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

# Tutorial x Lab 2 Selected Solution
# Sorting II & Abstract Data Type

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.

1. [Sorting – Leftover from TL01-Q3] 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.
   e. Quick-sort [Discuss in tutorial]
   f. Radix-sort [Discuss in tutorial]

2. Using time complexity of **merge sort** and **radix sort**, suggest a usage scenario where merge sort is preferred. Then suggest a second scenario where radix sort is preferred.

**Ans:**

Essentially compare the term lg N (merge sort) vs d (radix sort). The former is dependent on the input size, while the latter is not.

| N | lg N |
|---|---|
| 128 | 7 |
| 1,024 | 10 |
| 65,536 | 16 |
| 104,856 | 20 |
| 1,073,741,824 | 30 |

As an example: Merge sort is preferred when a list has less than 100 values, but each value is 10 digits (typical 32-bit range).

Another example: Radix sort is preferred when a list has 65,000 values and each value is a 7 digit value (e.g. NRIC number without the prefix and suffix).

Note that the above comparison only takes in time complexity as considerations. There are other matters that should be considered.

3. As discussed in lecture, quick sort can degenerate into worst case if the input is (mostly) sorted (either ascending or descending order). Let's try the following:
    a. Suggest a few ways to make quicksort less susceptible to skewed input.
    b. Suggest how do we make the partition() implementation **user-adaptive**, i.e. can allow the coder to apply idea in (a) easily.

Ans:

a. The key point is the selection of pivot item. Note that the simple quick sort algorithm we discussed is susceptible to **user constructed worst case**, i.e. the user purposely provide the input in a certain fashion to slow down the machine. A few common variants below:
    i. Middle element: Choose (low + high) / 2 as pivot. Note that this solve the (almost) sorted worst case, but you can still construct a worst case input manually.
    ii. Random: Randomly choose an item between [low…high]. Provided the random generator is not known (or not knowable) to the user, this provide simple protection of user constructed worst case.
    iii. Median-of-3: Take the median element between [low], [(low+high)/2] and [high] as the pivot.
b. A simple way is to allow user defined function for the pivot selection (remember higher order functions from IT5001?). For example (in Python):

```
def partition( array, i, j, selectPivot ):
      pivotIdx = selectPivot( array, i, j)
      swapElement(array, i, pivotIdx)

      pivot = array[i]
      middle = i

      for k in range (i+1,j+1):
            if array[k] < pivot:
                  middle = middle + 1
                  swapElement(array, k, middle)
      swapElement(array, i, middle)
      return middle
```

The **selectPivot()** is a user supplied function. It selects a pivot among items [i…j] and return the index. In our case, we choose to swap the selected pivot with [i], which allow the rest of the algorithm remains unchanged.

4. From the example tracing (Q1, part e), we know that quicksort is unstable.
    a. Focusing on the partition algorithm, discuss the cause(s) of this problem. i.e. identify step in the algorithm that can cause unstable sorting.
    b. Suggest a way to make quicksort stable. Discuss briefly the impact on time / space complexity.

Ans:

a. Key observation, when we perform an array element swap, say at index [x] and [y], the swap may break the relative order of items between [x+1…. y-1] (assuming x < y).

During quick sort partitioning, we perform two swaps:

i. When item `a[k] < pivot`.
Example (look at 9a, 9b):

| i | ... | m | m+1 | ... | k | ... | j |
|---|---|---|---|---|---|---|---|
| 7 | …. | ….. | 9a | 9b…. | 5 | ….. | |

Increase **m**, then swap [**m**] with [**k**], increase **k**:

| i | ... | ... | m | m+1…… | ... | k... | j |
|---|---|---|---|---|---|---|---|
| 7 | …. | …. | 5 | 9b…. | 9a | ….. | |

The swap disturbed the relative order of 9a and 9b. (note that 9b can be any way in the [m+2…k-1] region.

ii. At the end, we swap `a[i]` with `a[middle]`

The trace example in Q1(e) is an example of this issue.

b. It is very hard to make the partition stable while **inplace** (i.e. using only the original array). Intuitively, you can see that to solve the issue in (a), we need to **shift the items along the array**, which can potentially cause a $O(k^2)$ time complexity in the worst case for size k array (e.g. shift 1 item, then 2 items, then 3 items, …. $k = k(k+1)/2$ ➔ $O(k^2)$.

