1. function x = f(n)

```
x = 1;
for i = 1:n
   for j = 1:n
      x = x + 1;
```

1. **Find the runtime of the algorithm mathematically (I should see summations).**

Sol: Given Algorithm has 2 nested loops

i. Outer loop runs from 1 to n

ii. Also inner loop runs from 1 to n

iii. Inside inner there is constant operation (x = x + 1)

For each iteration of outer loop, inner loop runs n times. So, total number of iterations of inner loop is the sum from 1 to n, which can be written as

$$1 + \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

This can be simplified as, (adding 1 with n times, the inner sum is n)

$$1+ \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = 1+ \sum_{i=1}^{n} (n * 1)$$

$$= 1+ n * \sum_{i=1}^{n} 1$$

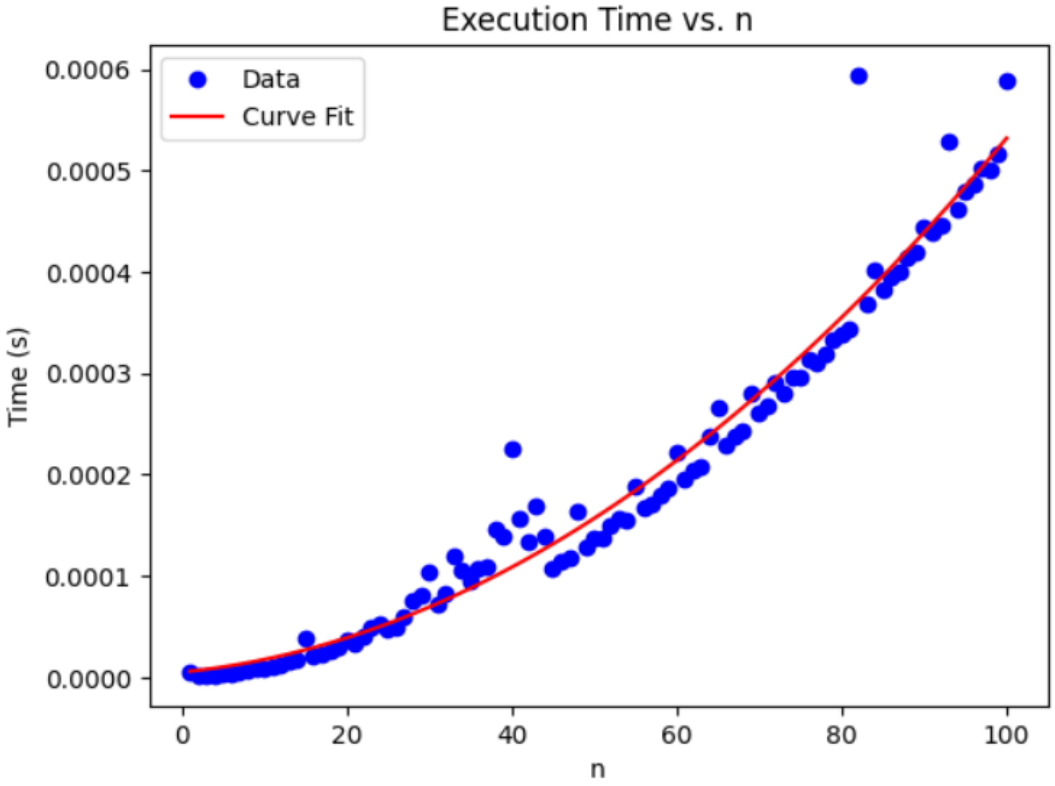
$$= 1+ n * n$$

$$= 1+ n^2$$

So, the runtime of algorithm is proportional to n^2.

Therefore, the runtime of the given algorithm is O(n^2).

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.

