**Algorithm Complexity**

Time Complexity

Big O Notation

Time Complexity

Compare the time taken to execute the program.

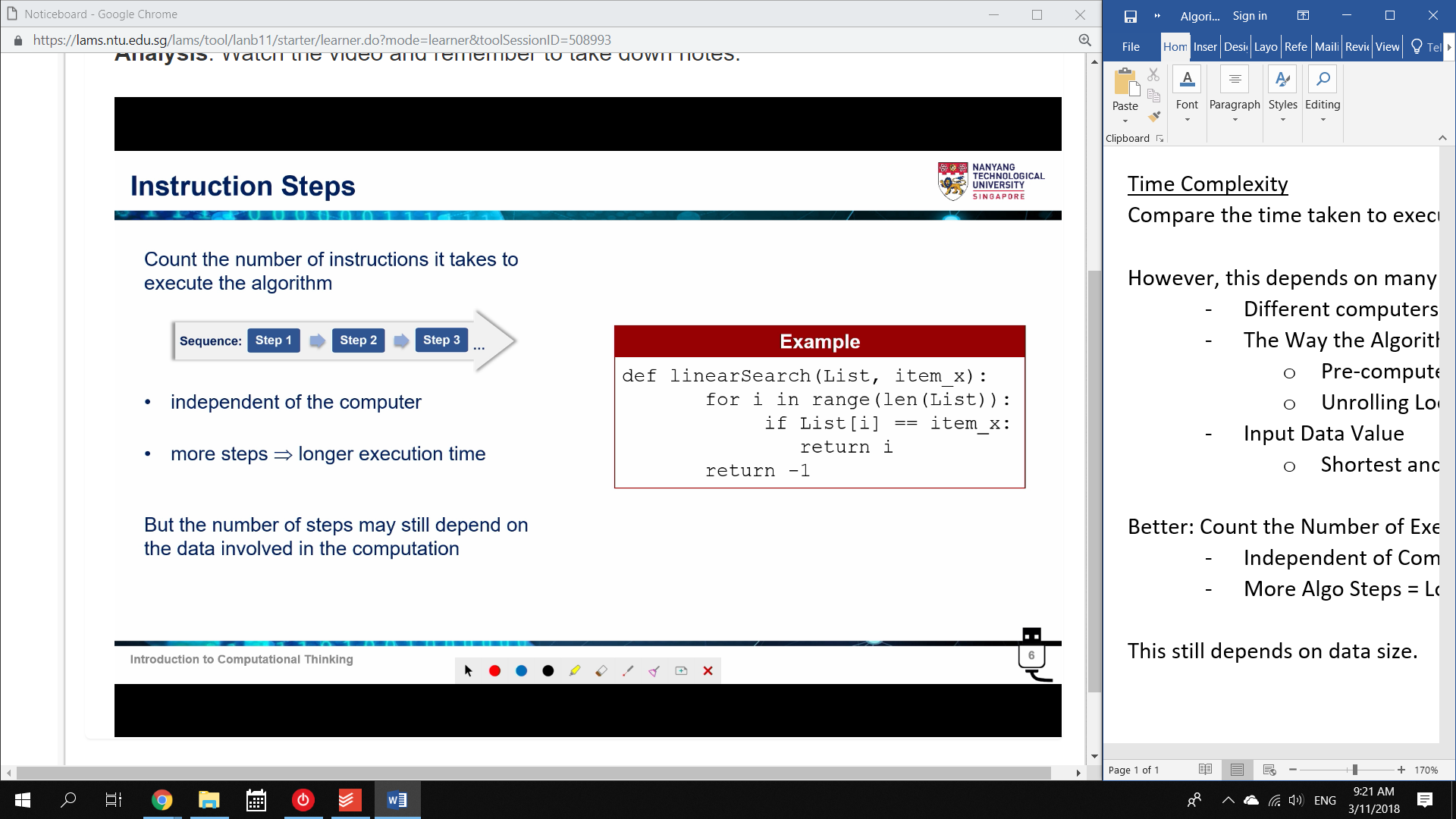
However, this depends on many factors.

* Different computers have different speeds
* The Way the Algorithm is implemented
  + Pre-compute Lookup table
  + Unrolling Loops
* Input Data Value
  + Shortest and Longest possible Cases

Better: Count the Number of Execution Steps.

* Independent of Computer
* More Algo Steps = Longer

This still depends on data size.



Depending on the position of item\_x

Therefore:

Consider only the Worst Case scenario.

Does difficulty scale linearly or Exponentially?

* The Asymptotic Behaviour

This is Big O notation: ‘Order’ of growth complexity.

Big O Notation

Measure and compare time complexity of algorithms.

Big O gives the upper bound on the asymptotic growth.

