CS 367 Homework 5

Mingren Shen (mshen32@wisc.edu)

Question 1

Shaker sort is a bi-directional bubble sort. The shaker sort algorithm is:

```
begin = 0, end = A.length-1

// At the beginning of every iteration of this loop, we know that the
// elements in A are in their final sorted positions from A[0] to A[begin-1]
// and from A[end+1] to the end of A.  That means that A[begin] to A[end] are
// still to be sorted.
do
    for i going from begin to end-1
        if A[i] and A[i+1] are out of order, swap them
    end--

    if no swaps occurred during the preceding for loop, the sort is done

    for i going from end to begin+1
        if A[i] and A[i-1]  are out of order, swap them
    begin++
 until no swaps have occurred or begin >= end
```

Part A: What is the **best-case time complexity** of shaker sort? **Briefly explain** the reasoning behind your answer. The best-case behavior would be achieved if the values in the array were(***ordered***).

Answer

best-case time complexity : $\mathcal{O}(N)$

If all values are already ordered, then no swaps have occurred and the program only goes from first element to the last one.

Part B: What is the **worst-case time complexity** of shaker sort? **Briefly explain** the reasoning behind your answer. The worst-case behavior would be achieved if the values in the array were(***reversely ordered***).

Answer

worst-case time complexity : $\mathcal{O}(N^2)$

If all values are reversely ordered, then swaps have occurred every step. So the `i` will change like the following, assuming **N** is the number of items in the array:

- 1st Pass , Left => Right Round , `i` : 0 => N - 2, in sum N - 1
- 1st Pass , Right => Left Round , `i` : N - 2 => 1, in sum N - 2

- 2nd Pass, Left => Right Round , `i` : 1 => N - 3, in sum N - 3
- 2nd Pass, Right => Left Round , `i` : N - 3 => 2, in sum N - 4
- . . .

After the first pass(Left => Right Round and Right => Left Round) the largest item and the smallest item will be in the right position.And so on for other passes. So in total, for reversely ordered array we need $\lfloor \frac{N}{2} \rfloor$ passes. So the total time complexity would be $N - 1 + N - 2 + N - 3 + N - 4 + \ldots\ldots$ and this has the $\mathcal{O}(N^2)$ time complexity.

Question 2

Sometimes you want to sort lots of data, but you only need the smallest (or largest) *K* items instead of all items. For example, maybe you want to give awards to the ten students with the best GPAs. If you have thousands of students, you could waste time sorting all of them just to find the top ten. Perhaps we can *adapt* an existing sorting algorithm so that it works faster if we just want to find and sort the smallest (or largest) *K* items.

Suppose we can start with either insertion sort or selection sort.

Part A: Which one is easier to modify so that it **efficiently** gives us only the smallest **K** items (in sorted order) instead of all items?

Answer

selection sort.

Part B: Describe the changes needed to make this happen.

Answer

Because selection sort picks the smallest element in the unsorted part of the array and then exchanges it with the leftmost unsorted element so the smallest unsorted part of the array is in the right places. And then check the remaining of the unsorted array. So each pass selection sort find the smallest element and put it in the right position.

So we only need to keep an internal counter variable bookkeeping how many passes have done and stop the sorting when it reaches **K** and return the first **K** items of the array. We can avoid to do sorting for the whole array.

Part C: Briefly explain why the other algorithm cannot be modified to do what we want efficiently.

Answer

For insertion sort, you never know the smallest item unless you have completed the sorting. So you have to do all the sorting for the whole array instead of just doing part of the array.

Question 3

Using median-of-three pivot selection, **show the successive changes** that are made as quick-sort partitions the following list of integers. Note you need only show the changes that occur during the the first partitioning, stopping just before the recursive calls.

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 80 | 90 | 50 | 10 | 80 | 70 | 30 | 40 | 70 | 50 | 40 | 20 | 60 |
        ------------------------------------------------------------------
```

Answer

(1) Choose the pivot

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   |[80]| 90 | 50 | 10 | 80 | 70 |[30]| 40 | 70 | 50 | 40 | 20 |[60]|
        ------------------------------------------------------------------
```

Arrange the 3 elements by putting the smallest of the 3 values in A[low], put the largest of the 3 values in A[high], and swap the pivot with the value in A[high-1].

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   |[30]| 90 | 50 | 10 | 80 | 70 |{20}| 40 | 70 | 50 | 40 |[60]|[80]|
        ------------------------------------------------------------------
                                                                pivot
```

(2) initialize

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 | 90 | 50 | 10 | 80 | 70 | 20 | 40 | 70 | 50 | 40 |[60]| 80 |
        ------------------------------------------------------------------
                |                                           |     |
              left                                      right pivot
```

(3) swap values

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 |{40}| 50 | 10 | 80 | 70 | 20 | 40 | 70 | 50 |{90}|[60]| 80 |
        ------------------------------------------------------------------
                |                                           |     |
              left                                      right pivot
```

(4) increment

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 | 40 | 50 | 10 | 80 | 70 | 20 | 40 | 70 | 50 | 90 |[60]| 80 |
        ------------------------------------------------------------------
                              |                        |          |
                             left                    right       pivot
```

(5) swap values

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 | 40 | 50 | 10 |{50}| 70 | 20 | 40 | 70 |{80}| 90 |[60]| 80 |
        ------------------------------------------------------------------
                              |                        |          |
                             left                    right       pivot
```

(6) increment

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 | 40 | 50 | 10 | 50 | 70 | 20 | 40 | 70 | 80 | 90 |[60]| 80 |
        ------------------------------------------------------------------
                                   |          |                   |
                                  left       right               pivot
```

(7)swap values

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 | 40 | 50 | 10 | 50 |{40}| 20 |{70}| 70 | 80 | 90 |[60]| 80 |
        ------------------------------------------------------------------
                                   |          |                   |
                                  left       right               pivot
```

(8) **increment and cross over**

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        ------------------------------------------------------------------
array   | 30 | 40 | 50 | 10 | 50 | 40 | 20 | 70 | 70 | 80 | 90 |[60]| 80 |
        ------------------------------------------------------------------
                                         |    |                   |
                                     right    left              pivot
```

(9) Put the pivot into its final place which is swapping the pivot and right + 1 element (`swap(A, right+1, high-1);`)

```
index   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
        -----------------------------------------------------------------
array   | 30 | 40 | 50 | 10 | 50 | 40 | 20 |{60}| 70 | 80 | 90 |{70}| 80 |
        -----------------------------------------------------------------
```

Finish the the first partitioning.