```
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).

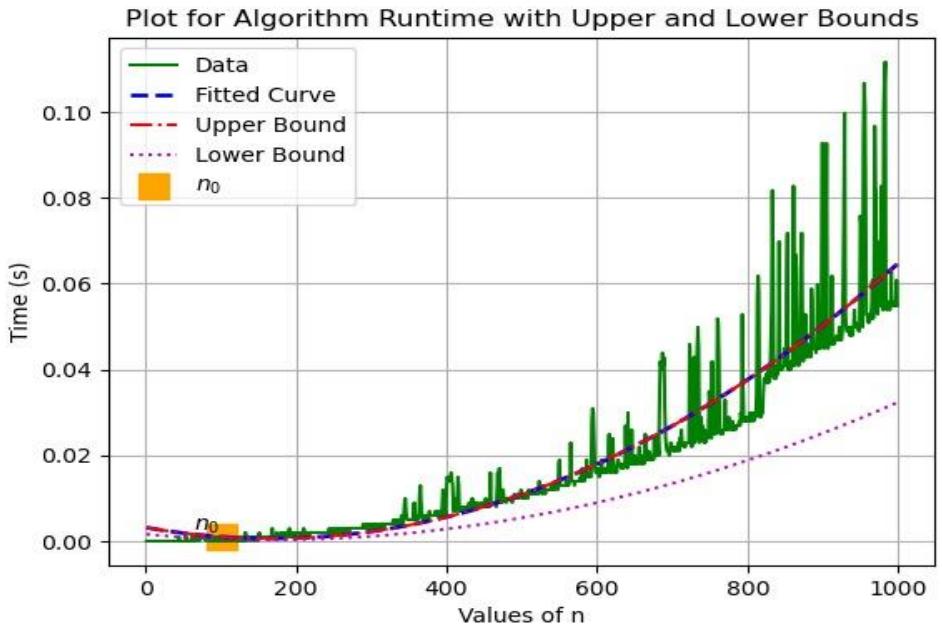
Runtime:
Each iteration of the x = x + 1 statement takes constant time (O(1)). Since this statement executes n^2 times, the overall runtime of the function is also **O(n^2)**, represented mathematically as:

Runtime = ∑(outer loop iterations) * ∑(inner loop iterations per outer loop) * (time per iteration)
$T(n)=n\sum i * n\sum j * 1 = n \cdot n$
Runtime = ∑(n iterations) * ∑(n iterations/outer loop) * (O(1) time/iteration)
Runtime = n * n * O(1)
Runtime = O(n^2)

The runtime of the algorithm T(n) = n^2 and the time complexity 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.

I have plotted data from 1 to 1000



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:

$g(n)=2an\text{^}2+bn+c$

where a, b, and c are the coefficients of the fitted curve.

Lower Bound:

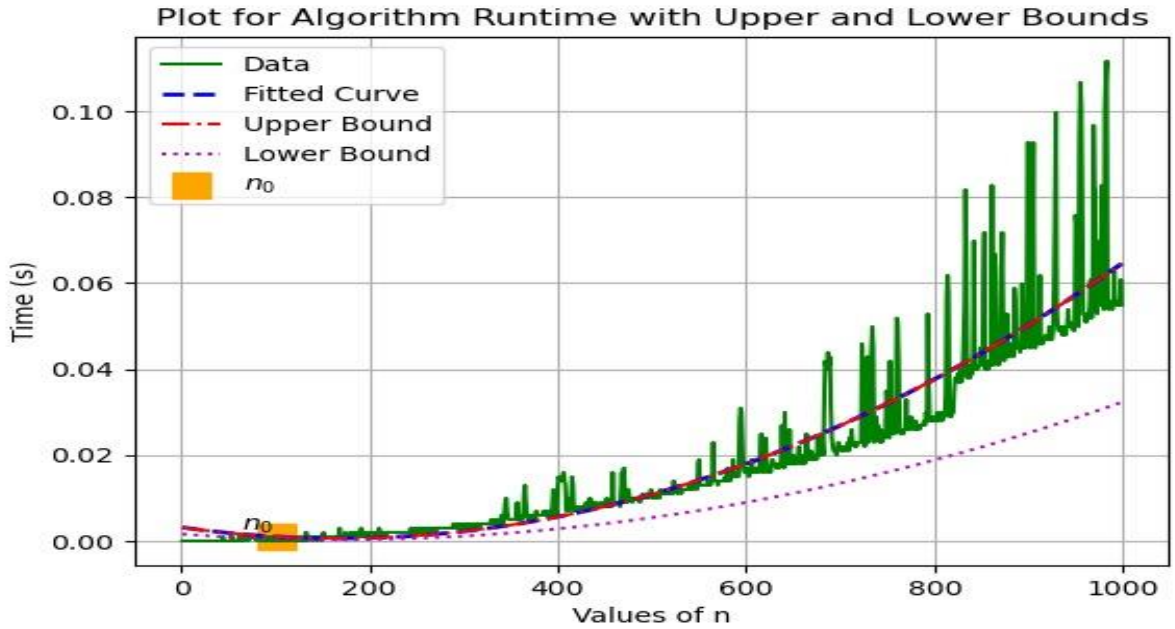
$h(n)=0.5an\text{^}2+bn+c$

Where a, b, and c are coefficients to the fitted curve

Big O Notation - O(n2)
Big Ω Notation - Ω(n2)
Big Θ Notation - Θ(n2)

4. Find the approximate (eyeball 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.

The approximate location of **n_0** is the point where the execution time data deviates from the trend of the fitted polynomial curve. By zooming in on the plot, we can visually identify the point where the algorithm's behavior changes.