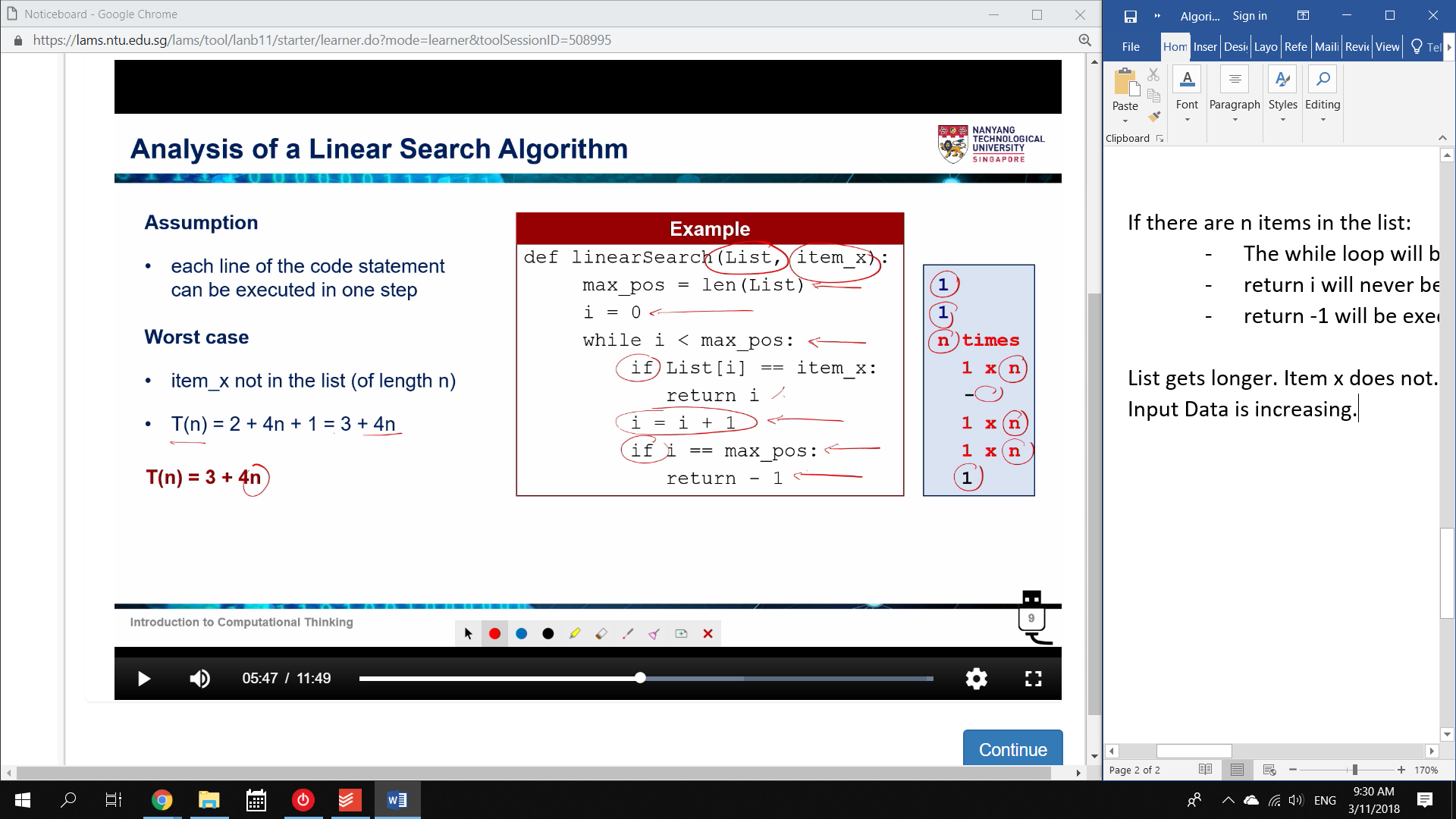
How does Execution Time increase as Input Data size Grows?

Execution Time

1. Number of Instruction Steps

2. Worst Case Scenario

Eg. Linear Search



If there are n items in the list:

* The while loop will be executed n times.
* return i will never be executed. X is not in the list.
* return -1 will be executed once, at the end.

List gets longer. Item x does not.

Input Data is increasing.

T(n) = 3 + 4n

* n increases linearly; we can ignore 3

T(n) = 4n for large n (asymptotic behaviour)

Therefore:

* T(n) increases proportionally with n
* T(n) doubles when n is doubled. It is Linear.

Growth Order: F(n) = n

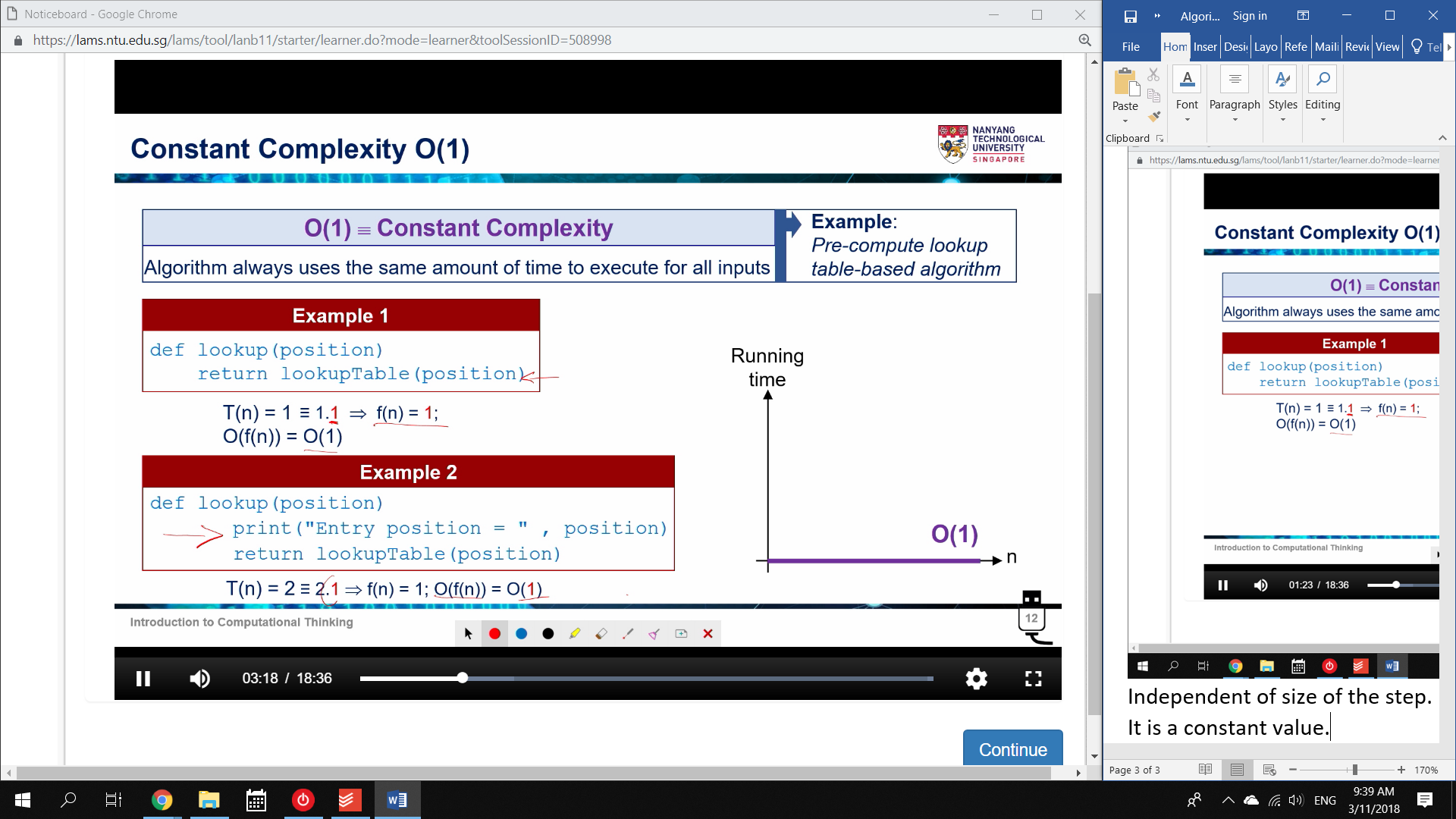
Big O notation: O(F(n)) = O(n)

