National University of Singapore
School of Computing

IT5003: Data Structure and Algorithm

Semester I, 2019/2020

Tutorial x Lab 1 Selected Solution
Complexity and Sorting

About selected solution: We will released solution to selected question only. Mainly for questions that are hard / information heavy. For the rest of questions, you should take notes during the discussion.

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?

Ans:

a) It takes a long time for a small item to move "backward". In the example, it take 4 outer-loop iteration for the "1" to move to its correct location, i.e. it can move only 1 position per outer-loop iteration.

b) Introduce a “backward” swapping loop in each iteration as shown in the pseudo code:

```
left = 0
right = n-1

    while left < right:

    for idx = left to right:
        swap a[idx] with a[idx+1] if a[idx] > a[idx+1]
    right -= 1

    if left >= right:
        end    # to handle array with odd size

    #the additional “backward” swapping loop
    for idx = right downto left:
        swap a[idx] with a[idx-1] if a[idx] < a[idx-1]
    left += 1
```

This variation of Bubble Sort is known as "**Cocktail Sort**" as it resembles shaking a cocktail mixer.

c) Cocktail sort performs better for almost-sorted sequence. However, in the average and worst case, the complexity is still $O(n^2)$.

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 / double values / strings 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.

Ans:

a. Changes highlighted below with comments.

```
def insertionSort(array):
    n = len(array)
    for i in range (1, n):
        next = array[i]
        j = i-1
        while j >= 0 and array[j] > next:
            swapElement(array, j+1, j)
            j = j-1
        array[j+1] = next;
```

3 types of change:

i. **Assignment / Copying related**: all assignments between array elements / *`next`* variable. Need to check whether items can be copied using simple "=" in the specific programming language. In reference-based language like Python, the "=" will be able to perform a shallow copy without change.

ii. **Comparison related**: i.e. (*`a[j] > next`*). Need to check how to compare two elements. Common approaches includes: ability to provide a *`compare()`* function/interface (e.g. C/C++, Java) or to provide ability to overload the operators <, <=, >, >=, == (e.g. C++, Python, etc).

iii. **Data type related**: In statically typed language (e.g. C/C++, Java), the declaration of the array, "next" will have to be modified. This does not concern Python as it is a **dynamically typed** language.

b. <Omitted>

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]

Ans:

0	……	n-1	0	……	n-1
A[0]			**B[0]**		

- If we **merge** the two arrays, the medians will be A[n-1] and B[0] Running time is O(n).

- But we can observe that, if the median is the element A[m], then B[n-m-1] ≤ A[m] ≤ B[nm]

......	**A[m]**	......	...	**B[n-m-1]**	**B[n-m]**	...
← m →		← n-m-1 →	← n-m →			

Setting **m = n/2** and check **A[m]** against both **B[n-m-1]** and **B[n-m]**. There are three cases to consider:

1. **B[n-m-1] ≤ A[m] ≤ B[n-m]:** As described above, median found!

2. **B[n-m-1] > A[m]:**
   There are less than n elements that have lower value than A[m]. Therefore, the median has higher value than A[m]. We continue looking for the median among the elements of A that have higher indices than m.

3. **A[m] > B[n-m]:**
   Then the median should have a lower value than A[m]. We continue looking for it among the elements of A that have indices lower than m.

With the above observations, we use ***binary search-based algorithm*** on array A to find the median. If the search fails, it means that the median is stored in array B. So we use the exact same algorithm with the roles of A and B exchanged to search for it in B. The algorithm is guaranteed to find the median, since it searches both arrays for it. The running time is O(lgn) since we basically do (at most) 2 binary searches in A and B.