CS 46B Lab 6
Bubble Sort

In this lab you will explore Bubble Sort, which is the simplest of all the array sorting algorithms. You will implement the algorithm, and then estimate its complexity experimentally.

As always: Work in pairs. One of you will be the "driver", who types into Eclipse. The other is the "scribe". Alternate jobs, week by week.

You will both submit lab reports.

Driver: At the top of your lab report write

CS 46B Lab Report
Your name
I was the driver, Xxxx Yyyy was the scribe.

Scribe: At the top of your lab report write

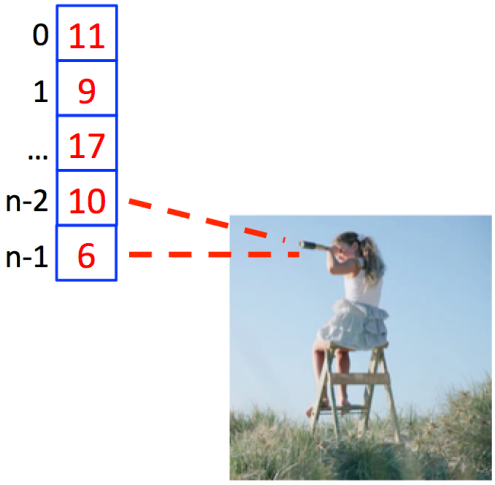
CS 46B Lab Report
Your name
I was the scribe, Xxxx Yyyy was the driver.

As you work through the lab assignment, instructions/questions highlighted in yellow should be discussed by both of you and addressed by the driver in the driver's report; instructions/questions highlighted in blue should be discussed by both of you and addressed by the scribe in the scribe's report.

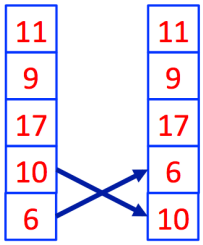
INTRODUCTION: THE BUBBLE SORT ALGORITHM

The bubble sort algorithm uses two nested loops to sort an array. The inner loop makes the array a bit more sorted. The outer loop runs the inner loop until the array is completely sorted.

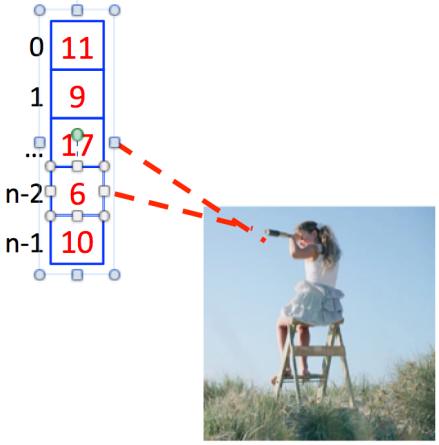
The inner loop looks at adjacent array members, starting at the end of the array:



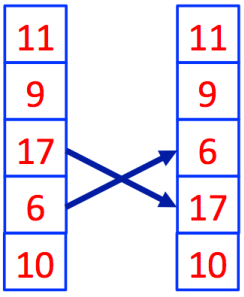
If the values are in the wrong order (as they are in this example), they are swapped:



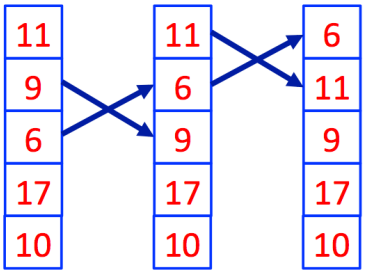
Then the algorithm compares the array members at n-2 and n-3:



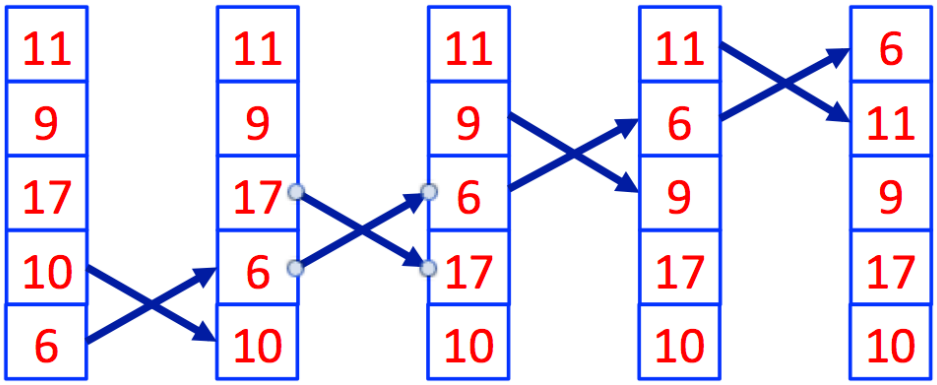
... and if the values are in the wrong order, they are swapped



