**Finals Reference Sheet – ICT**

**1. Flowchart, Pseudocode, Data Types**

Symbols:

Oval – Start and End

Parallelogram – Input and Output

Rectange – Process (Addition)

Diamond – Decision (leading to two flow lines)

Flow Lines – Next step

Pseudocode common words

IF, ELSE, WHILE

READ, PRINT, INIT, COMPUTE, ADD, SUBTRACT

* One statement per line
* Capitalize keywords
* Indent to show Hierarchy
* End multiline structures
* Independent of programming language

Algorithm Design

Language-independent.

Cannot be understood by Computers.

1. Unambiguous

2. Clear and precise

3. Sequence

4. Branching and Looping

5. Termination

Identifiers in Python

a-z

A-Z

‘\_’ underscore

0-9 numbers

52 character + 10 numbers + 1 underscore

52 + 10 + 1 = 63

Python Characteristics

Duck-typing: If the features of the object are suitable, it is suitable.

No variable declaration

Type of Value determines Type of Variable.

Operators

% 🡪 Modulus

Eg. Today is Tuesday. After 53 days, what day is it?

53 % 7 = 4

Tuesday + 4 = Saturday

// 🡪 Floor Division

\*\* 🡪 Exponent

Order of Operations

()

\*\*

-(x)

\*/%

+ -

**2. Loop**

Less tedious.

Accommodate different users.

Four Steps:

Initialize: Set a starting variable.

Test: If True, loop

Loop Body: What gets looped

Update: Should loop continue?

Two Types

Counter-controlled: For

* ends at count

Sentinel-controlled: While

* ends at value change

Range functions

for I in [0,1,2,3,4,5]

for I in range(6)

range(start, end, step)

* end is not included

sum of range(1,5,1) is not the same as 🡪 (1 + 2 + 3 + 4)

sum of range(5,1,-1) 🡪 (5 + 4 + 3 + 2)

For, else, break, continue

For: Repeat Loop for each Iteration

Else: Branching Option

Break: Exit loop now, skip Else (even if in same indent)

Continue: Go to top of loop now

Prime or Not Prime

1. Divide k by all numbers 2 < n < k/2 🡪 k divide n > k/2 will never be whole.

2. If the remainder is 0, then n is a factor of k

k = int(input(“Please enter an integer: “))

for n in range(2, int(k/2)):

if k%n == 0:

isprime == “no”

break

else:

isprime = “yes” *loop is not Exited*

**3. Abstraction**

Benefits of Abstraction:

* Simplifies
* Identify Important
* Manage Complex

Data Structures are Abstractions in Data.

Data Structures + Algorithms = Programs

Using Backslash

Backslash is the escape character. It is used to remove functionality of some things, like quote marks.

It is also used to create whitespace characters: \t \n \r

To use quote marks in a string:

‘Mike**\’**s book’

To start a new line:

'apple**\n**orange'

String Index

H E L L O W O R L D

0 1 2 3 4 5 6 7 8 9 10

… -3 -2 -1

mystr[1:4] = ELL (remember that the last number is not included)

mystr[-3:-5] = ‘’ (Needs step = -1 to go backwards, i.e [-3:-5:-1] 🡪 R O

mystr[3:-3] = LO WO (combined numbering systems)

mystr[::2] = HLOWRD (step)

String Commands

astr = “string to copy”

newstr = ‘’.join(astr) (This obtains a list of characters joined by ‘’, or nothing)

revstr = astr[::-1] 🡪 ypoc ot gnirts (this reverses the string)

*note that python auto-reverses the start/end*

failrev = astr[0:14:-1] 🡪 ‘’ (this fails because the start/end are not reversed)

astr.find(‘g’) 🡪 5 (index of character. If not present, **returns -1)**

Code to represent characters

ASCII

* 8 bit (256 characters)
* limited

Unicode

* 16 bit (65636 characters)
* 17 planes of character

ASCII

**Lowercase** > Uppercase > Numbers

* Lowercase has a higher ASCII code

ord(‘a’) = 97 Ordinal: Number of a object

chr(97) = ‘a’ Character: Letter of a number

This can be used for encryption.

code = ord(‘a’)

chr(code + 1) (a beomes b)

Strings

Immutable:

str = ‘spam’

str[1] = L 🡪 Error

newstr = str[:1] + ‘L’ + str[2:]

newstr = ‘sLam’