* Linear Complexity



Constant Complexity

Algorithm always uses same amount of time to execute.



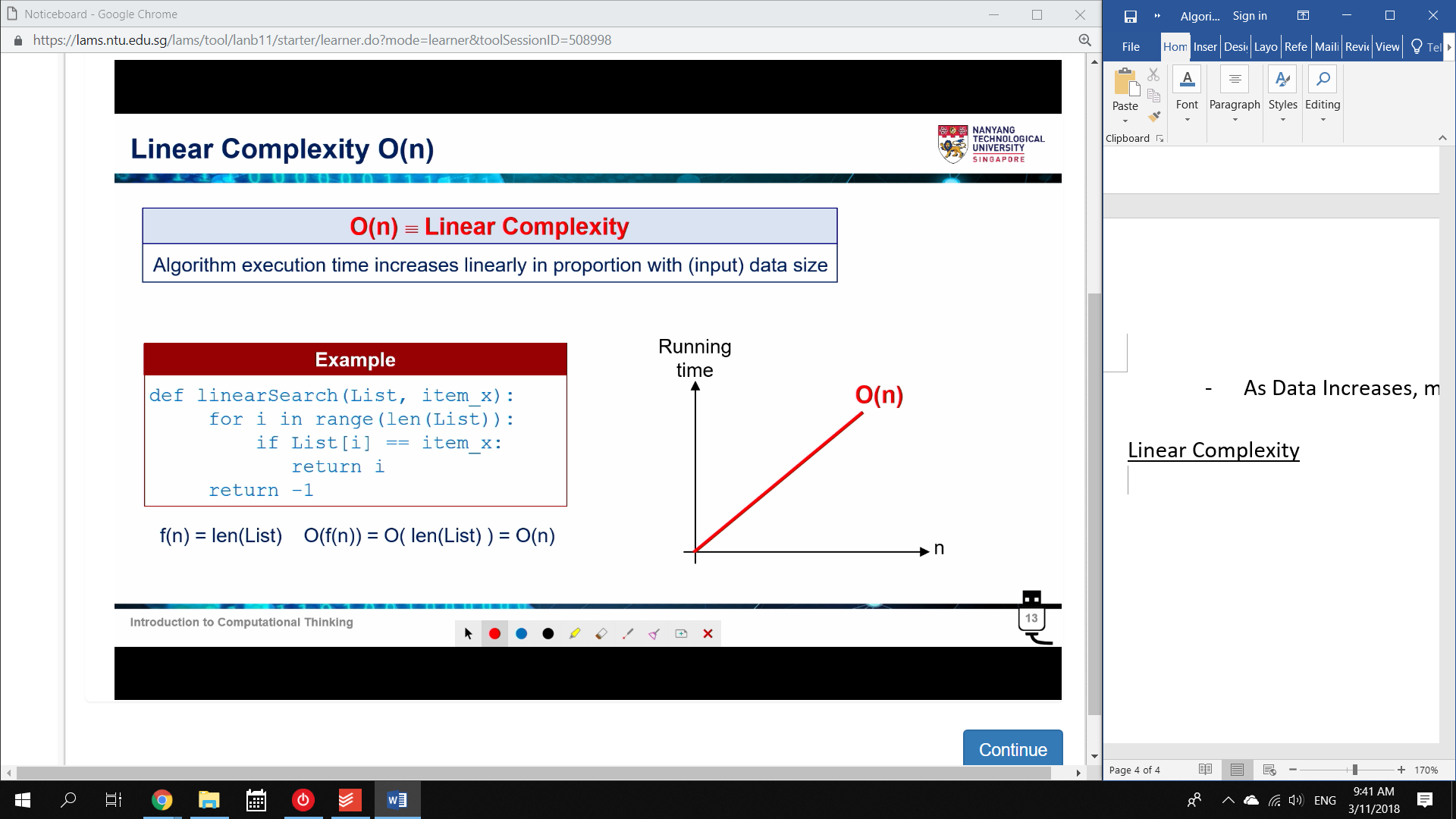
Independent of size of the step.

