Lesson 13 – Simple Sorting Algorithm

Introduction

Last week you learnt about sorting and its applications. This lesson covers several simple sorting techniques that can be applied in such applications. Moreover, we discuss about the advantages and disadvantages of these sorting algorithms.

Learning outcome

After completing this lesson, you would be able to describe simple sorting techniques. Thus, you should be able to,

- Describe simple sorting techniques
- Describe disadvantages or advantages of simple sorting techniques

13.1 Classification of Internal Sorting

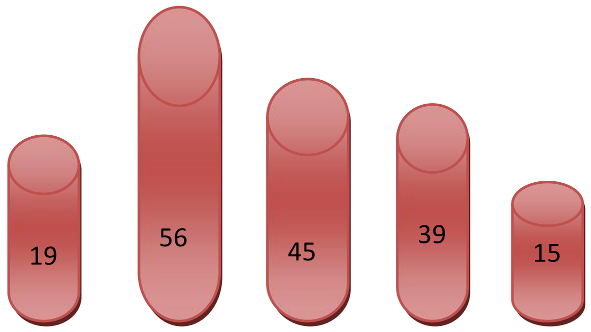
We have learnt that internal sorting is about sorting records that are in main memory. Internal sorting algorithms are often classified according to the amount of work required to sort a sequence of elements. The amount of work refers to two basic operations of sorting:

1. Comparing two elements and
2. Moving an element from one place to another

The major internal sorting techniques fall into one of three categories with respect to the work required to sort a sequence of ***n*** elements. Namely Simple sort, Advance sort and Radix Sort. The three sorting techniques: Selection Sort, Bubble Sort (Exchange Sort) and insertion sort comes under Simple Sorting.

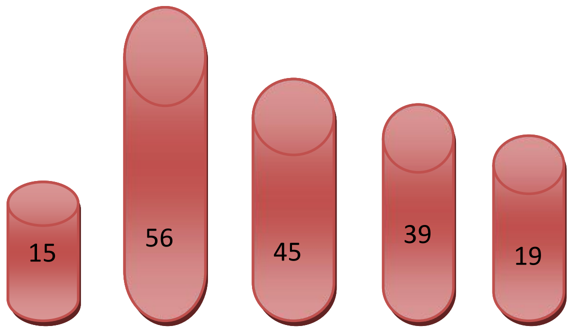
13.2 Selection Sort

A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions. The basic operation in a selection sort is the selection of smallest (or largest) element from a sequence of elements. For an example, suppose that a set of bars with different heights are arranged as follows and it is required to sort them in ascending order.

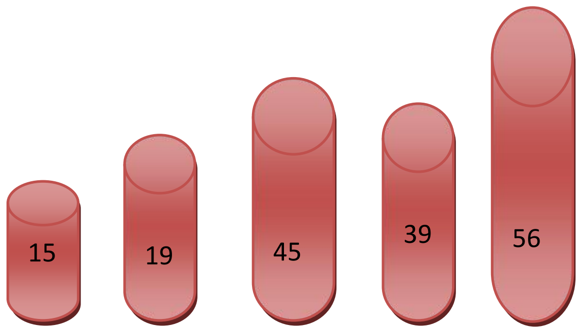


As the first step we scan the bars and find the smallest bar and exchange it with 1st bar. Then we repeat same process form the 2nd bar and so on. In this case after processing 4 process all these bars in correct order.

