**7. Computational Binary Tree**

Decomposition

The process of breaking down a problem into parts.

Divide and Conquer each Part of a Problem

* Less complex
* Parts can be solved by different parties
* Can be analysed from different perspectives

Complex problem 🡪 Subproblems 🡪 Sub-solutions 🡪 Final Solution

eg.

1. Complex:

1 + 2 + … + 100

2. Subproblems:

* 1 + 100
* 2 + 99
* …
* 50 + 51

3. Subsolutions:

* Each is 101 🡪 Recursion possible

4. 101 \* 50 = 5050

**In Python**

Topics:

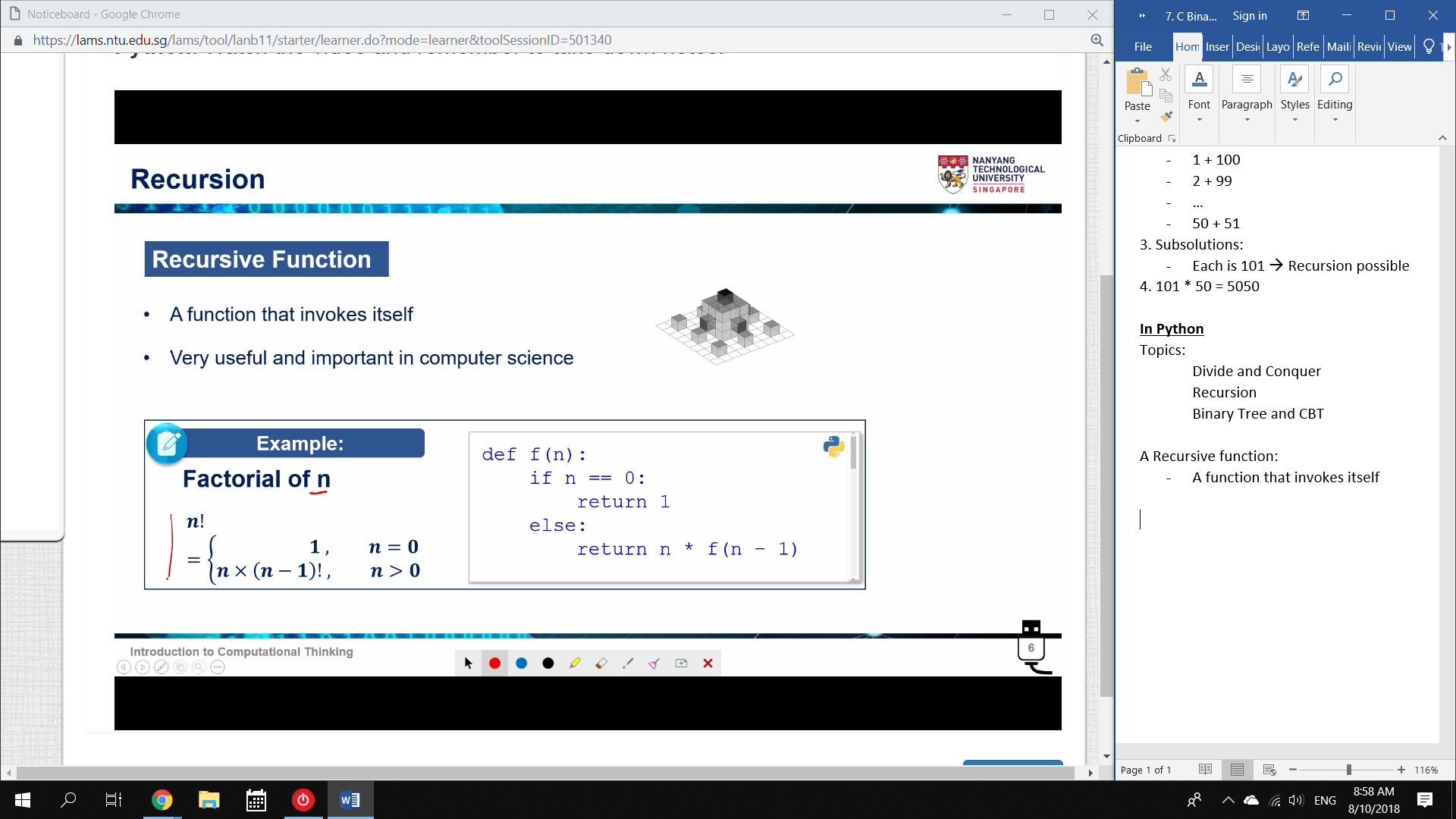
Divide and Conquer

Recursion

Binary Tree and CBT

A Recursive function:

* A function that invokes itself



Here, the function f(n) calls f(n-1).

This allows it to work with every number from n, n-1, …, 1.

* We want a return of: “Multiply every number from n to 1”

How to write a recursive Function

1A. Parameters

* What range of values do you need to work with?

1B. Return

* What do you need the result to be?

1C. Functionality

* How does this suit what you need?

2. Assume you have implemented the function

3. Function Body

* Decompose into Subproblems (simplest possible structure)
* Call This Function. Solve Each subproblem.
* Compose the final result, and return it.

Reversing a String

We want to reverse the order of letters in a string.

