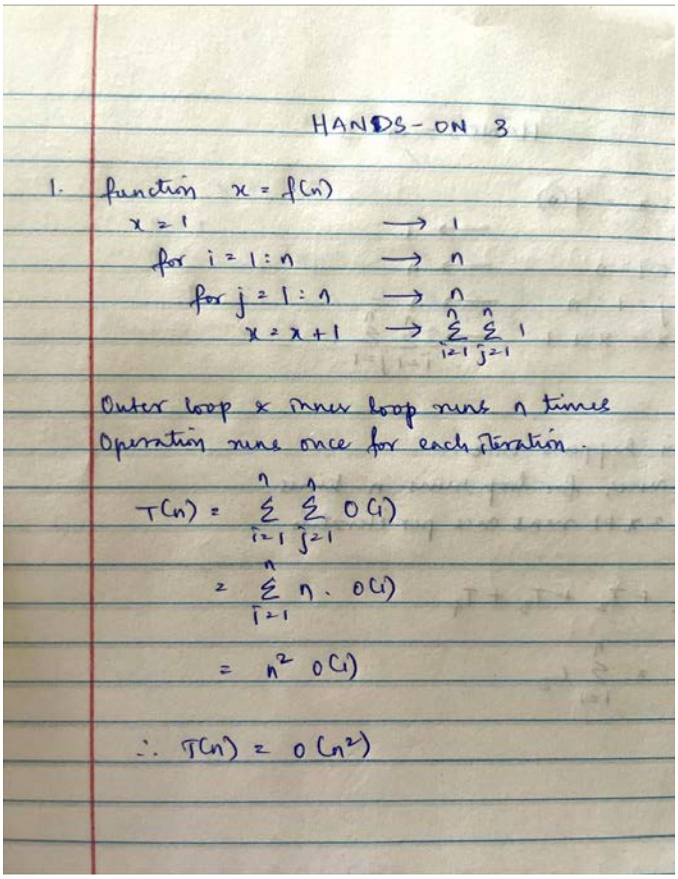
Hands-on 3

```
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).



HANDS-ON 3

1. function $x = f(n)$

$x = 1 \longrightarrow 1$

for $i = 1 : n \longrightarrow n$

for $j = 1 : n \longrightarrow n$

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

Outer loop & inner loop runs n times

Operation runs once for each iteration.

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} O(1)$$

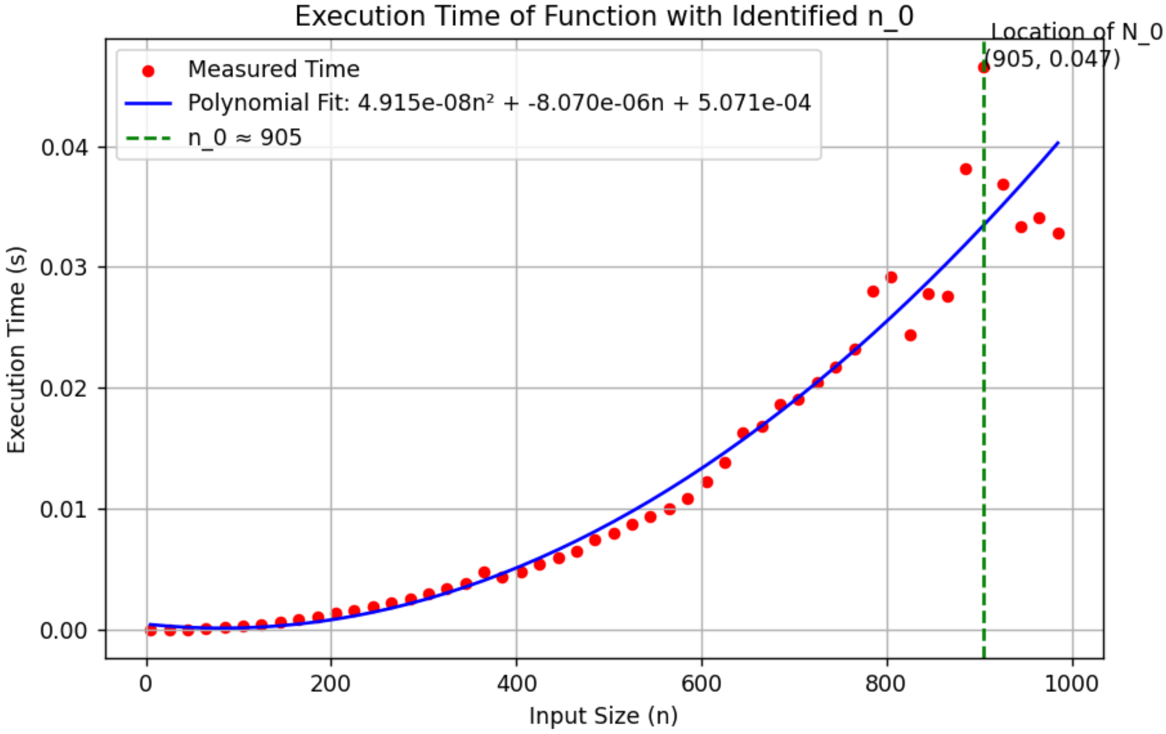
$$= \sum_{i=1}^{n} n \cdot O(1)$$

$$= n^2 \, O(1)$$

$$\therefore T(n) = O(n^2)$$

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.

   Below attached is the plot of Execution Time (in seconds) vs Input Size (n). The data has been fitted with a polynomial to show the relationship between the input size and execution time.



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.

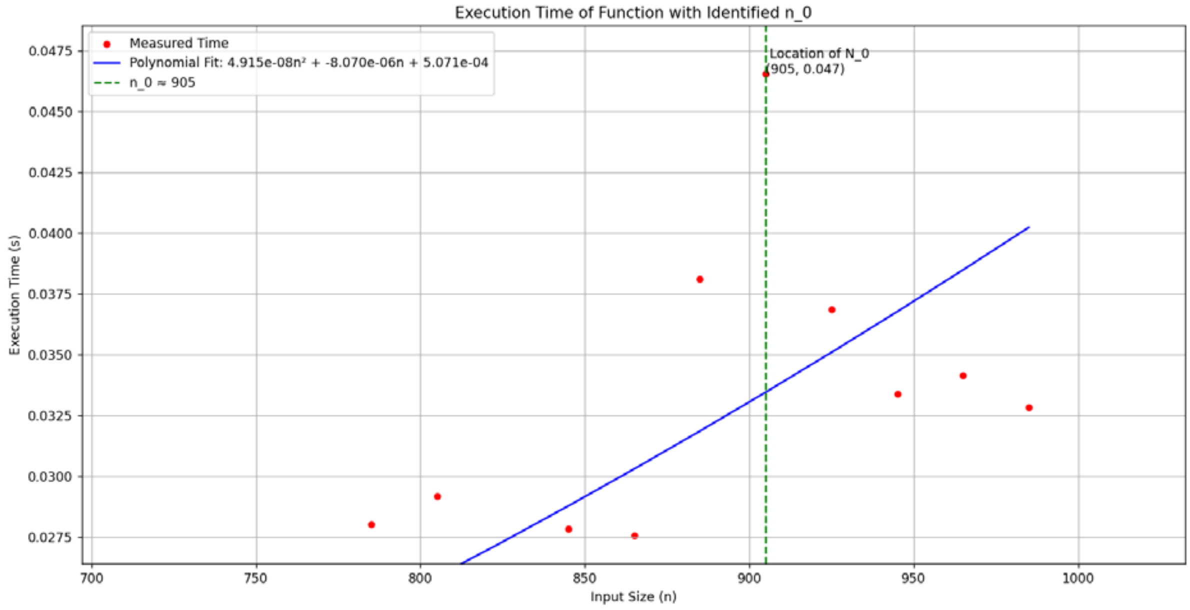
   The curve is a quadratic equation of the form: **y = ax^2 + bx + c**

   Big-O: O(n^2) because the upper bound grows quadratically with n.

   Big-Omega: Ω(n^2) because the lower bound grows quadratically with n.

   Big-Theta: Θ(n^2) because the runtime grows quadratically with n, and both the upper and lower bounds are 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.



Based on the plot, n_0 is indicated by the green line. Looking at the plot, we conclude that n_0 occurs at around **905**. n_0 = 905 is picked because it shows the maximum height deviation the occurred during the execution of the algorithm.

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;
```