It is a constant value.

Tradeoff: Storage Space.

* A Pre-Compute Lookup Table takes up space
* As Data Increases, more space is needed.

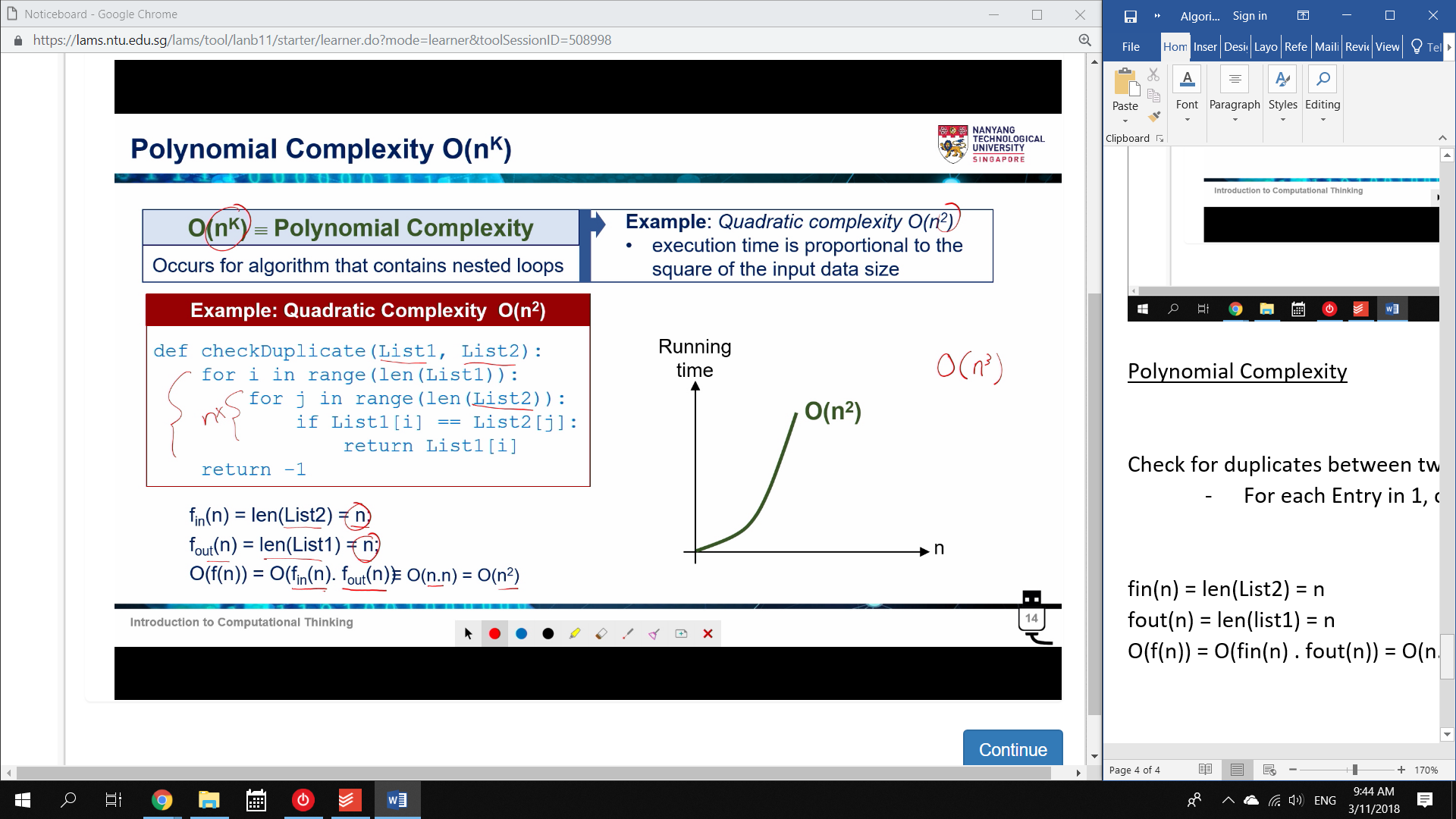
Linear Complexity



Polynomial Complexity

Check for duplicates between two Lists.