‘abcdef’ 🡪 ‘fedcba’

1. Base Case: ‘a’ 🡪 just print ‘a’.

2. Subproblems:

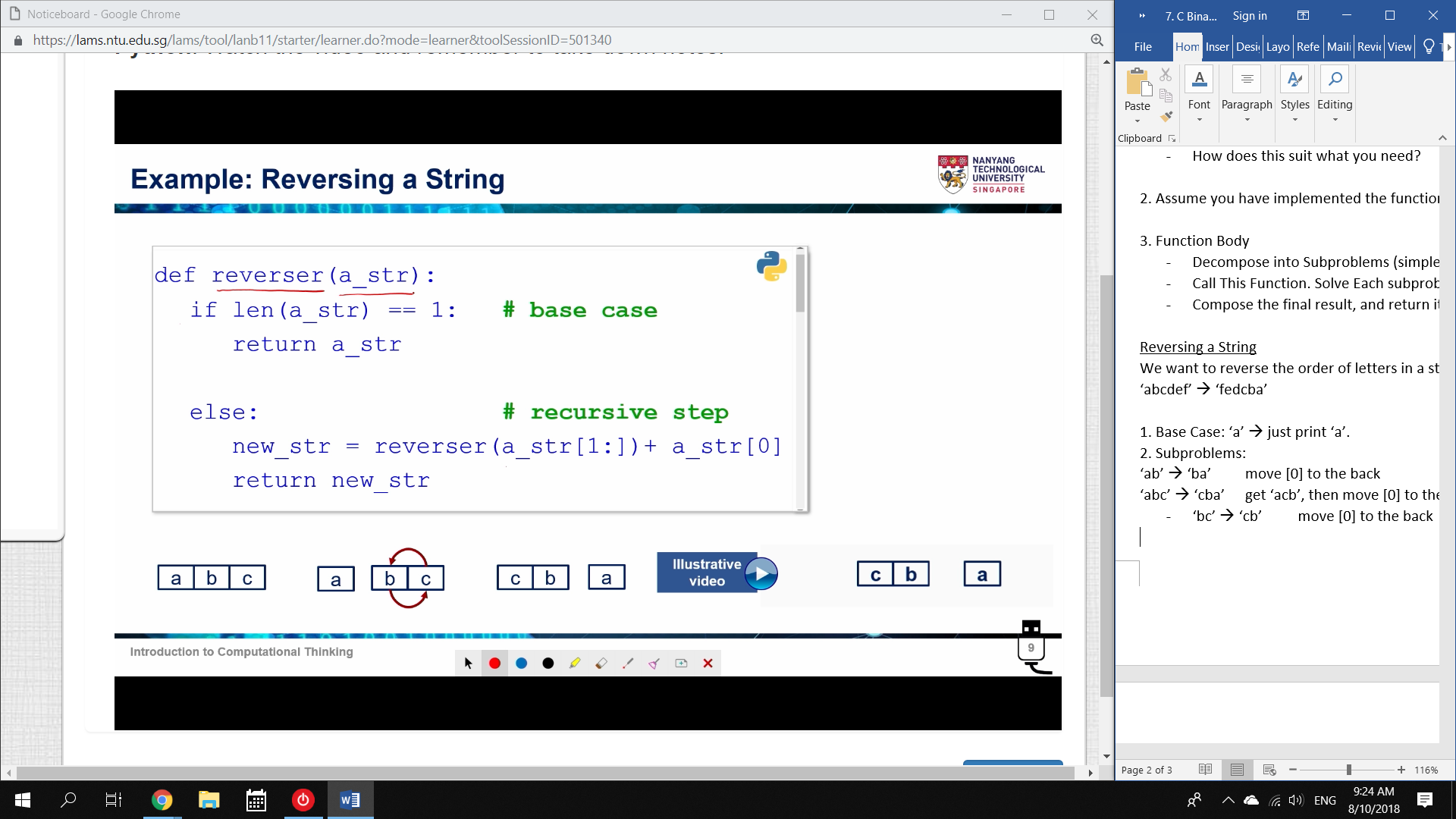
‘ab’ 🡪 ‘ba’ move [0] to the back

‘abc’ 🡪 ‘cba’ get ‘acb’, then move [0] to the back

* ‘bc’ 🡪 ‘cb’ move [0] to the back

Consider the method: “Move [0] to the back”

* (recursive function) + str[0]



This may not always be efficient:

eg. Fibonacci Number

F(5) = F(4) + F(3)