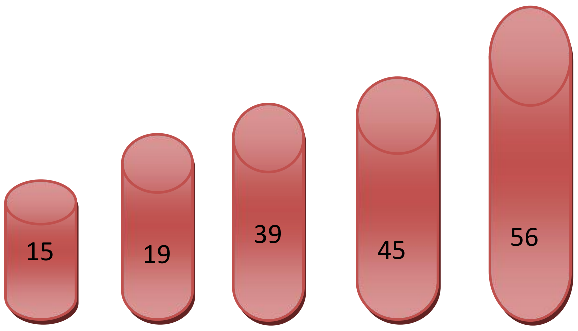
After 1st cycle



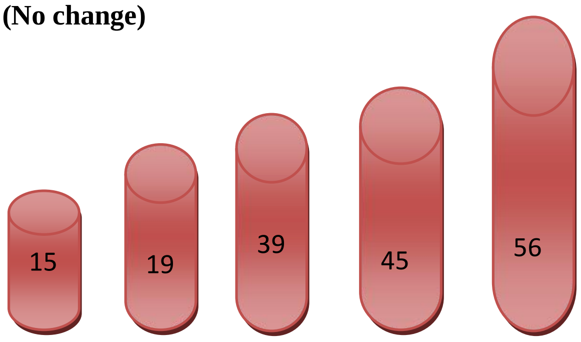
After 2nd Cycle



After 3rd Cycle



After 4th Cycle (No change)



The bars are sorted in ascending order of their heights.

Let's look at the following video. It also describes the same process of selection sort. The said video has also been uploaded to the course website.

Source: https://www.youtube.com/watch?v=EdUWyka7kpI

Let’s see how selection sort can be implemented in Java.

```
 public void selectionSort()
    {
    int out, in, min, temp;
      for(out=0; out<nElems-1; out++)
       {
            min = out;                    // minimum
              for(in=out+1; in<nElems; in++) // inner loop
                if(a[in] < a[min] )        // if min greater than the value in a[in]
                 min = in;              // we have a new min

              temp = a[out];
              a[out] = a[in];
              a[in] = temp;
        }
    } // end selectionSort()
```

The selection sort method has been implemented using one outer loop and one inner loop. In *selectionSort* method each of these are implemented using Java “for” construct.

The worst case, inner loop is executed following number of times:

Number of comparisons = $(n-1) + (n-2) + ...+ 3 + 2 + 1 = n (n-1)/2 = O (n^2)$.

The worst case, outer loop is executed n-1 times.

Each execution of the inner loop required one comparison and unknown number of assignments. Each execution of outer loop requires one exchange.

The total effort can be summarized as;

Exchanges = $n-1 = O(n)$

Comparison = $n (n-1)/2 = O (n^2)$

Note that the effort required, as measured by the number of comparisons and exchange is independent of the arrangement of the data been sorted. On the negative side, selection sort requires $O(n^2)$ comparisons no matter how the data are initially ordered. On the positive side, it never required more than O(n) moves. This makes selection sort is a good method for large elements (that are expensive to move) with short sort items (that are easy to compare).

Activity 13.1

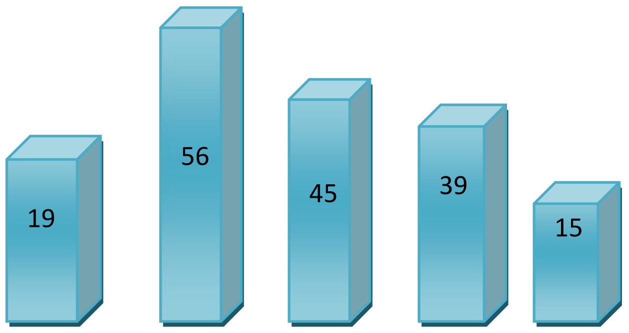
Suppose that you are going to use selection sort for sorting 5 integers: 390, 205, 182, 45, and 235. Show that the output of each process (cycle) when sorting using selection sort.

Activity 13.1 - Answer

After 1st cycle:	45	205	182	390	235
After 2nd Cycle:	45	182	205	390	235
After 3rd Cycle:	45	182	205	390	235 – no change
After 4th Cycle:	45	182	205	235	390

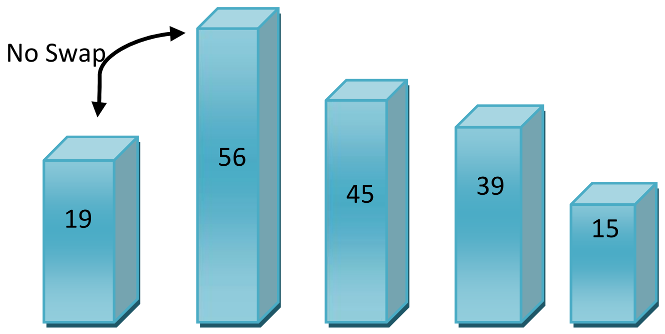
13.3 Bubble Sort (Exchange Sort)