Lists

list(‘abc’) 🡪 [‘a’, ‘b’, ‘c’]

List of lists:

myLst = [‘a’, [1,2,3], ‘a’]

myLst[1][0] 🡪 1

len(myLst) 🡪 [a,[1,2,3],a] 🡪 3 len only takes top level

max([‘a’, 1, ‘B’]) 🡪 Error

sum([‘a’, 1, ‘B’]) 🡪 Error

List Commands

mylist.append(e) 🡪 Append ‘e’ to the back

mylist.extend(L) 🡪 Append elements in ‘L’ to the back

mylist.pop(4) 🡪 remove an index from the list (default is the last, -1)

mylist.insert(4,e) 🡪 insert e at index [4]

mylist.remove(e) 🡪 remove the first e

mylist.sort() 🡪 sort list

mylist.reverse() 🡪 reverse list

list.extend([8,9]) 🡪 [6,5,5, 8,9]

list.append([8,9]) 🡪 [6,5,5, [8,9]] (A list is appended, whereas extend just adds elements)

lst = [0, ‘b’]

lst.insert(0,1) 🡪 [1,0,’b’] (insert puts before that index)

lst.insert(-1,’a’) 🡪 [1,0,’a’,’b’] (BEFORE the index. -1 does NOT mean element is now at -1)

lst.insert(10,’c’) 🡪 [1,0,’a’,’b’,’c’] (setting a high number is fine.)

Sorting a string: Convert to List

string = ‘xyzabc’

lst = list(string)

lst.sort()

string = ‘’.join(lst) 🡪 ‘abcxyz’

Return

Returns the value of Functions.

def adding(a,b)

return a + b

Some functions do not return anything.

lst = [4,7,1,2]

lst.sort() (sorts the list)

lst 🡪 [1,2,4,7]

lst = lst.sort() (assigns NOTHING to lst. This is bad.)

Split

“This is a test”.split() 🡪 [“This”, “is”, “a”, “test”] Split returns a list.

Tuple

Immutable list. (avoid accidents)

Tuple commands:

tuple[3]

tuple[3:5]

len()

print()

The Comma creates the Tuple:

mytuple = 1,2 🡪 (1,2)

mytuple = 1, 🡪 (1)

myint = 1 🡪 1 (not tuple)

Fast List Iteration

[x + y for x in range(5) for y in range (4)] (x+y for x=1 for y in range(4),…)

[

1+1, 1+2, 1+3,

2+1, 2+2, 2+3,

3+1, 3+2, 3+3,

4+1, 4+2, 4+3

]

Dictionary

my\_dict[‘bill’] = 25 (creates a key-value pair)

A key-value pair can be deleted at any time.

del my\_dict[‘bill’]

Deleting my\_dict[‘bill’] deletes the whole pair.

Dictionary Commands

.keys() 🡪 ‘bill’

.values() 🡪 25

.items() 🡪 ‘bill’: 25

.clear() 🡪 empties dictionary

.update(dict\_2) 🡪 adds all contents of dict\_2

All common functions (eg. len, sort) check KEYS unless otherwise specified.

Functions

Invoke a function:

myfunction(argument)

Defining a function:

def myfunction(parameter) 🡪 same thing, just different name

**Functions should do one thing at a time.**

**This makes them reusable.**

Procedures:

Functions that do not return results.

They perform actions (modify, print, store)

**4. Decomposition and Recursion**

Recursive functions: A function that invokes itself.

1. Parameters What are your Values?

2. Base Case What is the smallest subproblem?

3. Subproblems How do you combine base cases?

Eg. Reverse a String:

‘abcdef’

Base case: ‘a’ just print a.

Subproblems:

‘bc’ 🡪 ‘cb’ move [0] to the back

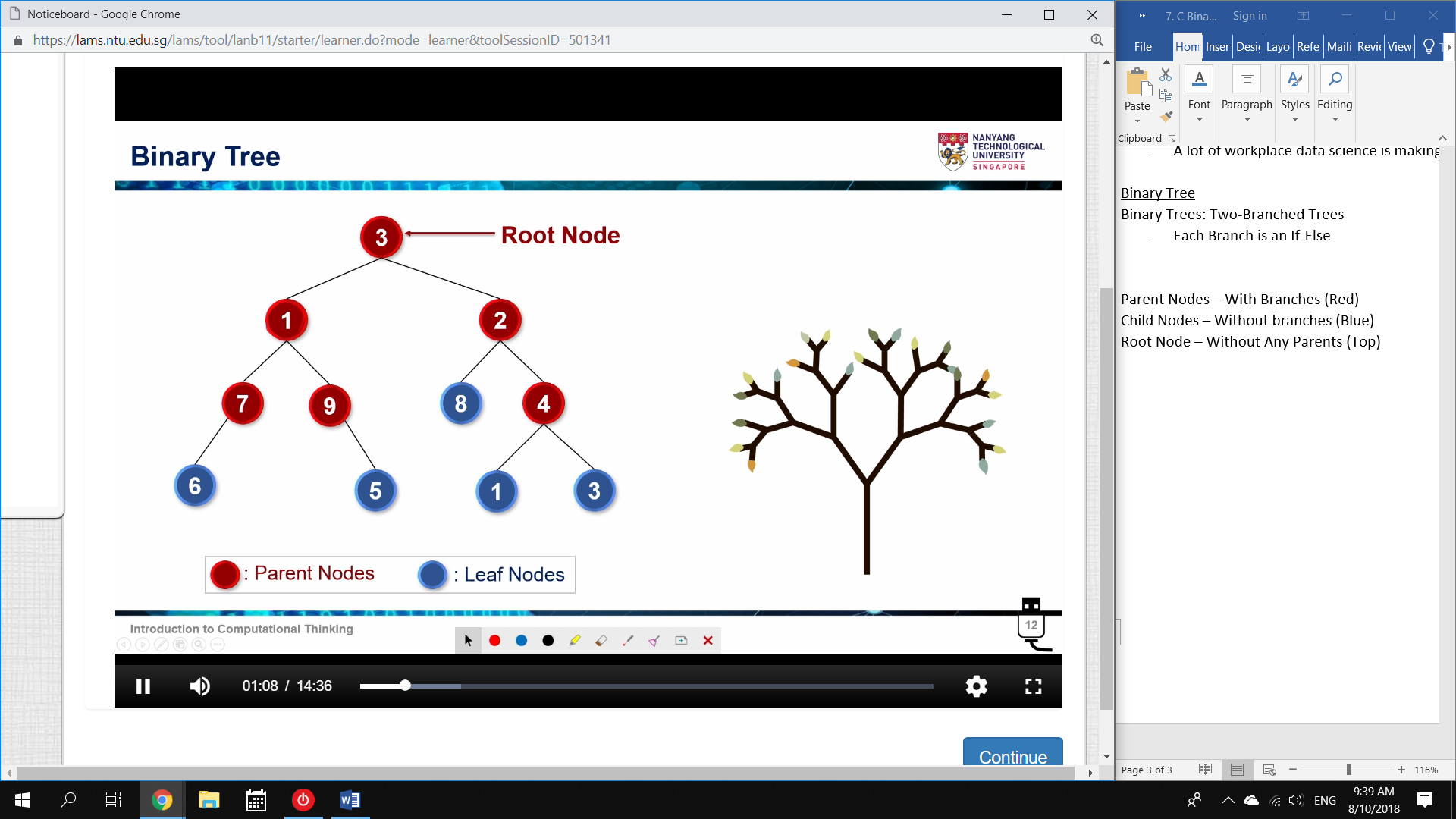
‘abc’ 🡪 ‘cba’ get ‘acb’, then move [0] to the back

Therefore:

[*recursive function*], then move [0] to the back.

Binary Trees

Each Branch is an IF-ELSE statement.



**Complete** Binary Tree: Every parent has exactly 2 Nodes.

[ **[ [6],7,[]], 1, [[],9,[5] ]**, 3, **[ [8], 2, [[1],4,[3]] ]** ]