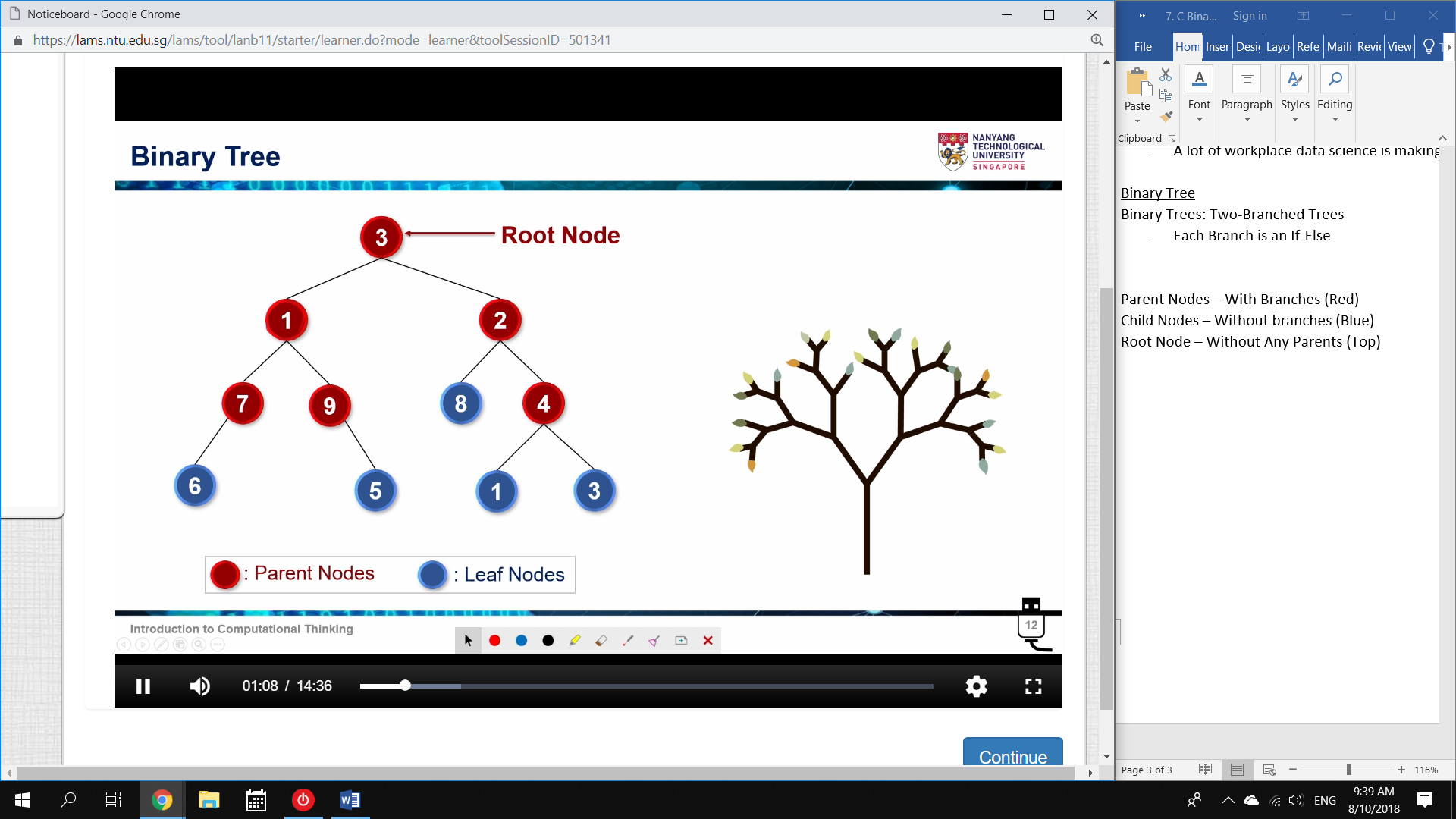
Here, the recursive function calculates F(3) twice and F(2) thrice.

* A lot of workplace data science is making computation faster and simpler.

Binary Tree

Binary Trees: Two-Branched Trees

* Each Branch is an If-Else



Parent Nodes – With Branches (Red)

Child Nodes – Without branches (Blue)

Root Node – Without Any Parents (Top)

Complete Binary Tree

Every parent has exactly 2 nodes.

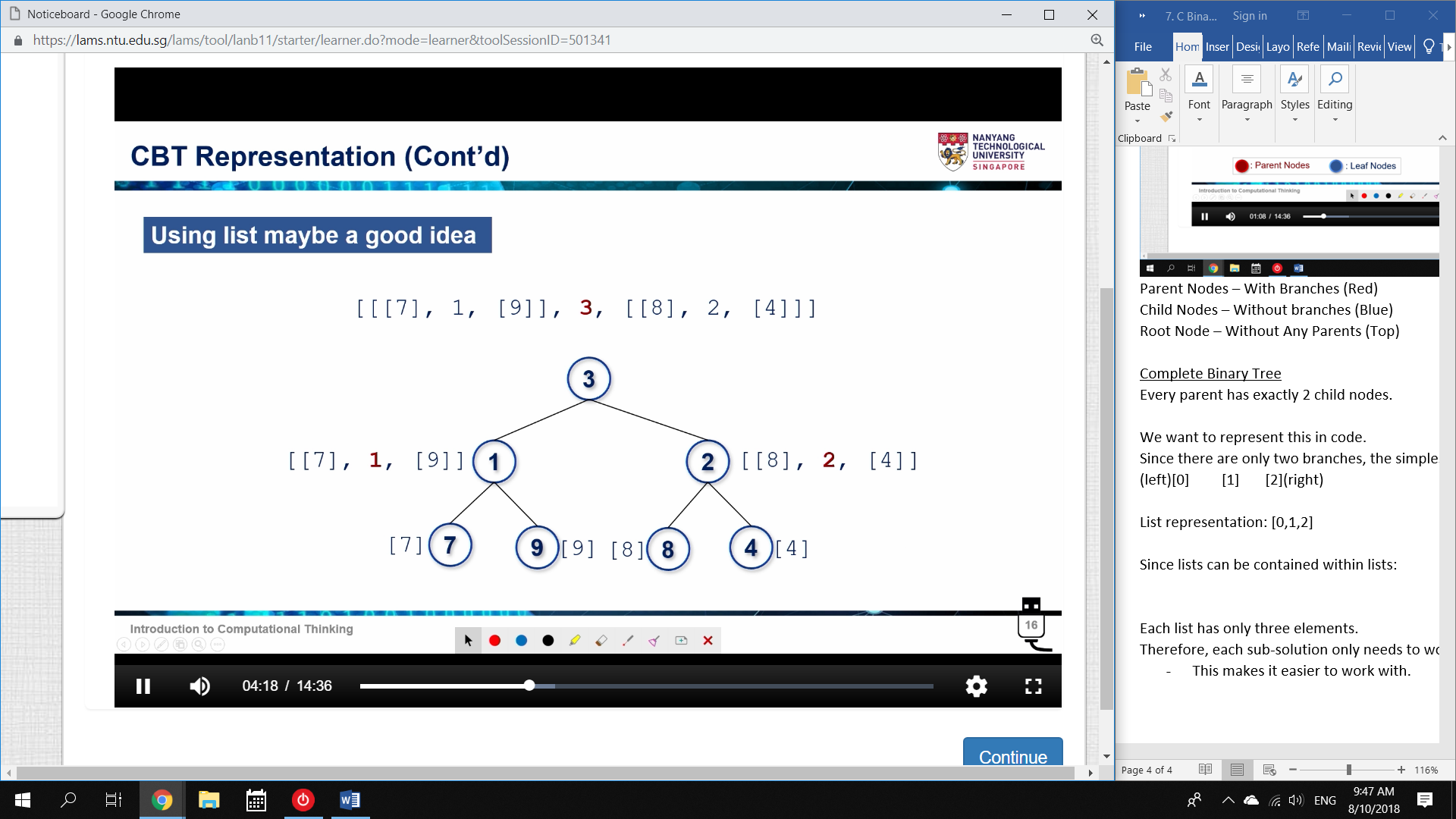
We want to represent this in code.