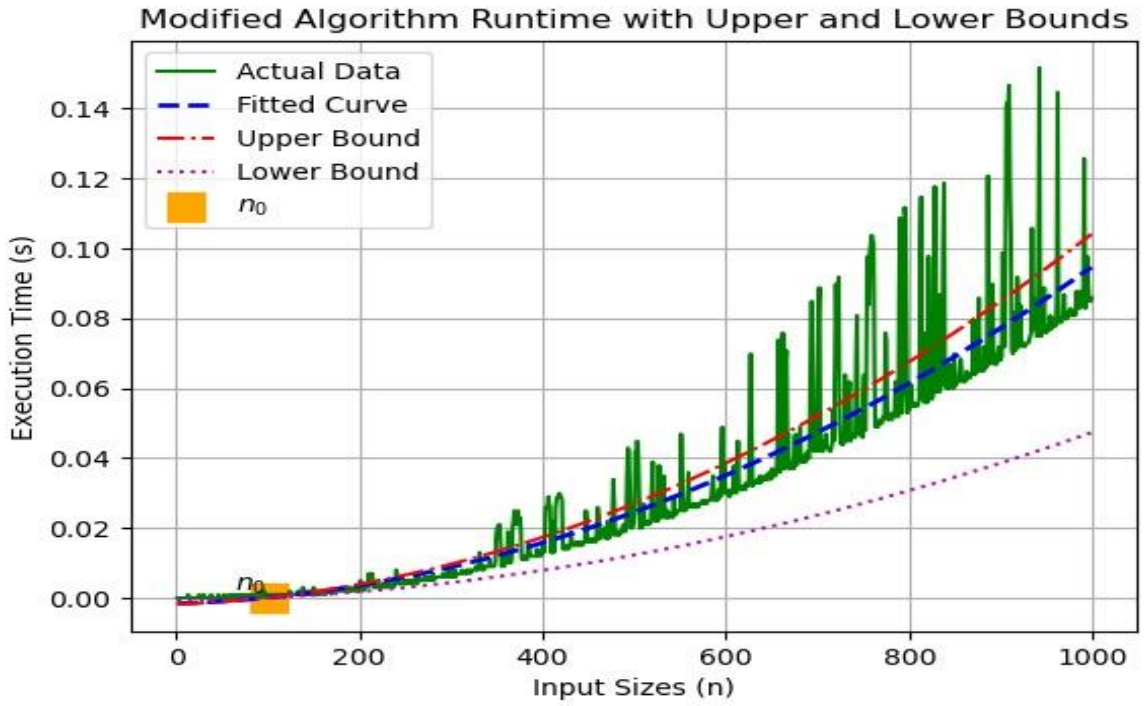
If I modified the function to be:

```
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)?

The modification introduces an additional operation, y= i + j; inside the inner loop. This operation has a constant time complexity, and it doesn't depend on the size of the input n. Therefore, the overall time complexity of the modified function remains quadratic (O(n^2)). The additional operation does not change the order of growth; it just adds a constant factor to the running time.

In terms of actual runtime, the modified function may take slightly longer due to the extra arithmetic operation, but the impact would likely be minor, especially for larger values of n.



5. Will it effect your results from #1?

The results from #1, including the fitted curve and the upper/lower bounds, might be affected, but not significantly. The additional arithmetic operation is constant time and doesn't change the asymptotic behavior, which is captured by the polynomial fit.

In terms of big-O, big-Omega, and big-Theta notations, they are determined by the dominant term in the time complexity, and that remains (O(n^2)).. The exact coefficients of the polynomial might change, but the overall order of growth remains the same.