Operations with CBT

**def numOfNodes(t):**

**if len(t) == 1:**

**return 1;**

**elif len(t) == 0:**

**return 0;**

**else:**

**numLeft = numOfNodes(t[0])**

**numRight = numOfNodes(t[2])**

**return (numLeft + 1 + numRight)**

**def maxNode(t):**

**if len(t) == 1:**

**return t[0]**

**elif len(t) == 0:**

**return –(math.inf)**

**else:**

**leftMax = maxNode(t[0])**

**rightMax = maxNode(t[2])**

**maxValue = t[1]**

**if leftMax > maxValue:**

**maxValue = leftMax**

**if rightMax > maxValue:**

**maxValue = rightMax**

**return maxValue**

**def mirror(t):**

**if len(t) == 1 or len(t) == 0:**

**return t**

**else:**

**parent = t[1]**

**mirrorLTree = mirror(t[0])**

**mirrorRTree = mirror(t[2])**

**return [mirrorRTree, parent, mirrorLTree]**

**def printTree:**

**if len(t) == 1:**

**print(“ “ \* level, end = “”)**

**print(t[0])**

**elif len(t) == 0:**

**print(“”)**

**else:**

**printTree(t[2], level +1)**

**print(“ “ \* level, end = “”)**

**print(t[1])**

**printTree(t[0], level+1)**

**5. Pattern Recognition**

1. Find common elements and differences.

2. Describe a pattern

3. Make a prediction

eg1.

(1) + (1/2) + (1/3) + … + (1/10)

it1 = 1

it2 = 1/2

…

It10 = 1/10

**for i in range(1,n+1)**

**result += 1/i**

eg.2

(0/1) – (1/2) + (2/3) – (3/4) + (4/5) – (5/6) + (6/7) – (7/8) + (8/9) – (9/10)

**for i in range(1,n+1):**

**result += (-1\*\*i)((i-1)/i)**

**6. Algorithm Design**

Sorting:

BubbleSort – Swaps items with each pass

MergeSort – Splits and recombines recursively

BubbleSort

[1, 2, 3, 4, …, n-1, n]

There are n items to sort.

There are n-1 pairs to swap.

Is 1 larger than 2?

YES 🡪 Swap

NO 🡪 Don’t swap

After the first pass, the largest value is always at the bottom.

There are n-2 pairs left to swap.

**def bubblesort(lst):**

**for passnum in range(len(lst) – 1):** 🡪 n-1 passes

**swapped = False** 🡪 If no swaps occur, end

**for I in range(len(lst) – passnum – 1):** 🡪 n-1 – number of already sorted

**if lst[i] > lst[i+1]:**

**temp = lst[i]**

**lst[i] = lst[i+1]**

**lst[i+1] = temp**

**swapped = True**

**print(“pass”, passnum+1, “:”, lst)** 🡪 pass number, and current list

**if not swapped:**

**break**

Mergesort

[1, 2, 3, 4, …, n-1, n]

Keep splitting the list in half [], []

Base case is one element.

Compare two base cases, and place the smaller in a list.

Place the second at the back.

Recursive case is a sorted list.

Compare the First Element of two sorted lists, and place the smaller in a list.

Repeat until one list is empty.

Extend the second at the back. (since its already sorted)

**def mergesort(lst):**

**lst\_len = len(lst)**

**if lst\_len < 2: #Base case is one element**

**return lst**

**left\_lst = lst[ : lst\_len // 2] #Split the list in half**

**right\_lst = lst[lst\_len//2 : ]**

**left\_lst = mergesort(left\_lst) #Keep splitting the list in half until base**

**right\_lst = mergesort(right\_lst)**

**return merge(left\_lst, right\_lst) #Compare first element, new list, extend**

**def merge(left\_lst, right\_lst)**

**result\_list = []**

**while left\_lst and right\_lst:**

**if left\_lst[0] < right\_lst[0]:**

**result\_list.append (left\_lst[0])**

**left\_lst.pop(0)**

**elif left\_lst[0] > right\_lst[0]:**

**result\_list.append(right\_lst[0])**

**right\_lst.pop(0)**

**else:**

**result\_list.append(left\_lst[0])**

**result\_list.append(right\_lst[0])**

**left\_lst.pop(0)**

**right\_lst.pop(0)**

**if left\_lst:**

**result\_list.extend(left\_lst)**

**elif right\_lst:**

**result\_list.extend(right\_lst)**

**return result\_list**

TIMSORT

TIMSORT is a hybrid of Mergesort and Insertion sort (like bubblesort, but reversed. It stops faster)

TIMSORT is used to:

1. Find subsets of ordered Data (subsets are called RUNS in TIMSORT. Funny, I hate running)

2. Merge a subset with existing subsets

**7. Searching**

Linear Search: Iterates over all items until found.

* If alphabetically sorted, then you can just check a single letter (subset)

Binary Search: Starts in Middle. Checks if larger/smaller, then only considers that Half.

* Can only be done with values. Needs sorting.