Since there are only two branches, the simplest method is: ‘Left’ and ‘Right’

(left)[0] [1] [2](right)

List representation: [0,1,2]

Since lists can be contained within lists:



Each list has only three elements.

Therefore, each sub-solution needs to work with exactly three elements.

* This makes recursion possible

Operations with CBT

numOfNodes(t): Total number of nodes

sumNodes(t): Summation of node Values

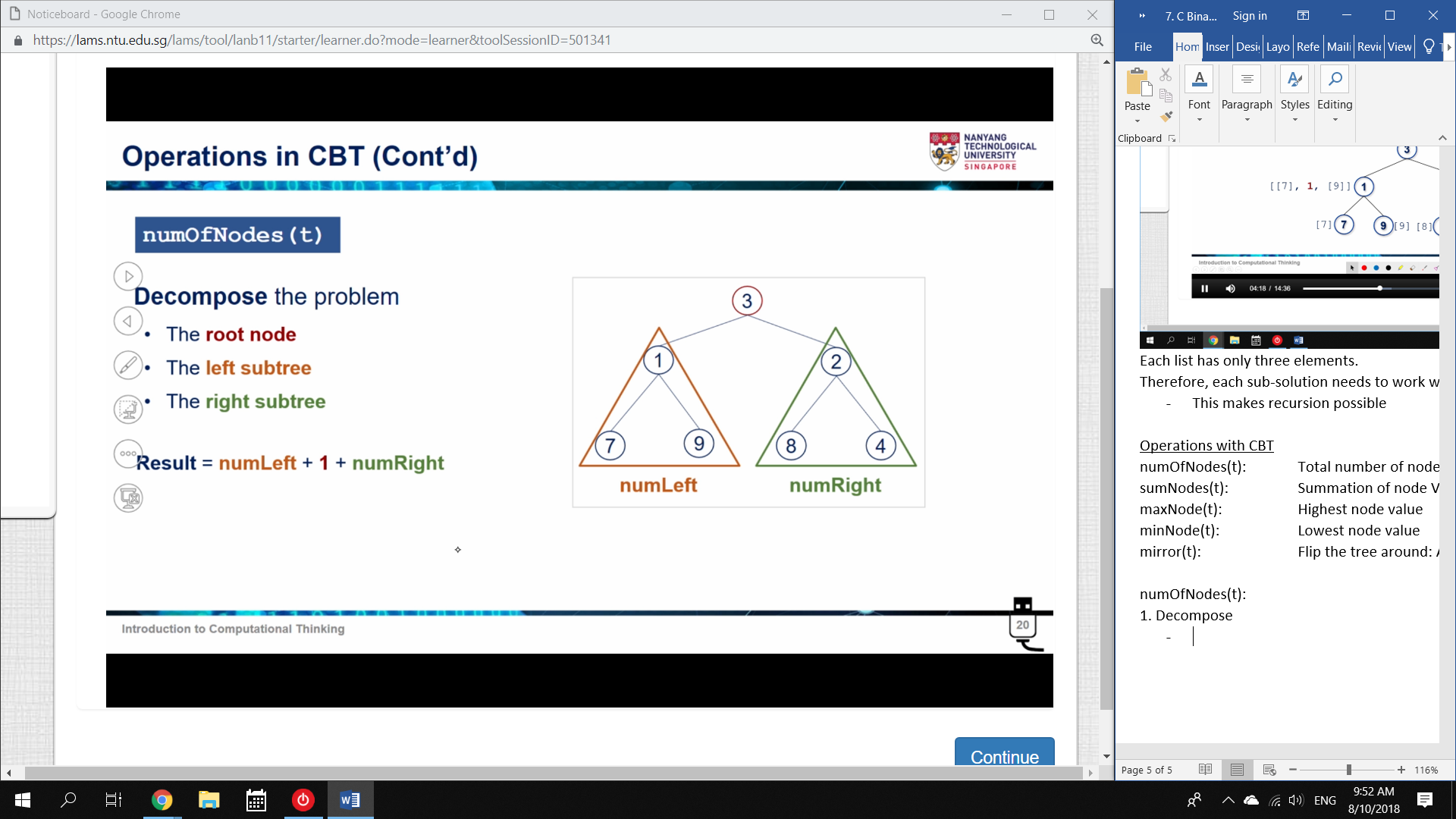
maxNode(t): Highest node value

minNode(t): Lowest node value

mirror(t): Flip the tree around: A mirrored CBT

numOfNodes(t):