So, if we use an extra array B of size k, we can then:

i. Partition from i+1 to… j
ii. keep the leftIdx = i+1, and rightIdx = j
iii. Place item < pivot in the left region at B[leftIdx], increase leftIdx afterwards.
iv. Similarly, place item >= pivot in the right region at B[rightIdx], decrease rightIdx afterwards.

At the end of partitioning, we copy:

i. [0… leftIdx-1] back to A[], starting at [i]
ii. Pivot item into A[leftIdx]
iii. [j….rightIdx+1] back to A[], continuing at leftIdx+1 (note the reverse direction)

Impact on time complexity:

- We incur an addition k items of copying from the extra array B[] back to original A[]. However, the original algorithm is O(k) with k item, so no change (O(k) + O(k) ➔ O(k)).

Impact on space complexity:

- The extra array B[] needs to be accounted for. Since the B[] array is used only in partition() function (i.e. we do not need multiple B[] during recursion), the most we need is at the first level of partition, where B[] is the same size of A[], i.e. an additional of O(N) memory space is required.

Conclusion:

- It is uncommon to implement a stable quicksort due to the memory space requirement since that makes quicksort inferior / the same as merge sort in both time and space complexity.

For question 5 and 6. We explore the idea of **abstract data type**. For each question:

a. Draft a **specification** in Python (i.e. class with empty methods). Focus on the functionalities, the parameters (arguments) for the methods. As the problem is quite open ended, try to draft an **essential set of functionalities.**

b. **Discuss** possible implementation(s). Focus on a) how / what to store by objects of the class to support the specification. Discuss the involved algorithm / data structure only when necessary.

5. We want to support **quadratic equation** in Python. A quadratic equation has form:

$$ax^2 + bx + c = 0$$

where a, b and c are **real numbers** and a != 0.

If you are feeling a bit rusty on this topic 😊, take a look at the Wikipedia page https://en.wikipedia.org/wiki/Quadratic_equation as a quick review.

Discuss the specification and implementation for this **QEquation** ADT.

Ans:

**<Suggested>** Common functionality

a. Instantiate an object. how to create an object of QEquation.
b. Solve the QEquation, i.e. return root(s) of the quadratic equation.
c. Print
d. Etc...

Implementation

Rather straightforward as there are not many alternatives.

Key info is to keep the a, b and c constant. A few common choices:

a. As individual object attributes
b. As a tuple object attribute
c. As a list object attribute

6. We want to support **a deck of playing cards (poker cards)** in Python. A playing card has 4 possible **suits** {Spade, Heart, Club, Diamond} and 13 possible **ranks** {Ace, Two,…., Ten, Jack, Queen, King}. A deck of playing cards has 52 cards (4 suits x 13 ranks).

   The **Deck** ADT should be designed to support simple card game.

Ans:

For simplicity, can assume that there is a simple Card ADT that stores Suit and Rank.

| **<Suggested>** Common functionality |
| --- |
| a. Instantiate a Deck of cards with full 52 cards.<br>b. Shuffle the deck, i.e. mix up the **remaining card** in the deck in random order.<br>c. Take one card from the top.<br>d. Get the number of remaining cards.<br>e. Etc. |

| Implementation |
| --- |
| Python list is probably the best match. (How do we make sure that taking a card from the top of the deck is efficient in terms of time complexity?)<br><br>Python provides a list shuffling built-in function 😊, but it is good to know how to randomize a collection of items in general. |