Worst Case (Linear): O(n)

* ie., it takes n tries

Worst Case (Binary): O(log n)

* ie., it takes log(n) tries.

def linearsearch(lst):

if i in lst:

print(True)

**def binarysearch(lst, target, low = 0, high = None) :**

**if high == None:**

**high = len(items) – 1 #base case**

**if low > high:**

**return False #**

**mid = (low + high) // 2**

**if target == lst[mid]:**

**return True**

**elif target > lst[mid]:**

**return binarysearch(lst, target, low = (mid + 1), high = high)**

**else:**

**return binarysearch(lst, target, low = low, high = (mid – 1))**

.index(argument) 🡪 returns the index of the argument in a list

**8. Time Complexity**

Two ways to compare:

1. Time taken

2. Number of steps

Time taken can depend on:

* Computer Speed
* Algorithm
* Pre-compute lookup Table
* Unrolling a Loop

Therefore, Number of steps is much better.

Only consider the worst case scenario.

Not actually lines of code. Each line consists of multiple steps.

* We use it for representation

Big O Notation

Big O Gives:

* The Upper Bound on Execution Time as Input Data Size grows.

via number of execution steps

1. Constant Complexity: O(1) [Common: Prelookup Table]

* Same time for all inputs

**def lookup(position)**

**return lookupTable(position)**

T(n) = 1 🡪 O(f(n)) = O(1)

T(n) is the literal steps number (eg 4n + 2)

f(n) is the asymptotic behaviour (as data approaches high, eg. 4n)

O is the big O notation: the order of growth. (eg. n)

2. Logarithmic Complexity: O(log n) [Common: Binary Search]

* Time increases less rapidly as data size increases

**def binarySearch()**

**…**

**low = mid + 1**

**high = mid - 1 #basically, if the data gets reduced quickly**

f(n) = log2(len(List))

O(f(n)) = O(log2(len(List)) = O(log n)

3. Linear Complexity: O(n) [Common: FOR loop]

* Time increases linearly with data size

**def linearsearch()**

**…**

**for i in range() #basically, if number of steps correlates with number of data**

f(n) = len(List)

O(f(n)) = O(len(List)) = O(n)

Write out the number of steps per line (WORST CASE).