1. Decompose



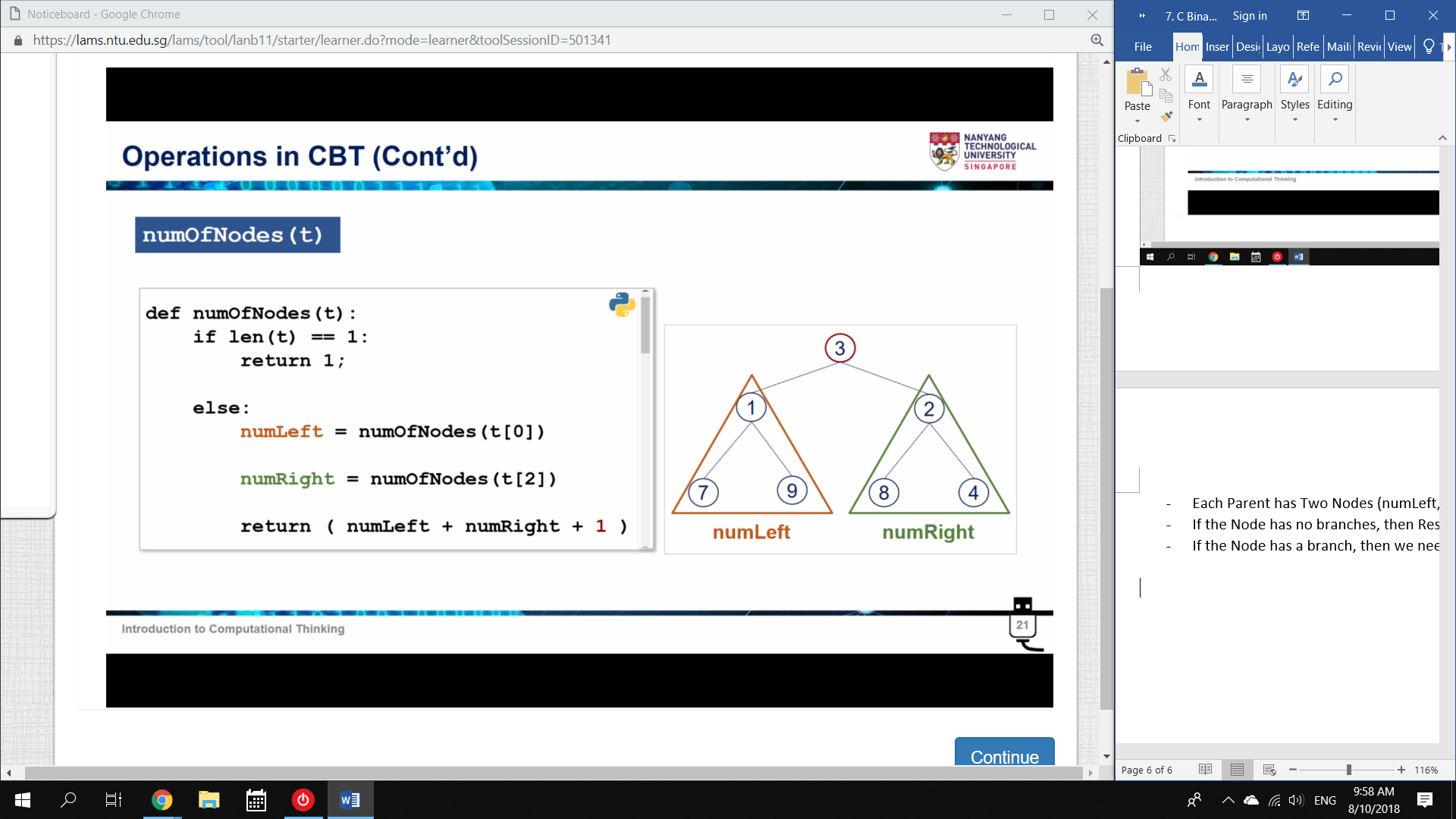
* Each Parent has Two Nodes (numLeft, numRight)

If a Node has no branches, the number of nodes is 1.

