Week 3: Sorting Algorithms

This week you will write the Java code for two of the sorting algorithms studied so far: Bubble Sort and Cocktail Shaker Sort.

Before starting the tasks below, make sure you remember the difference between the sorting algorithms studied so far. Make use of you notes, lectures' slides and Google if you need.

Part A – Bubble Sort and Cocktail Shaker Sort

Using the Source code provided in the project (SortingTest.java) on Blackboard attempt the following exercises. You can also find the pseudo-code of Bubble Sort and Cocktail Shaker sort at the end of this document. Use them as a guide to write the Java code.

1) Implement the BubbleSort algorithm (simple version, not optimised yet). Check that the results are as expected (i.e. the integer values are sorted in ascending numerical order). – see pseudo-code 1 and video if you need.

2) Now implement the optimised version of Bubble sort, which monitors if no swapping operations occurs in a pass and if so, exits the program early - see pseudo-code 2 and video if you need.

3) Finally, improve the Bubble Sort version you implemented in Task 2 by reducing the number of comparisons by one each time. This is a valid step because after each run of the inner loop, the biggest item is at the end of the array (its final position) and you don't need to include it in the new comparison-loop. See pseudo-code 3 if you need.

4) Check by running the three algorithms several times on the same set of data that there is indeed a reduction in the number of steps required to sort the data when using the optimised version. You could do this by introducing a "counter" variable to store the number of steps. This variable increases by one every time a new comparison is made, and by three every time a swap is made (remember that a swap consists of three assignments, which means three steps).

5) Now implement the CocktailShaker Sort algorithm. You can find the pseudo-code for CocktailShaker Sort here and a video if you need.

6) You will now have two main sorting methods at your disposal (Bubble Sort from Task 3 and Cocktail Shaker Sort from Task 5). Arrange these methods so that the first sort arranges the array elements in ascending order and the second in descending order. Print the values in the array after each sort has been completed. Check that the results are as you would expect them to be.

Part B – measuring tasks

1. Add a counter to the methods bubbleSort_opt2() and shakerSort() that counts the number of comparisons made. Run the methods with arrays of different sizes (say, n = 10, 20, 100 and 500 random integers). How do these algorithms compare? On what size, if any, does the difference in the number of comparisons become significant? Build a table of results.
2. A more comprehensive analysis would require you to keep two types of counter, one to keep track of the number of comparisons made, and one to keep track of the number of assignments made (remember, a shift operation or an insert operation is worth one assignment, but a swap operation takes three assignments). Obtain a feel for the relative cost of the two sort algorithms seen so far by running them on the same arrays of data and comparing their counter values. Build a table of results. Repeat including bubbleSort() and bubbleSort_opt1().

Extension Tasks – Project Euler

Project Euler is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.

Use the link below to access the website, start from problem 1 and keep practicing every week. You can sign-up to save your progress (it’s free), or you can simply implement the problems without signing up.

https://projecteuler.net/archives

Appendix 1 – Pseudo-code Bubble Sort

```
 PROGRAM BubbleSort:
     Integer Age[8] <- {44,23,42,33,16,54,34,18};
     FOR j IN 0 TO N-1
         DO FOR Index IN 0 TO N-2
               DO IF (Age[Index+1] < Age[Index])
                      THEN Swap(Age[Index+1], Age[Index])
               ENDIF;
         ENDFOR;
     ENDFOR;
 END.
```

Appendix 2 – Pseudo-code Bubble Sort Optimised 1

```
PROGRAM BubbleSortOptimized1:
      Integer Age[8] <- {44,23,42,33,16,54,34,18};

        FOR j IN 0 TO N-1
          DidSwap <- FALSE;
            DO FOR Index IN 0 TO N-2
                 DO IF (Age[Index+1] < Age[Index])
                        THEN swap(Age[Index+1],Age[Index]);
                              DidSwap <- TRUE;
                     ENDIF;
            ENDFOR;

        IF (DidSwap = FALSE)
            THEN EXIT;
          ENDIF;
      ENDFOR;
END.
```

Appendix 3 – Pseudo-code Bubble Sort Optimised 2

```
 PROGRAM BubbleSortOptimized2:
      Integer Age[8] <- {44,23,42,33,16,54,34,18};

        FOR j IN 0 TO N-1
           DidSwap <- FALSE;
          DO FOR Index IN 0 TO N-2-j
              DO IF (Age[Index+1] < Age[Index])
                    THEN swap(Age[Index+1],Age[Index]);
                         DidSwap <- TRUE;
                 ENDIF;
          ENDFOR;

        IF (DidSwap = FALSE)
          THEN EXIT;
         ENDIF;
       ENDFOR;
END.
```

Appendix 4 – Pseudo-code Cocktail Shaker Sort

The simplest form goes through the whole list each time:

```
procedure cocktailShakerSort(A : list of sortable items) is
    do
        swapped := false
        for each i in 0 to length(A) − 2 do:
            if A[i] > A[i + 1] then // test whether the two elements are in the wrong order
                swap(A[i], A[i + 1]) // let the two elements change places
                swapped := true
            end if
        end for
        if not swapped then
            // we can exit the outer loop here if no swaps occurred.
            break do-while loop
        end if
        swapped := false
        for each i in length(A) − 2 to 0 do:
            if A[i] > A[i + 1] then
                swap(A[i], A[i + 1])
                swapped := true
            end if
        end for
    while swapped // if no elements have been swapped, then the list is sorted
end procedure
```