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

The difference between the original and the modified function is variable y.

Initialization (y=1) -> O(1)

Operation (y=i+j) -> O(n^2)

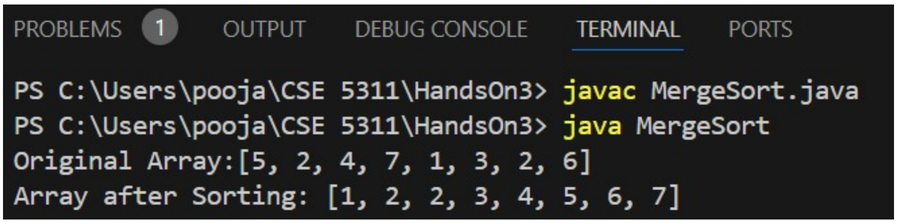
Therefore, though the additional operation will increase the runtime a bit, the overall time complexity of the algorithm will remain the same i.e. **O(n^2)** and the growth remains quadratic.

6.Will it effect your results from #1?

No, it will not affect the results. The time complexity will remain the same (O(n^2)), and the added operation does not change the behavior, it will only increase the constant time factor.

7.Implement merge sort, upload your code to github and show/test it on the array [5,2,4,7,1,3,2,6].

> https://github.com/poojaapari/CSE-5311---Design-and-Analysis-of-Algorithms-/blob/master/MergeSort.java


PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\pooja\CSE 5311\HandsOn3> javac MergeSort.java
PS C:\Users\pooja\CSE 5311\HandsOn3> java MergeSort
Original Array:[5, 2, 4, 7, 1, 3, 2, 6]
Array after Sorting: [1, 2, 2, 3, 4, 5, 6, 7]