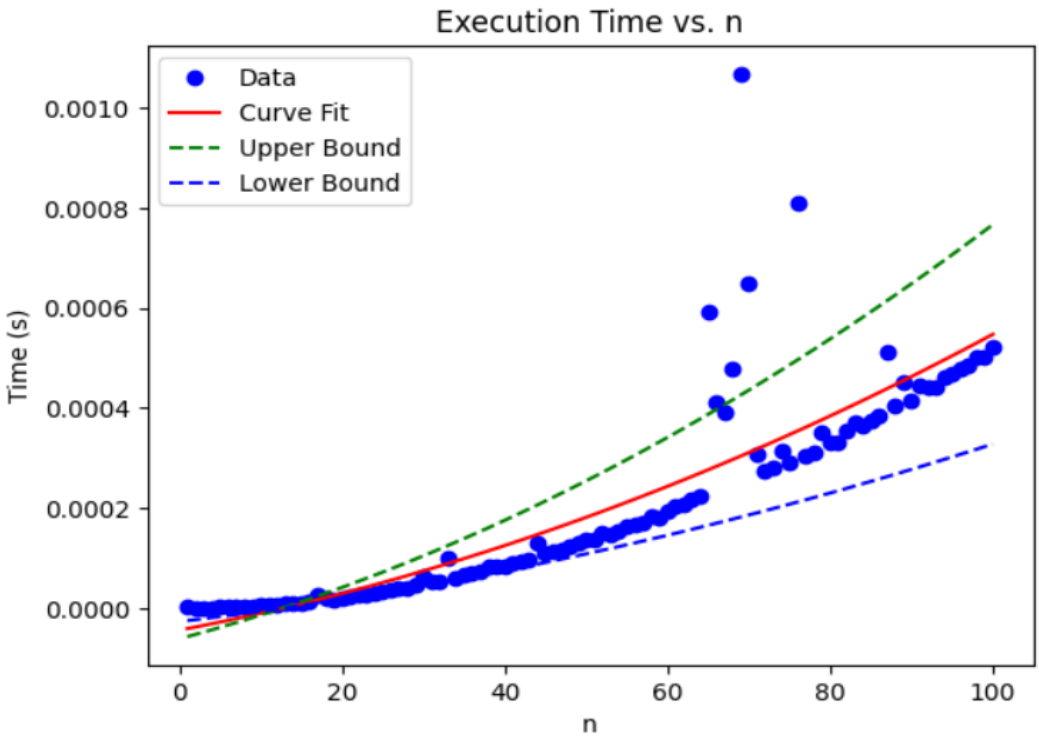
Sol : Upper bound : n^2

Lower bound : n

Big-O(n^2)

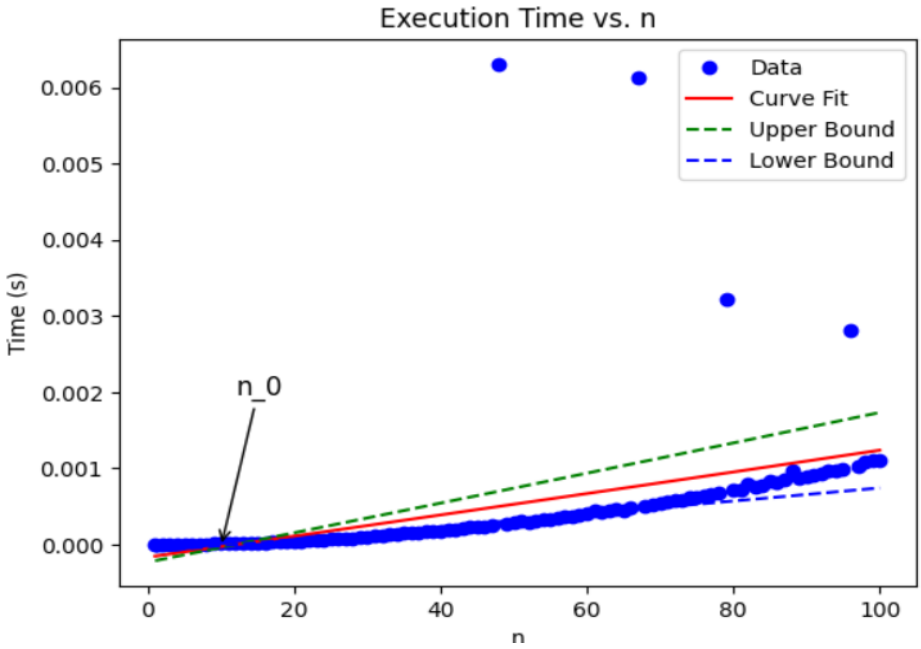
Big-Omega(n)

Big-theta(n^2)



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.

