Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 4

Week: 04

Due: Nov 11, 2019

**Lab 4**

Online version: <https://repl.it/@QuanDoan/AlgoLabs>, Lab4.java

**Problem 1**:

**Answer**:

Based on the original implementation of BubbleSort, we can optimize the algorithm by adding a flag. The flag raises when the order is not in place and after the swapping. This bases on the article <https://en.wikipedia.org/wiki/Bubble_sort>.

We optimize the original algorithm by stopping the algorithm if inner loop didn't have any swap. By using the flag, if the array is already sorted, the inner loop occurs only once for each incremental i, that means the outter loop will run O(n) time, and inner loop runs once.

Source code (as BubbleSort1 class)

**private** **void** bubbleSort1() {

**int** len = arr.length;

**boolean** swapFlag = **true**;

**for** (**int** i = 0; i < len; ++i) {

**for** (**int** j = 0; j < len - 1; ++j) {

**if** (arr[j] > arr[j + 1]) {

swap(j, j + 1);

swapFlag = **true**;

}

}

**if** (swapFlag == **false**) {

**break**;

}

}

}

**Problem 2**:

**Answer**:

The size of the effective array is the original size reduced by one after each step. Thus, if the initial size of the array to be sorted is lim, the size of each successive effective array is lim, lim -1, lim - 2, etc. We have included a debug statement in bubbleSort2() to trace the bubble process after each bubble step. This function, sortLargestLastPostion(), compares adjacent elements of an array of the specified size in sequence and swaps them if necessary.

Source code:

**private** **void** bubbleSort2() {

**int** len = arr.length;

**for** (**int** i = 0; i < len; i++) {

**int** newn = len - i;

// Debug purpose

//System.out.println("Effecitve array of size: " + newn);

sortLargestLastPostion(arr, newn);

}

}

**private** **void** sortLargestLastPostion(**int**[] arr, **int** size) {

**for** (**int** i = 0; i < size - 1; i++) {

**if** (arr[i] > arr[i + 1]) {

// swapping

**int** tmp = arr[i];

arr[i] = arr[i + 1];

arr[i + 1] = tmp;

}

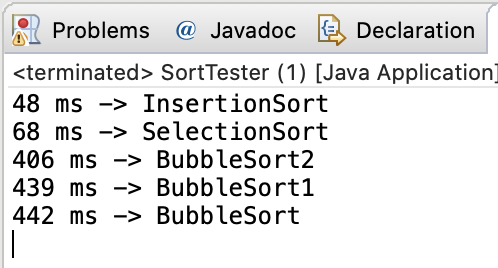
}

}

**Problem 3**:

**Answer**:

The result is captured below:



Bubble Sort is the slowest, because it costs , which is similar to Insertion Sort and Selection Sort (ref: <https://en.wikipedia.org/wiki/Sorting_algorithm>)

In details, the comparison between 3 sorting algorithms we can see in the internet.

1. Selection Sort performs a bit better than Bubble Sort. The reason is the selection sort does a smaller number of comparison and swapping steps compared to the bubble sort does every single comparison. Reference: <https://techdifferences.com/difference-between-bubble-sort-and-selection-sort.html>
2. Insertion Sort is faster than Bubble Sort, because for bubble sort, at any iteration, we have inner iteration in total. In contrast, insertion sort have maximum i iterations at step, and on average as we can stop the inner loop earlier as we found the right position for the current element. That means we have total complexity.
3. Insertion Sort is slightly faster than Selection Sort, because the time complexity of selection sort is always , whereas insertion sort has its worst-case complexity is . That means insertion sort will take lesser or equal to the worst-case or selection sort complexity. Reference: <https://cheetahonfire.blogspot.com/2009/05/selection-sort-vs-insertion-sort.html>.

**Problem 4**:

**Answer**:

1. The algorithm is correct. When we have n = 0, then or , which is correct with . The following 2 lines are to calculate the exponent n of m when n is even or odd. The reason of division by 2 for the exponent n is to reduce the stack of the recursion.

If n is even, then

If n is odd, then

Reference: <https://www.cs.cmu.edu/~cburch/survey/recurse/fastexp.html>

1. Proving:

By following the above, let assume, it’s correct with n even. We will see if correct for n odd.

which means

For example, 7/2 = 6/2. Therefore,

**Problem 5**:

**Answer**:

As we know, the power of 3 (or similar) usually takes O(logn) time. For other functions, we have O(1).

That means the total running time can be aggregated as following.

In other word, amortized cost is O(1), each operation will cost O(1) running time.

**Problem 6**:

**Answer**:

As we have the elements of the array are from the set of {0, 1, 2}, that means we just need to re-order the elements into the order 0 🡪 1 🡪 2.

So, we can simply:

* Count how many items as 0, as 1, and as 2.
* The place 0, 1, 2 accordingly to the counter from left to right in the array

Following is the code implemented.

**public** **static** **int**[] sort(**int**[] arr) {

**int** counter0 = 0;

**int** counter1 = 0;

**int** counter2 = 0;

// Loop for counters

**for** (**int** i = 0; i < arr.length; i++) {

**if** (arr[i] == 0) {

counter0++;

} **else** **if** (arr[i] == 1){

counter1++;

} **else** {

counter2++;

}

}

// Loops to arrange elements in order from left to right

// Starting with loop for 0

**for** (**int** i = 0; i < counter0; i++) {

arr[i] = 0;

}

// Then loop for 1

**for** (**int** i = counter0; i < counter0 + counter1; i++) {

arr[i] = 1;

}

// And loop for 2

**for** (**int** i = counter0 + counter1; i < counter0 + counter1 + counter2; i++) {

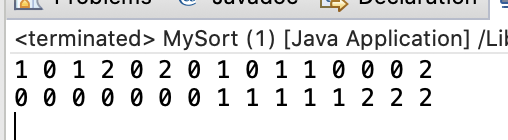
arr[i] = 2;

}

**return** arr;

}

The test as below:



With this algorithm, the running time is:

* First loop for counters costs O(n)
* The following loops placing 0, 1, and 2 costs totally O(n)
* In summary, the algorithm costs O(n)