* For each Entry in 1, code will execute len(list2) times



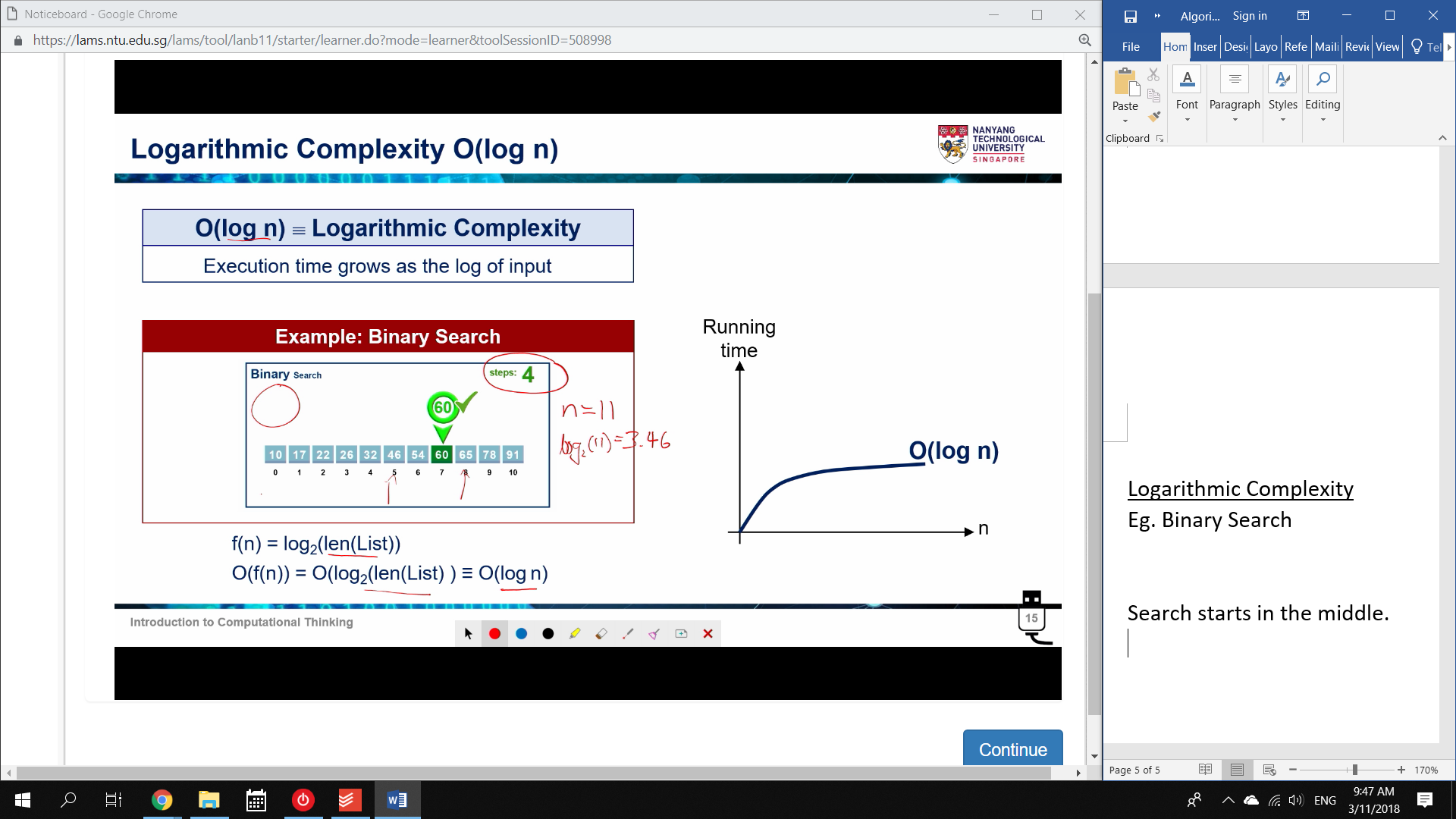
fin(n) = len(List2) = n

fout(n) = len(list1) = n

O(f(n)) = O(fin(n) . fout(n)) = O(n.n) = O(n2)