If a Node has branches, the number of nodes is 1 + leftbranch + rightbranch

1. Decompose:

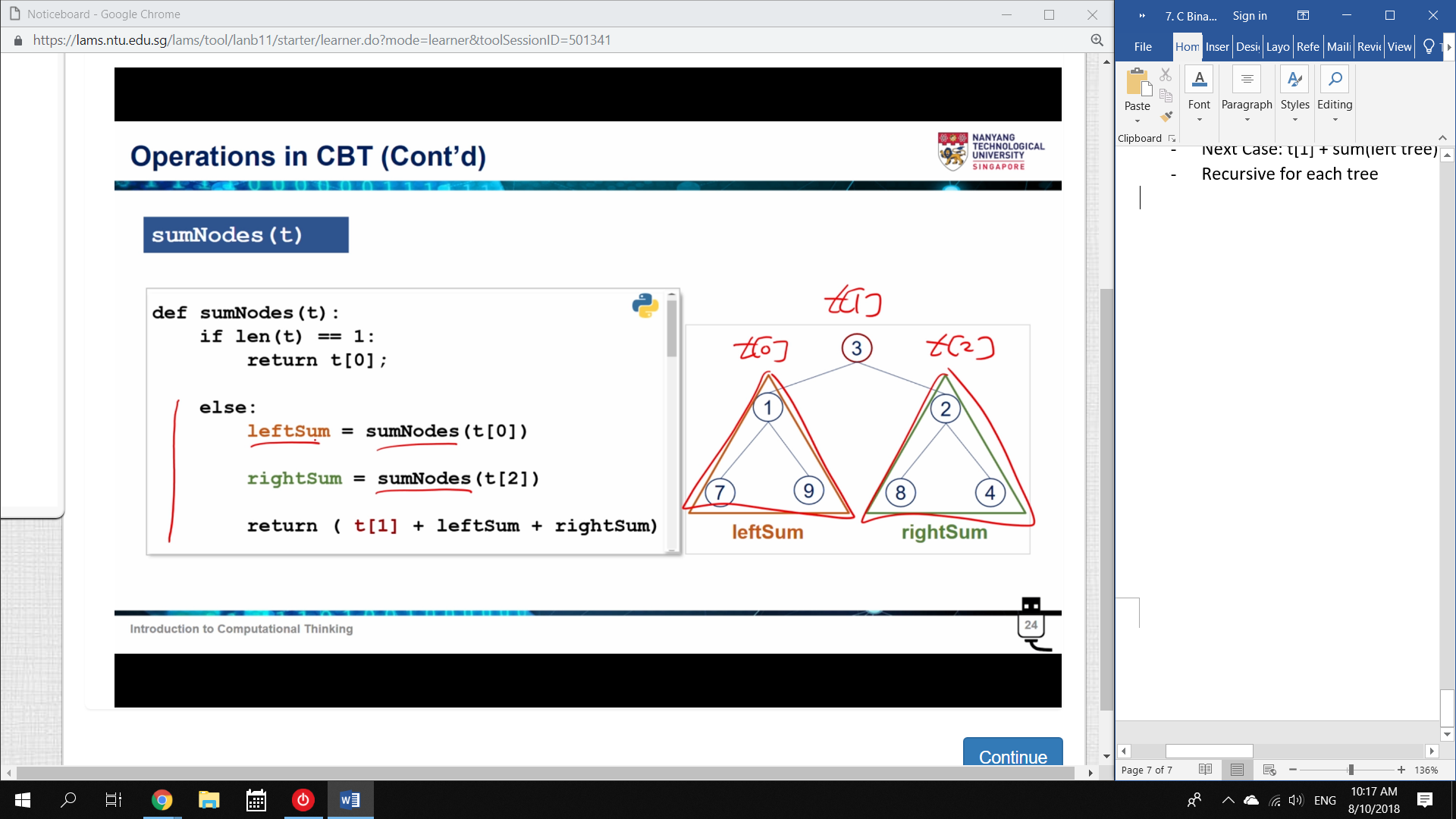
* Base case = 1
* Next case = 1 + (left tree) + (right tree)
* Recursive for each tree



sumNodes(t):

Decompose:

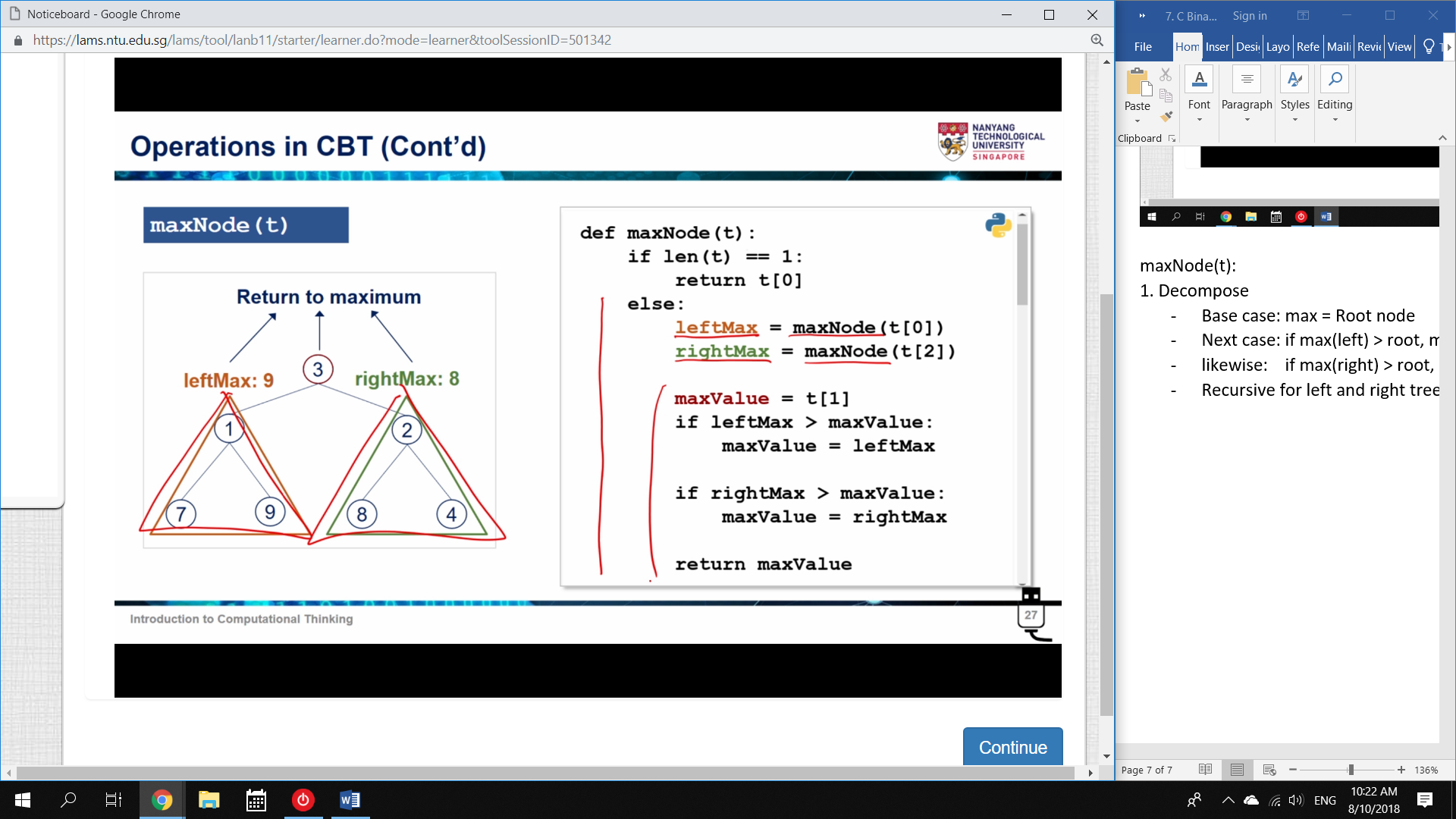
* Base Case: t[1]
* Next Case: t[1] + sum(left tree) + sum(right tree)
* Recursive for each tree