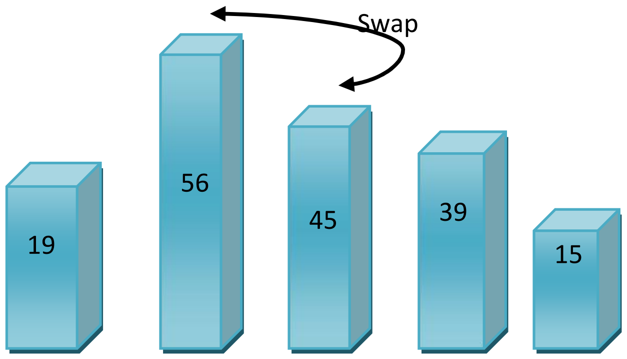
The simple sort, which is most widely known among beginning students of programming is Bubble sort. One of the characteristics of this sort is that it is easy to understand and program. The basic operation in an exchange sort is the exchange of adjacent pair of elements. The overall sort process consists of a number of passes over the data. Each pass starts at one end of the array and works towards the other end. Each pair of elements that is out of order is exchanged. Suppose that a set of bars with different heights are arranged as follows and it is required to sort them in ascending order.

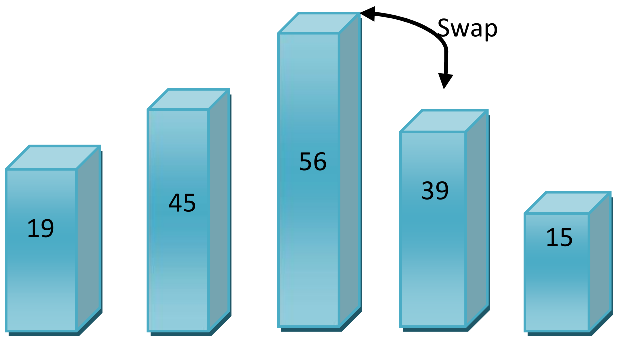


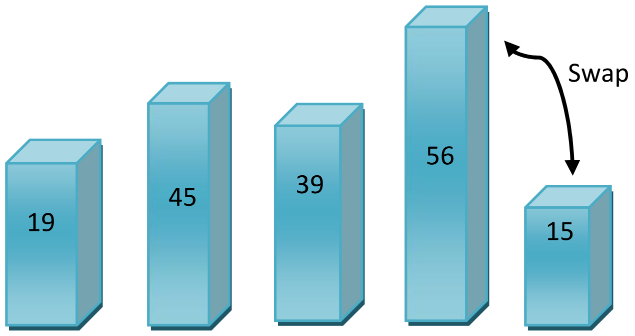
The first pass moves the largest element to the last position (Here at 5th position), forming a sorted list of length 1. The second pass only has to consider 4 elements and it moves the second largest element to the 4th position. The third pass only has to consider 3 elements and it moves the third largest element to the 3rd position. The forth pass only has to consider 2 elements and it moves the forth largest element to the 2nd position. The element remaining at 1st position is already sorted. If no exchanges are made during one pass over the data, the data are sorted and the process terminate. This produces following cycle.

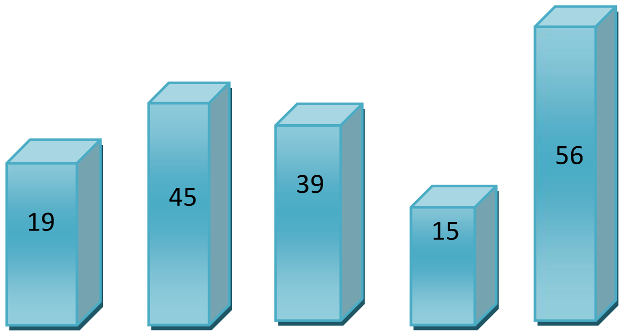
1st Pass




Swap
19
56
45
39
15


Swap
19
45
56
39
15


Swap
19
45
39
56
15



Let's look at this Video and see how bubble sort works.

