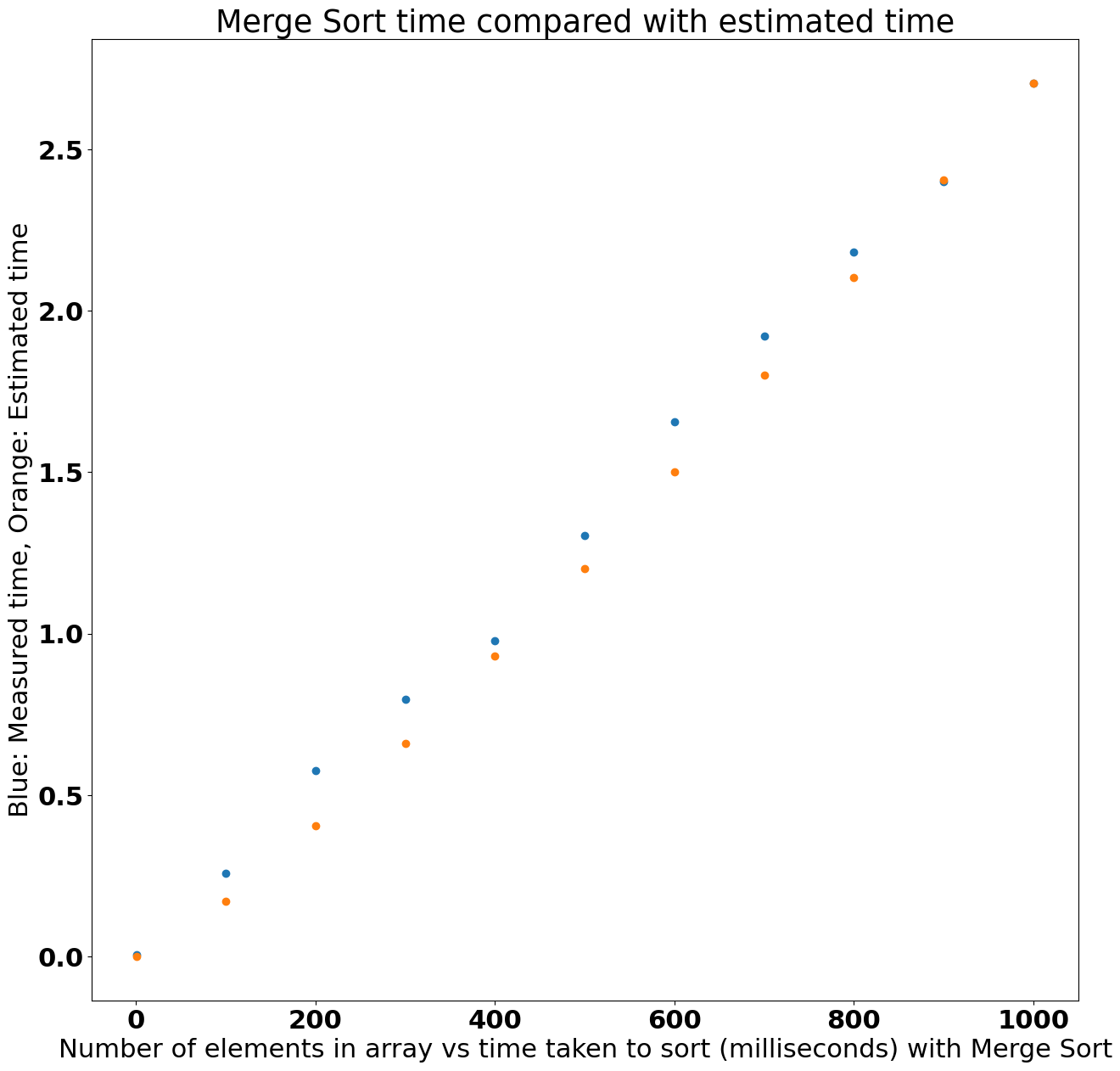
**Part 3 c**: We analyze the computational time for the Merge Sort algorithm, placing it in the worst-case scenario, where it must sort from descending order to ascending. Just like in Part 2, we had to generate 11 different arrays in descending order and use Python’s Time function for computing the total time spent.



**Part 3 d**: For the maximum number of comparisons in Merge Sort, the total can be calculated using this recursive formula: . Note that the first division uses the floor function, while the second division uses the ceiling function.



**Part 3 e**: For this part, we have to not only compare and contrast the plots in parts 3c and 3d but also the two plots from Part 2.



**Part 3 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 Merge Sort in Python is more efficient than Insertion Sort in C, telling the reader to take note of the algorithms each language used.