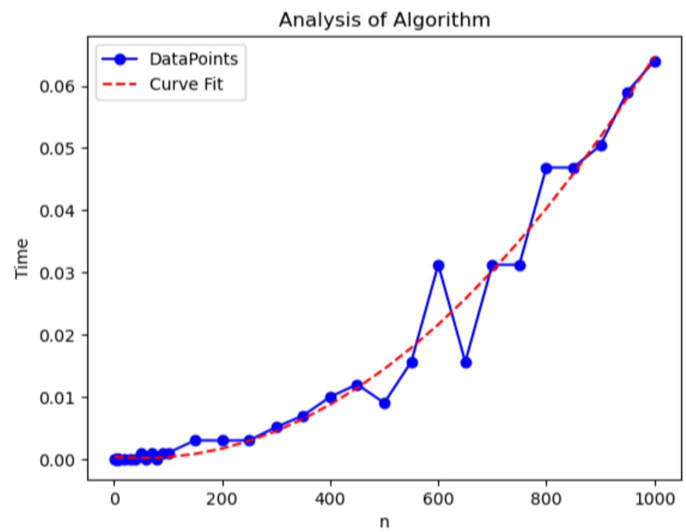
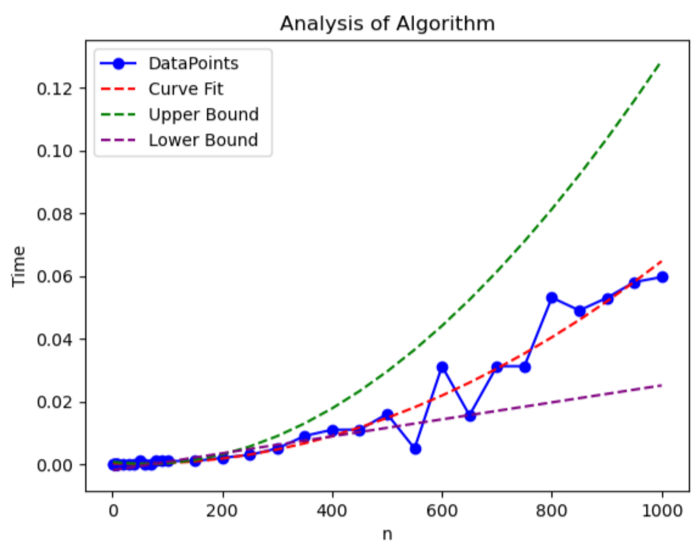
2) **Time this function for various n e.g. n = 1,2,3.... You should have small values of n all the way up to large values. Plot "time" vs "n" (time on y-axis and n on x-axis). Also, fit a curve to your data, hint it's a polynomial.**



3) **Find polynomials that are upper and lower bounds on your curve from #2. From this specify a big-O, a big-Omega, and what big-theta is.**



Upper Bound is (n^3) and Lower Bound is (n)

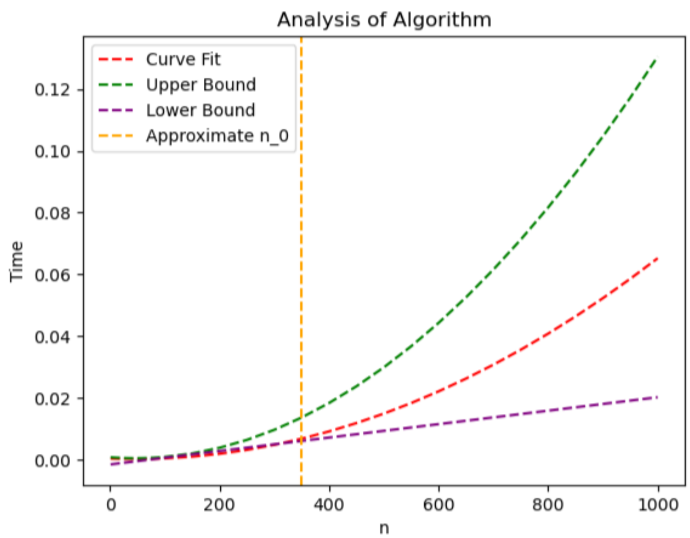
Therefore, Big-O is O(n^3) which represents worst case time complexity. (Cubic Polynomial)

Big- Omega is Ω(n) which represents best case time complexity. (Linear Polynomial)

Big-Theta is Θ(n^2) because the algorithm behavior is quadratic.

4) **Find the approximate (eye ball it) location of "n_0". Do this by zooming in on your plot and indicating on the plot where n_0 is and why you picked this value. Hint: I should see data that does not follow the trend of the polynomial you determined in #2.**

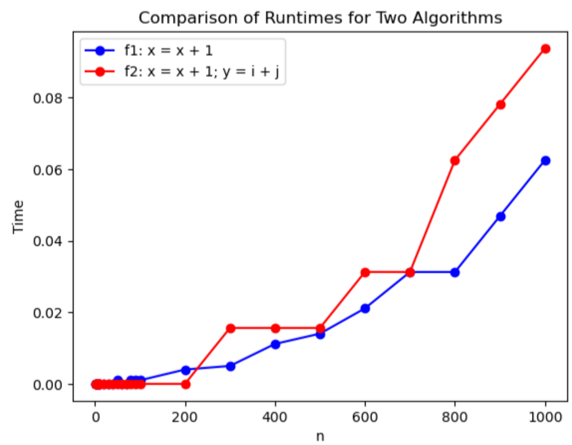
```
Approximate n_0 Point: (350, 0.006854627924722713)
```



Modified function to be:

```
x= f(n)
  x = 1;
  y = 1;
  for i = 1:n
       for j = 1:n
            x = x + 1;
             y = i + j;
```

4) Will this increate how long it takes the algorithm to run (e.x. you are timing the function like in #2)?



The runtime will increase as the modified function contains **y = i+ j** which takes more to execute compared to first algorithm where only the **x** increases. So, the runtime increase to the twice than the first one.

5) Will it effect your results from #1?

No, this will not affect the result from #1, though the steps count increases because the second one contains additional operation (y = i+j) but the runtime still will remain same as in the both the algorithm, the inner and outer loop runs n times. Hence, the time complexity will be O(n^2) for both the algorithms.