# COMP 2231_SW3 - Data Structures and Algorithms (Fall 2020 Carruthers)



## Assignment 1: Analysis of Algorithms & Searching and Sorting (8%)

This programming project should be completed and submitted by Monday of Week 3 if you are following the suggested course schedule. It is worth 8% of your final grade. Please refer to the "Assessments Overview" tab for details on submission of your work. Your overall course assessment information is found in your Course Guide.

1. The Shell sort (invented by Dr. Donald Shell) is a variation of the bubble/exchange sort. Instead of comparing adjacent values, the Shell sort adapts the partitioning concept from the binary search to determine a "gap" across which values are compared before any swap takes place. In the first pass, the gap is half the size of the array. For each subsequent pass, the gap size is cut in half. For the final pass(es), the gap size is 1, so it would be the same as a bubble sort. The passes continue until no swaps occur.

   Below is the same set of values as per the bubble sort example in Chapter 18 (p. 667), showing the first pass of the Shell sort:

```
9  6  8  12  3  1  7          -- size of array is 7, so gap would be 3

9  6  8  12  3  1  7          -- 9 and 12 are already in order, so no swap
^---------^
9  6  8  12  3  1  7          -- 6 and 3 are not in order, so swap
   ^---------^
9  3  8  12  6  1  7          -- 8 and 1 are not in order, so swap
      ^---------^
9  3  1  12  6  8  7          -- 12 and 7 are not in order, so swap
         ^---------^
9  3  8   7  6  1 12          -- end of pass 1
```

   The pseudo-code for the Shell sort is as follows:

```
gap = size / 2
do until gap <= 0
   swapflag = true
   do until swapflag is false
      swapflag = false
      for s = 0 to size - gap
         if num[s] > num[s + gap]
            swap num[s] with num[s + gap]
            swapflag = true
         end-if
      end-for
   end-do
   gap = gap / 2
end-do
```

   Modify `Sorting.java` to include a `shellSort` method that implements the above algorithm. Include an output of the array any time a swap occurs to demonstrate that your code works correctly. For the driver, create an Integer array using an initializer list to reproduce the above example and two additional random sets of 10 and 20 integers.

2. The bubble sort algorithm shown in Chapter 18 is less efficient than it could be. If a pass is made through the list without exchanging any elements, the list is sorted and there is no reason to continue. Create a copy of the `bubbleSort` method called `bubbleSort2` that implements this algorithm so that is will stop as soon as it recognizes that the list is sorted. Do **not** use a break statement! Include outputs of the array for both sorts for each pass through the array so you can demonstrate that the code is working correctly. The driver should test both methods with a random set of 10 integers and an already sorted set of 10 integers.

   **Hint:** You have to introduce a `swapflag` that is set true if a swap occurs on a given pass. Replace the outer "for loop" with a "while loop" that tests the `swapflag` and takes care of "index counting" within the loop.

   **Hint:** Remember that arrays are passed by reference in Java. Use `Arrays.copyOf()` or a similar method to test each sort method on the exact same data content in separate arrays.

3. Modify the `shellSort`, `bubbleSort`, and `bubbleSort2` methods from above by adding code to each to tally the total number of comparisons made between the elements being sorted (i.e., just the `compareTo()` calls; ignore relational operators between indexes). Also tally and report the number of swaps that occur. Determine and report the total execution time of each algorithm. Comment out the outputs of each swap or pass for this final step. Execute each of these sort algorithms against the same list, recording information for the total number of comparisons and total execution time. The driver should construct lists of size 10, 100, and 1000—both in random and already in sorted order. Use the Integer wrapper class for the array type.

   Use a spreadsheet to present the test cases you have prepared, along with the comparisons, swaps, and execution time. Describe how the data obtained relates to the theoretical discussion of algorithm efficiency presented in the chapter.

   **Hint:** Remember that arrays are passed by reference in Java. Use `Arrays.copyOf()` or a similar method to test each sort method on the exact same data content in separate arrays.

| Assignment Marking Criteria | Weighting |
|---|---|
| **Correctness of solution:** Algorithm is implemented and produces correct results for the stated problem. | /4 |
| **Testing:** Submission of test exhibits to indicate the solution works for a range of cases (e.g., minimum and maximum inputs) and handles unexpected exceptions. | /2 |
| **Comments and documentation:** Source code contains comments that explain in plain English what the code is intended to do.<br>### Note<br>Javadoc style is not required. | /2 |
| **Total** | **/8** |

# Submission status

| | |
|---|---|
| **Attempt number** | This is attempt 1. |
| **Submission status** | No attempt |
| **Grading status** | Not graded |
| **Last modified** | - |

Submission comments ▸ Comments (0)

Add submission

You have not made a submission yet.

◀ Suggested Schedule

Jump to...

Assignment 2: Stacks & Queues (8%) ▶

805 TRU Way
Kamloops, BC V2C 0C8
Canada
**Contact Us**