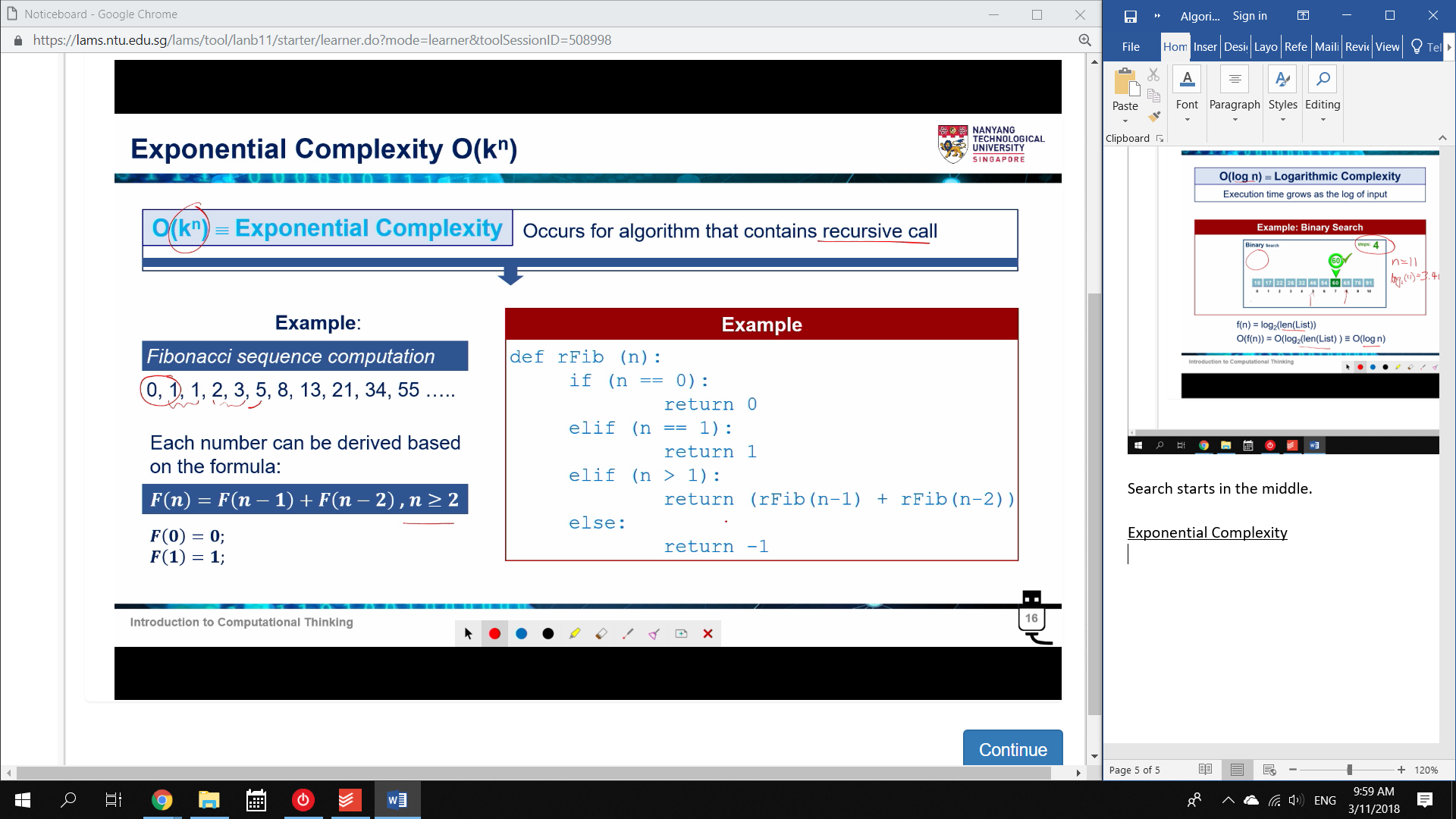
Logarithmic Complexity

Eg. Binary Search

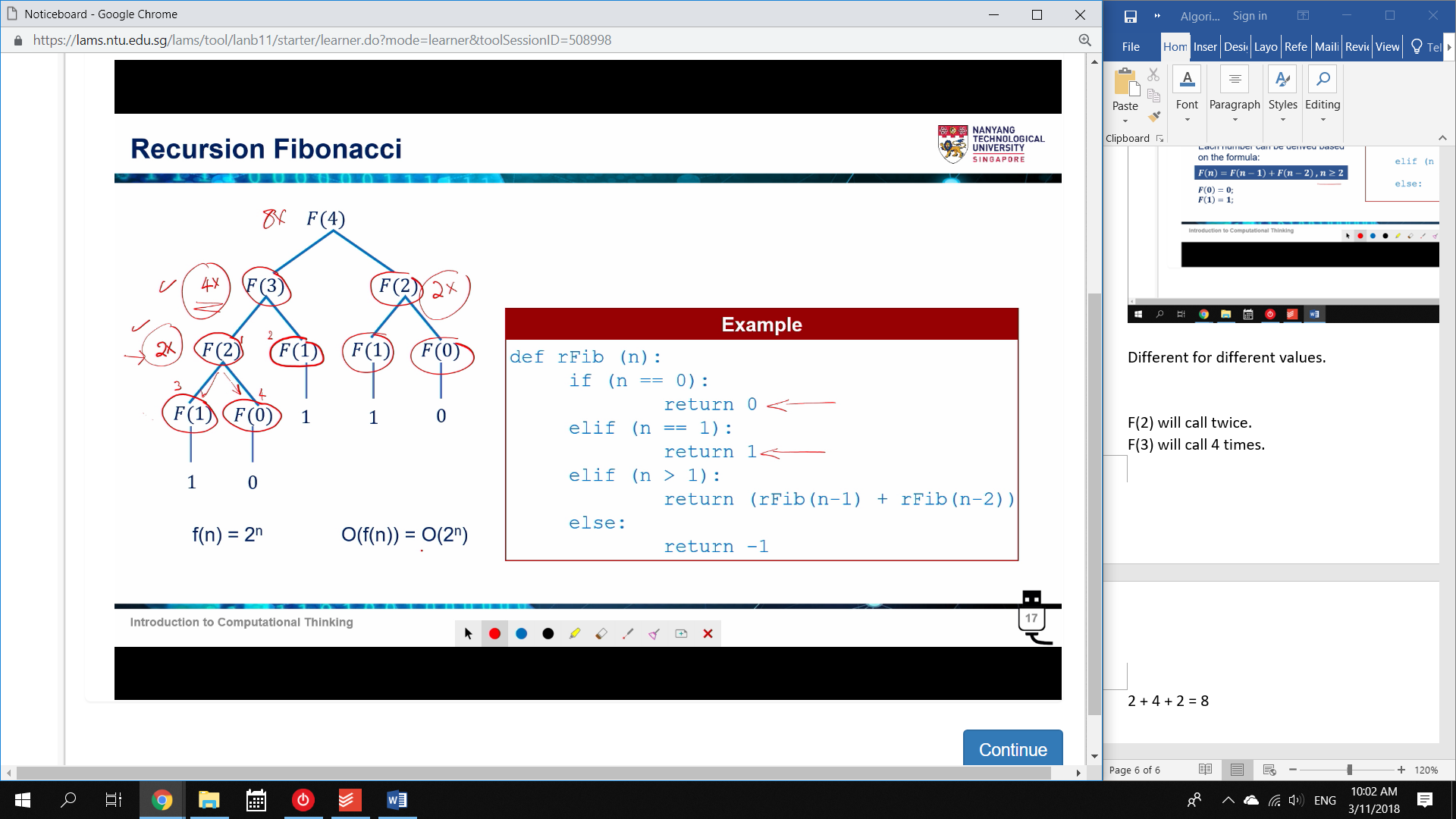


Search starts in the middle.

Exponential Complexity



Different for different values.



F(2) will call twice.

F(3) will call 4 times.

