National University of Singapore
School of Computing
IT5003: Data Structure and Algorithm
Semester I, 2019/2020

Tutorial x Lab 1
Complexity and Sorting

General idea: Tutorial type questions will be listed first in this document. Some tutorial question will then be implemented in the lab portion (later part of this document). Lab exercise cases to be submitted on Coursemology will be marked [SUBMISSION] at the front.

1. [Complexity] Rearrange the following 10 terms in ascending order of their Big-Oh time complexity:

$$4N^2, \log_5(N), 20N, N^{2.5}, N^N, 3^N,$$
$$2^N, 2^{N+1}, 2^{2N}$$

Do note that some of the above terms have equal Big-Oh complexity. Remember that in Big-Oh notation, we only care about the dominating term of the function, without its coefficient. As N gets very large, the effect of other terms on the volume of code executed becomes insignificant.

You should write down some simple justification for the non-obvious cases to have a better understanding.

2. [Not very complex complexity] Give the big-O for following code fragments

a.	`for i in range(2*n):` `  <3 operations>`

b.	`for i in range (n):`
	`  for j in range(n):`
	`  <2 operations>`

c.	`for i in range (n):`
	`  for j in range(3):`
	`  <5 operations>`

d.	`for i in range (n):`
	`  for j in range(n – 1, i, -1):`
	`  <4 operations>`

e.	`i = 1`
	`j = 0`
	`while j < n:`
	`  if i % n == 0:`
	`  j += 1`
	`  <1 operation>`
	`  i += 1`

f.	`for i in range (n):`
	`  if i % 2 == 0:`
	`  for j in range(n):`
	`  <3 operations>`
	`  else:`
	`  <7 operations>`

3. You are given the following array: `1285, 5a, 150, 4746, 602, 5b` and `8356`. Subscripts 'a' and 'b' are used to just distinguish the two 5's. Trace each of the following sorting algorithms as it sorts the above array into **ascending** order.

    a. Selection sort [Self-Check]
    b. Bubble sort [Self-Check]
    c. Insertion sort [Self-Check]
    d. Merge-sort [Discuss in tutorial]
    e. Quick-sort [Discuss in tutorial]

4. [Bubble Sort Version 3.0] Let us see how bubble sort can be further improved.

    a. [What's the issue?] Try sorting an array like {2, 3, 4, 5, 1}. How many outer-loop iteration do we need? Identify the issue with the standard bubble sort algorithm.
    b. [Solve the issue] Solve the issue posed by (a). Hint: It is like bubble sort with a twist….
    c. [Analyzing the change] Did we improve the big-O of bubble sort?

5. [Sorting is general] For simplicity, sorting is almost always taught using an integer array. However, it should be clear that the sorting algorithms can be easily generalized. Let us take the **insertion sort** code as a case study in this question.
    a. [What to change?] Identify all necessary changes for the insertion sort code if we need to sort a different type of array (e.g. an array of student records / complex numbers, etc). Whenever possible, focus more on the higher level requirement (**"what kind of operation is needed?**") rather than low level details ("**how do I write this in Python?**")

    b. [Actual change] Using your findings in (a), change the insertion sort to work on an array of **StockItem** object as defined below:

```
class StockItem:
    def __init__(self, name, barcode, price, stock):
        self._name = name
        self._barcode = barcode
        self._price = price
        self._stock = stock
```

We want to sort the StockItem in ascending order according to the following rule:

- Item with cheaper price are placed in front
- Item with same price are ordered in reverse order of their stock (i.e. more stock = in front)

In Python, programmer can change the meaning of comparison operators like "<", "==", ">=" by **operator overloading**. You need to provide the code for the corresponding methods as follows:

Operator	Method
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`
==	`__eq__(self, other)`
!=	`__ne__(self, other)`

Add the relevant method to the class **`StockItem`**, so that the **`insertionSort`** can work **without modification**. You can use the given **GeneralSorting.py** file to try out.

6. [Application of Sort] Assume that you are given two sorted arrays A and B, each of them containing **n** numbers. Give an efficient algorithm for computing one of the two medians of all **2n** numbers and analyze its running time. [Hint: Merging is not the best approach]

~~~ Lab Question ~~~

1. Using the provided Python code "**General Sorting.py**", implement the changes as discussed in tutorial question 5b.

2. [Submission] Using the provided Python Code "**Cocktail.py**", implement two functions:

Function 1 `def bubbleSort( array, nPos, nPass )`
Modified the standard bubble sort such that: - The bubble sort only sort up to and includes the item at [nPos] - The bubble sort will stop after [nPass] outer iteration, i.e. instead of complete N passes by default.

Sample Test Setup: `array = [283, 1560, 2150, 1061, 123, 2154, 222, 4]`

Test 1:
`bubbleSort(array, len(array)-1, len(array))`
#this will sort the whole array, i.e. like the original bubble sort
`[4, 123, 222, 283, 1061, 1560, 2150, 2154]`

Test 2:
`bubbleSort(array, len(array)-1, 3)`
#this will sort the whole array, but only for 3 passes
`[283, 123, 1061, 222, 4, 1560, 2150, 2154]`

Test 3:
`bubbleSort(array, 3, 2)`
#this will sort the first four items and only for 2 passes
`[283, 1061, 1560, 2150, 123, 2154, 222, 4]`

The given source code has a few more tests for you to try out.

Function 2

```
def cocktailSort( array, nPos, nPass )
```

Implement cocktail sort as discussed in tutorial question 4. The sorting function will have similar modifications as function 1 such that:

- The sort only sort up to and includes the item at [nPos]
- The sort will stop after [nPass] outer iteration, i.e. instead of complete N passes by default. **Note: one pass consists of both the forward and the backward swapping loops. Note$_2$: We will assume the backward swap always start from position 0 for simplicity.**

Sample Test Setup:

```
array = [283, 1560, 2150, 1061, 123, 2154, 222, 4]
```

Test 1:

```
cocktailSort(array, len(array)-1, 1)
#Perform one pass of cocktail sort on the whole array
[4, 283, 1560, 1061, 123, 2150, 222, 2154]
#Notice that "4" is move to the front!
```

Test 2:

```
cocktailSort(array, len(array)-1, 1)
#Perform two passes of cocktail sort on the whole array
[4, 123, 283, 1061, 222, 1560, 2150, 2154]
```