... and so on:



When all the swaps are shown together below, notice how the 6 that started at the end of the array seems to float to the top, like a bubble rising through water or champagne:



Now the 1st pass through the inner loop is finished, and the 1st array slot contains the correct value. The 2nd pass then starts at the end of the array (the 10), and again works its way up the array, swapping adjacent values whenever a[i] < a[i-1]. After the 2nd pass through the inner loop, the 1st and 2nd slots contain the correct values; and so on.

GETTING STARTED

Create an Eclipse Java project containing package "bubble". Import the 3 starter files BubbleSorter.java, BubbleSortTestCaseMaker.java, and Statistician.java.

PART 1: Implementing BubbleSorter

Implement a very simple BubbleSorter class that records how many array visits and how many swaps are performed. Look at the starter file before reading on.

In the sort() method, the outer loop should execute n times, where n is the length of the array. The inner loop should compare all adjacent pairs of elements, starting with a[n-1] vs. a[n-2], and ending with a[1] vs. a[2]. If comparison determines that the elements should be swapped, the inner loop should do the swap and increment nSwaps. Whether or not the elements are swapped, the inner loop should increment nVisits by 2. Note that this isn't the most efficient way to control the loops, but it's simple so it's ok for now.

Complete the isSorted() method, which you can later use to test your code. When writing this method, remember that a value might appear more than once in the array.

Before testing, look at BubbleSortTestCaseMaker.java. It provides 4 methods that return test case arrays. The first 3 methods are obvious. The last method builds an array containing random ints; its contents will be different every time the method is called.

Test your BubbleSorter code in main(). The starter file sorts a tiny test array. When that works, replace getTiny() with getAlreadySorted(), then with getBackward(). For each case, record the number of visits and swaps; you'll need them later.

When your first 3 test cases are correctly sorted, test several times with a larger input array, returned by BubbleSortTestCaseMaker.buildRandom(100, 1000). Caution: this method returns a different random array every time it is called. Large random test cases are good for increasing confidence that an algorithm is correct, but they are difficult to debug because of their size and because they aren't repeatable. So it's good practice to start with simple test cases that make debugging easy, then move on to bigger cases after you have fixed all the bugs you can find with simple cases. Run the test twice, and record the number of visits and swaps for both runs.

Scribe: make a table like the following, and enter the counts that you observed.

Test case	Number of visits	Number of swaps
Tiny		
Already sorted		
Backward		
Random #1		
Random #2		

PART 2: BubbleSorter complexity

Complete the Statistician class, and use it to do experiments on your bubble sorter to determine if its complexity is $O(n^2)$. You can't draw valid conclusions from a single observation, or from a small number of observations. You need to execute a statistically large number of times, so that your conclusions have statistical strength. For this lab, an experiment will be 5000 executions. Define a private final static int called N_REPETITIONS, whose value is 5000.

The getStats() method of the Statistician class has an arg for specifying an array length. In that method's loop, a random array of the specified length is created. Then a BubbleSorter sorts the array. Replace the "Assert …" comment line with a line that asserts that the sorter has correctly sorted. Driver: paste that line into your report. Remember to configure Eclipse to run with assertions enabled (Run -> Run Configurations, then "Arguments tab" and type "-ea" into VM Arguments). If an assertion error is ever thrown, go back and fix your sorter. After the assertion line, replace the next comment with lines that retrieve the number of visits and swaps from the sorter, and store those numbers in the appropriate array lists.

After the loop in getStats(), write code that analyzes the 2 array lists, and prints the minimum, average, and maximum observed number of visits and number of swaps.

Before running the statistician, think about what you expect to see. You haven't been told the complexity of the bubble sort algorithm, but for now suppose that for an input array of length n, both the number of visits and the number of swaps are $O(n^2)$. Scribe: If sorting an array of length=1000 requires v visits and s swaps, approximately how many visits and swaps are required to sort an array of length=3000?

In main(), the starter code calls getStats() for array lengths 1000 and 3000. Run main(). If it takes more than 2-3 minutes, try with shorter array lengths (e.g. 500 and 1500); the second length should be 3x the first length. Driver: paste your output into your report. Scribe: does the output support the hypothesis that bubble sort is $O(n^2)$?