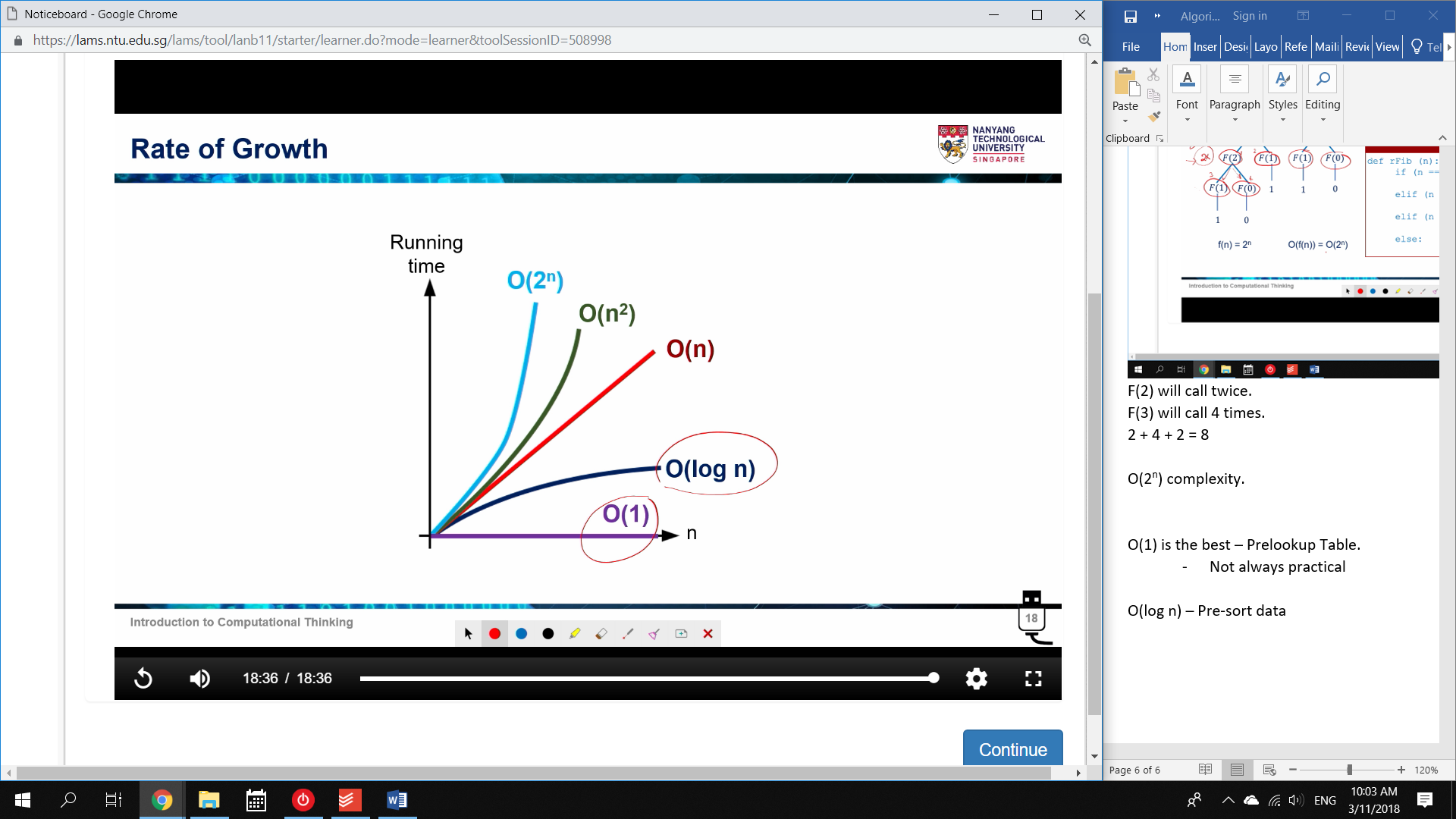
2 + 4 + 2 = 8

O(2n) complexity.

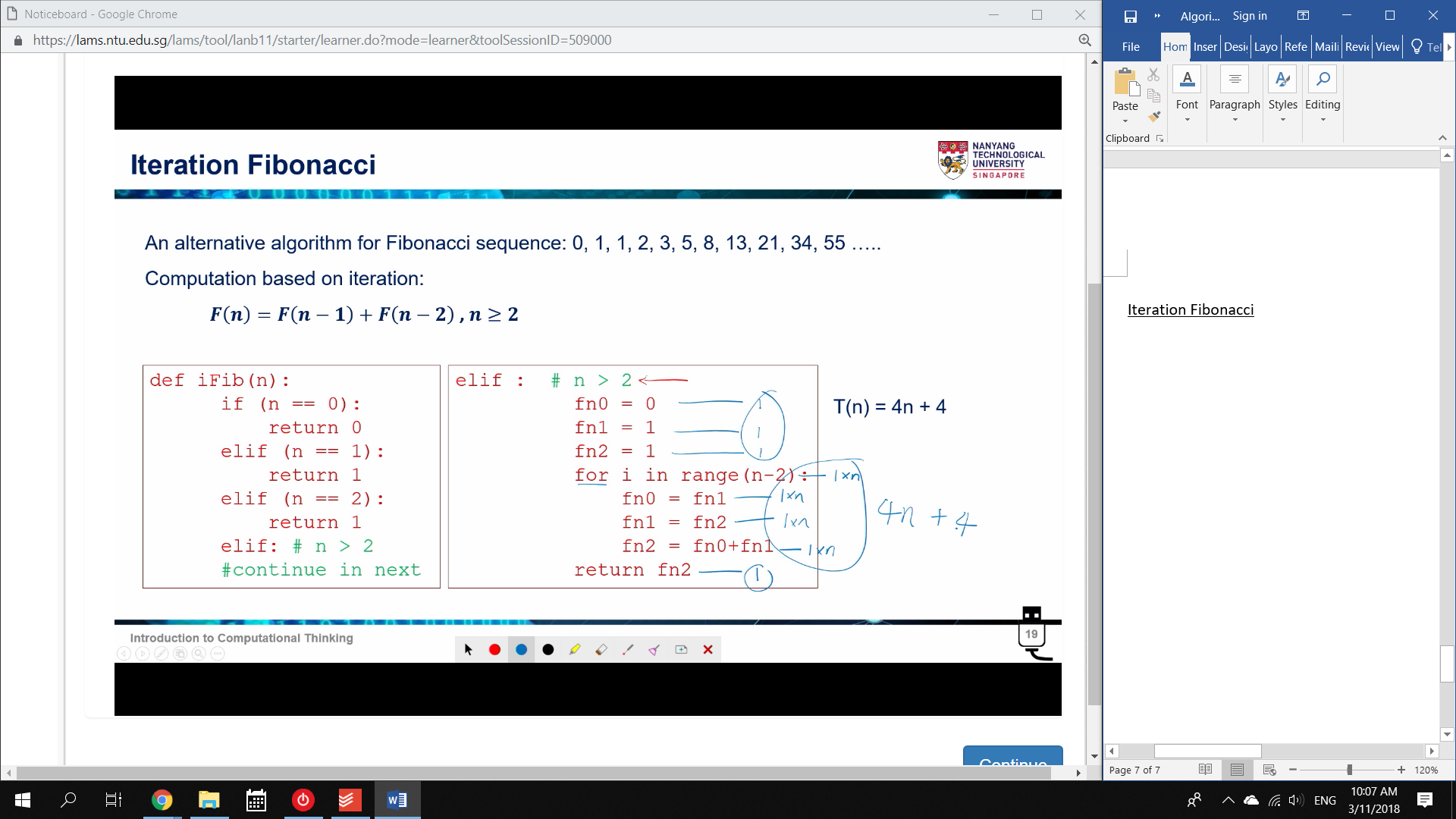
O(1) is the best – Prelookup Table.