4. Polynomial Complexity: O(n2) [Common: Nested Loop]

* Time increases quickly as data size increases

**def check\_2\_lists()**

**…**

**for I in list\_1:**

**for j in list\_2:**

**…**

fin(n) = len(List\_2) = n

fout(n) = len(List\_1) = n

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

5. Exponential Complexity: O(kn) [Common: Recursive Call]

* Time grows astronomically fast due to Recursive call

**def recursiveFib()**

**…**

**elif (n > 1):**

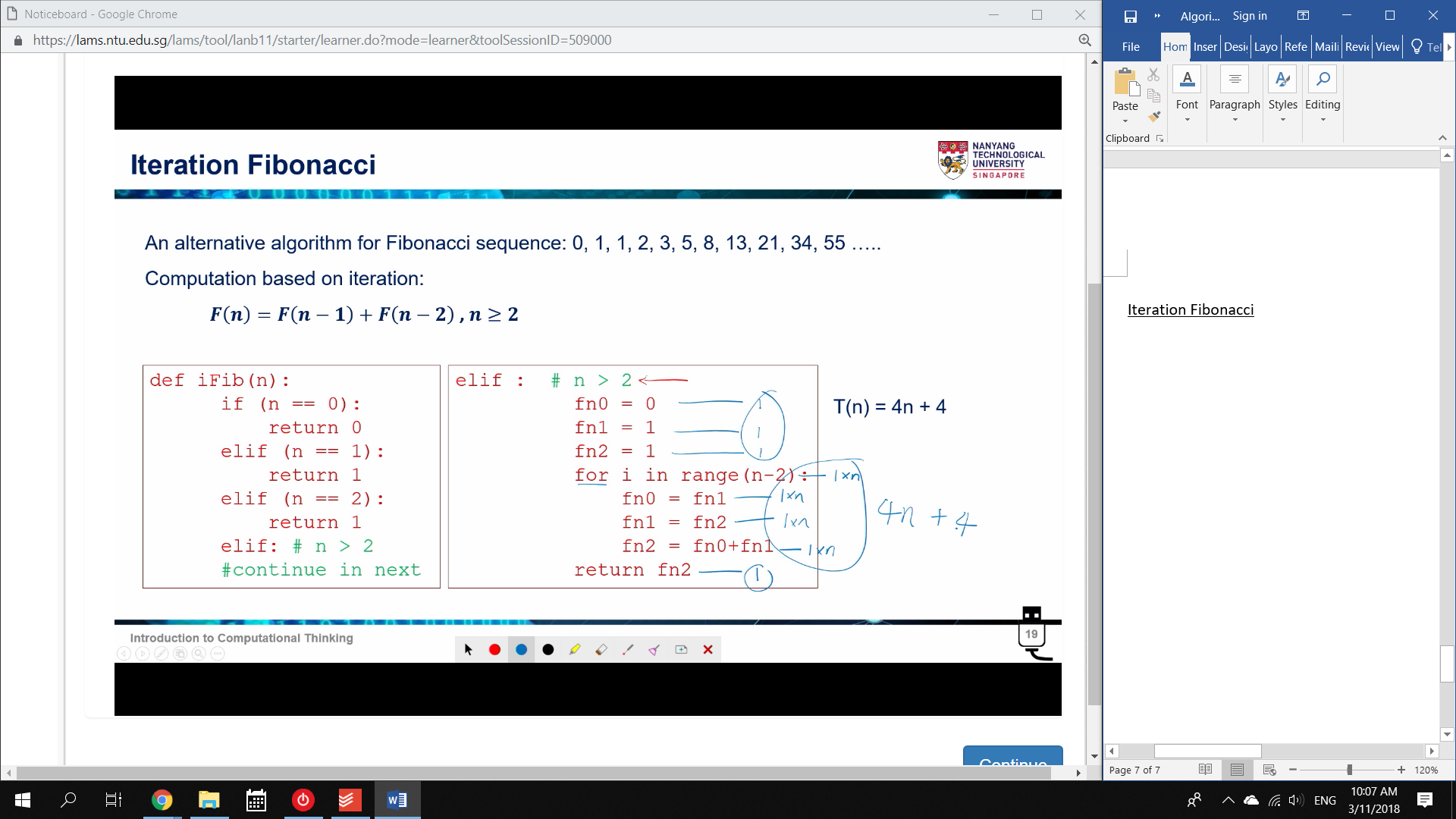
**return (recursiveFib(n-1) + recursiveFib(n-2))**

**…**

f(n) = 2n

O(f(n)) = O(2n)

Iteration Fibonacci



Obviously, Faster.

Additional Notes

O(1): Pre-lookup tables. Not always Practical

O(log n): Pre-sort data.

**Full List of Functions**

**all()** returns true if all the conditions in the arguments are true(acts like AND

function in math

**any()** returns true if any of the conditions mentioned in the arguments is true

acts like OR function in math

**randint(1,5)**

output will be 1 - 5

**list.index('abc')**

returns the index value in which 'abc' occurs

To use quote marks in a string:

‘Mike**\’**s book’

To start a new line:

'apple**\n**orange'

**astr.find(‘g’)** 🡪 5 (index of character. If not present, **returns -1)**

**astr.replace(‘a’,’z’) 🡪** replaces all ‘a’ character with ‘z’ in the string

**ord(‘a’)** 🡪 97 Ordinal: Number of a object

**chr(97)** 🡪 ‘a’ Character: Letter of a number

List Commands

**max(list) and min(list)** 🡪 max and min work with Unicode.

**mylist.append(e)** 🡪 Append ‘e’ to the back

**mylist.extent(L)**  🡪 Append list ‘L’ to the back

**mylist.pop(4)**  🡪 remove an index from the list (default is the last, -1)

**mylist.insert(4,e)** 🡪 insert e at index [4]

**mylist.remove(e)** 🡪 remove the first e

**mylist.sort()**  🡪 sort list by unicode

**mylist.reverse()**  🡪 reverse list

**mylist.split()**

Separates values and returns a list.

set(list)

deletes all the duplicate values present in  
 the list.

**sorted(list,reverse=True/False)**

**sorted** can take any iterable. (unlike .sort, which only takes lists)

**key=str.lower** will sort lowercase first.

**reverse=True** will sort from lowest to Highest. Upper before Lower.

Tuple Commands

mytuple = 1,2 🡪 (1,2)

mytuple = 1, 🡪 (1)

myint = 1 🡪 1 (not tuple)

Dictionary Commands

.clear() 🡪 empties dictionary

.update(dict\_2) 🡪 adds all contents of dict\_2

.keys() 🡪 ‘bill’

.values() 🡪 25

.items() 🡪 ‘bill’: 25