maxNode(t):

1. Decompose

* Base case: max = Root node
* Next case: if max(left) > root, max(left) = max
* likewise: if max(right) > root, max(right) = max
* Recursive for left and right trees.



minNode(t): Same as maxNode(t)

Note: The skill to learn is

* Identify the base case
* Find code that will solve the next largest step using the Base case

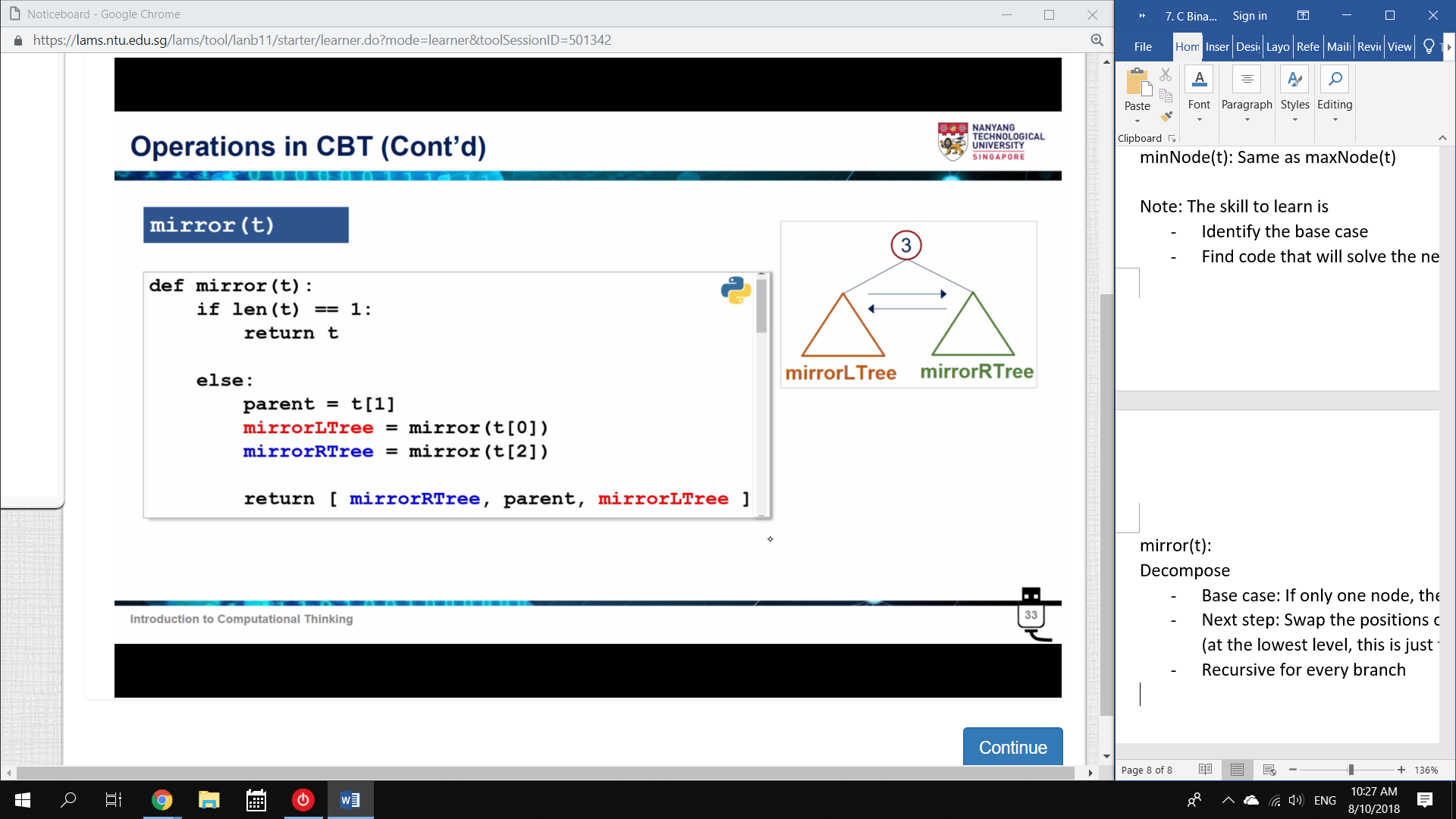
mirror(t):