* Not always practical

O(log n) – Pre-sort data



Iteration Fibonacci



4 lines in the *for* loop 🡪 4n

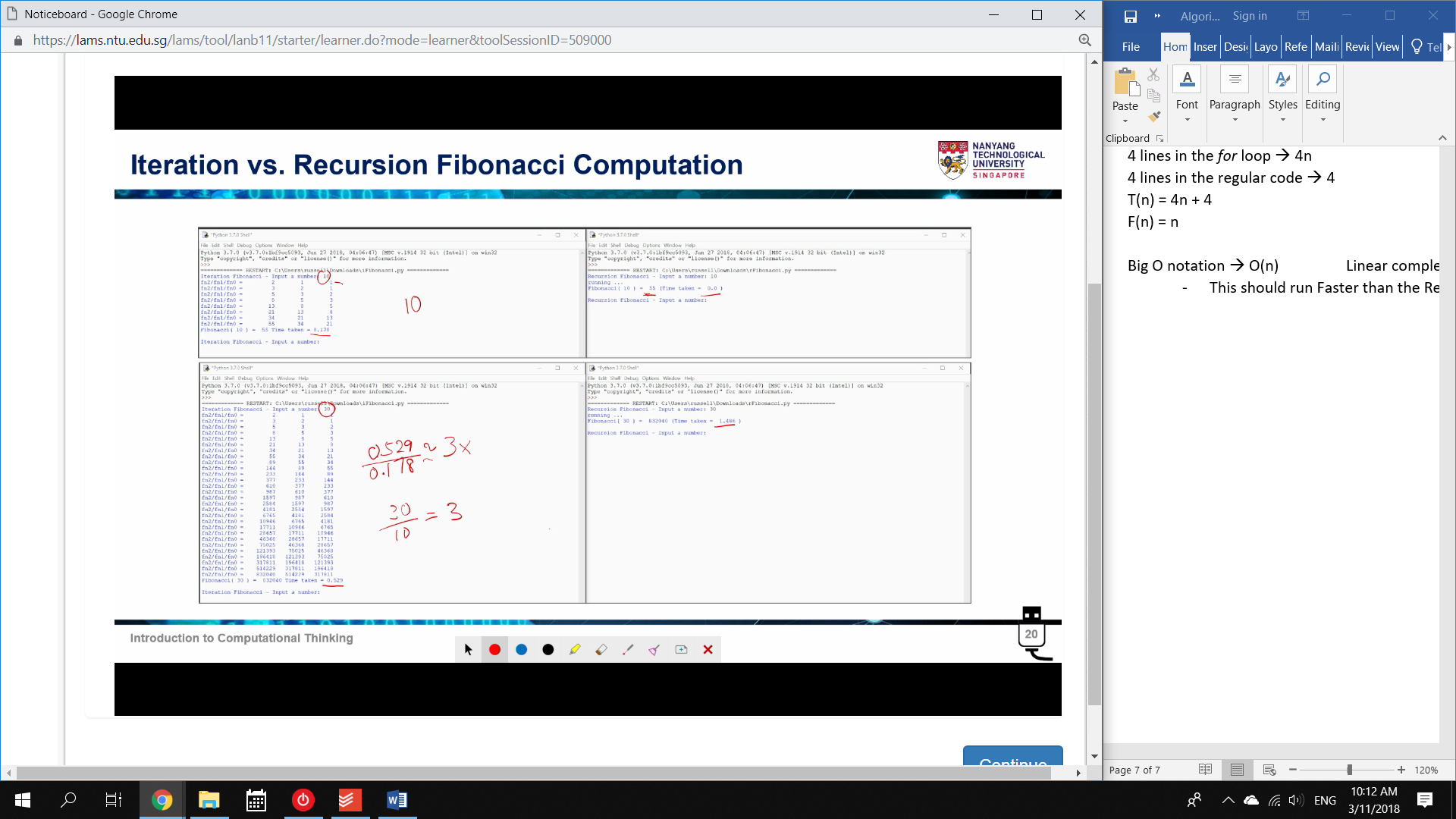
4 lines in the regular code 🡪 4

T(n) = 4n + 4

F(n) = n

Big O notation 🡪 O(n) Linear complexity

* This should run Faster than the Recursive version.



Entries Iterative Recursive

10 0.178s 0.00s

30 0.529s 1.50s

35 0.643s 15s

37 0.654s 38s

Therefore, the larger the data, the longer recursive takes in comparison.