Sol: I visualised and identified the point at the actual data starts deviating from the fitted curve and I have choose the point n_0 and selected the point to be 10. At this point execution time start deviating from the polynomial curve and I have marked the n_0 with an arrow and labelled it on the plot as n_0.



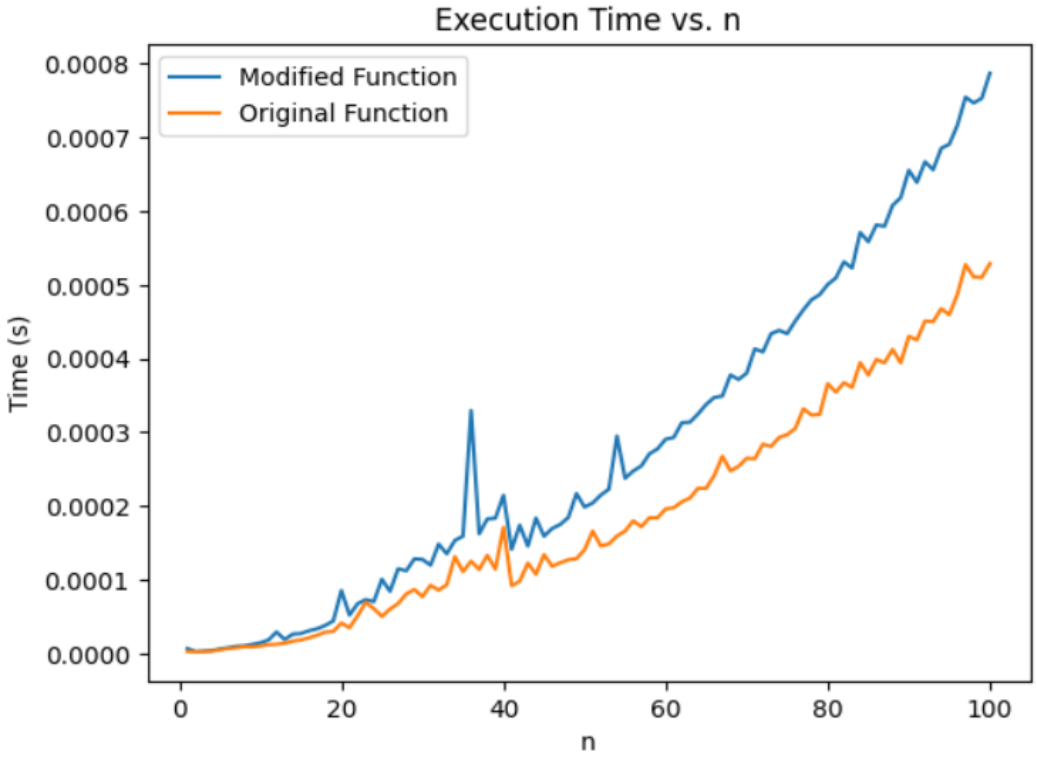
If I modified the function to be:

```
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)?

Sol: Yes, modifying the function will increase the time for running the algorithm. In the original function 'x=f(n)', the time complexity is O(n^2). And in modified function, there is an extra operation i.e 'y = i + j' in the nested loop. As this operation is also of constant time, it still increase the total number of operations that will be performed in the nested loop. As a result, overall time complexity will likely increase. Therefore, modifying the function additional operations within nested loops tends to increase in the runtime of the algorithm.

Below is the snap for comparing the modified function and original function runtime.



From the graph we can say that there's an slightly higher execution time for modified function when compared to original function.

5. Will it effect your results from #1?

Sol : No, it does not effect the results because mathematically, for both the modified and original functions algorithm's runtime is O(n^2). Its because of the dominating term in the runtime is nested loop iterating from 1 to n and there is an extra constant operation y = i + y does not change the complexity. But this constant factor results in the longer execution times for modified function compared to the original function.