Decompose

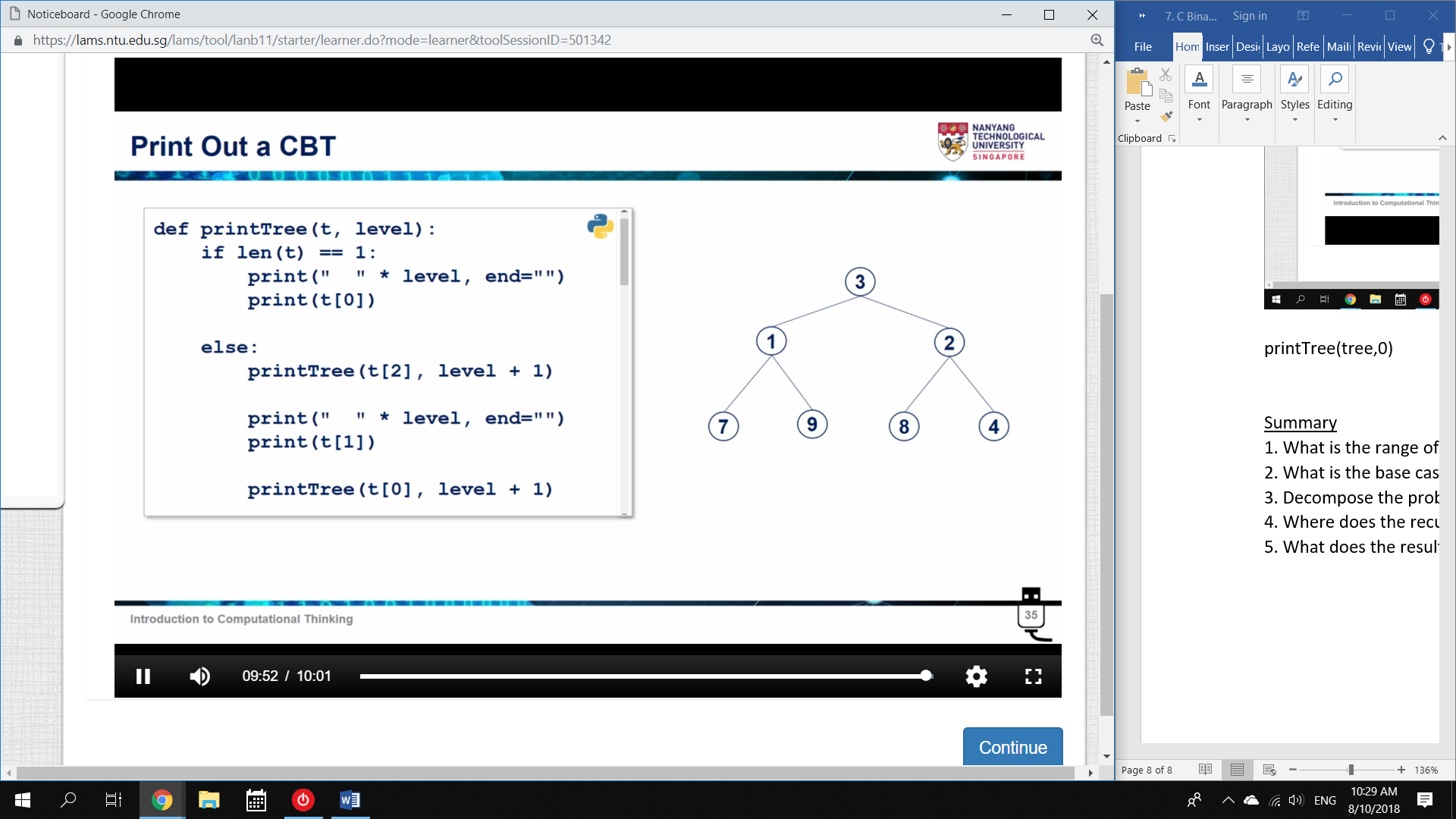
* Base case: If only one node, then mirror(t) = (t)
* Next step: Swap the positions of two subtrees

(at the lowest level, this is just two nodes)

* Recursive for every branch



printTree(tree,0)



1. Base case: [1]

* if len(t) ==1, print t[0] 🡪 the single content of t

2. Next Step: [[2],1,[3]]

* print t[2] 🡪 but indent it to show level
* print t[1]
* print t[0] 🡪 but indent it to show level

3. Recursion:

* else: function(lefttree) 🡪 in this case, t[0]

print t[1]

function(righttree) 🡪 in this case, t[2]

4. Result:

* function(t[0], +1 indent)
* t[1]
* function(t[2], +1 indent)

Summary

0. What is the range of inputs?

1. What is the base case?

2. Decompose the problem. What is the smallest next step?

3. Where does the recursion happen?

4. What does the result look like?