Source: https://www.youtube.com/watch?v=yIQuKSwPlro

Let's see how bubble sort can be implemented in Java.

```
public void bubbleSort()
    {
    int out, in;
    for(out=nElems-1; out>1; out--)   // outer loop (backward)
    for(in=0; in<out; in++)       // inner loop (forward)
        if( a[in] > a[in+1] )      // out of order?
         swap(in, in+1);        // swap them
    } // end bubbleSort()
```

```
//--------------------------------------------------------------
private void swap(int one, int two)
   {
         double temp = a[one];
         a[one] = a[two];
         a[two] = temp;
   }
```

In the above method, the outer loop and inner loop are implemented with a java "for" statement. The inner loop is executed the following number of times:

$(n-1) + (n-2) + \ldots + (n-\text{ksorted}) = ½ (2n – \text{ksorted} -1)\text{ksorted}$,

Where ksorted is the number of executions of the outer loop before there is pass during which there are no Exchanges. The inner loop contains one comparison and sometimes an exchange.

The best performance occurs when ksorted is one; this means that no exchanges occur because the original data were sorted. The number of comparisons use by above method to determine this is,

$$½ (2n – 1 - 1) = n-1 = O (n)$$

The worst performance occurs when ksorted is n-1. The inner loop is executed the following number of times.

$$½ (2n – (n-1) -1)(n-1) = ½ n(n-1) = O (n^2)$$

Further when we translate loop executions into exchanges and comparisons we obtain the following results.

	Exchanges	**Comparisons**
Worst case	1/2n (n-1)	1/2n (n-1)
Sorted data	0	n-1

An exchange sort has two major drawbacks. First its inner loop contains an exchange that requires three moves. Notice that in the selection sort method there are no moves in the inner loop. Second, when an element is moved, it is always move to an adjacent position. In selection sort, an element may move a long distance, and it always moves to its sorted position. All in an, bubble sort is one of the slowest sorting algorithms known.

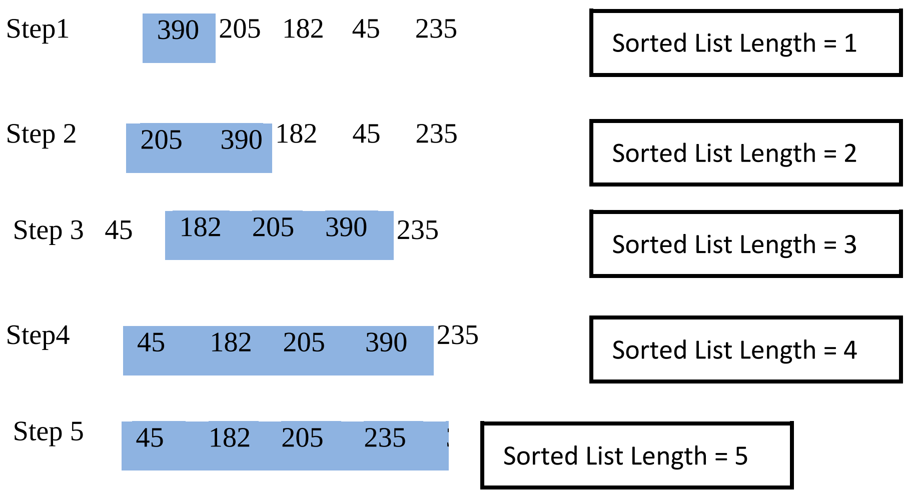
13.4 Insertion Sort

An insertion sort is the one that sorts a set of records by inserting records into an existing sorted file. The basic operation in an insertion sort is the insertion of a single element into a sequence of sorted elements so that resulting sequence is still sorted.

Thus the insertion sort algorithm proceeds on the idea of keeping 1st part of the list, when once, examined, in the correct order. An initially list can only one item (element) is automatically in order. If we suppose that we have already sorted the 1st (i-1) items, then we take item i and scan through this sorted list of length i-1 to see where to insert item i.

As an example in the case of sorting in ascending order as soon as an item (element) is found that is large compared to the element to be inserted, the scan terminates and insertions occur ahead of the large element.

For example consider the list of five integers. 390 205 182 45 235



Look at the video on course web site, which demonstrates how insertion sort works.

```
public void insertionSort()
    {
    int in, out;
     for(out=1; out<nElems; out++)    // out is dividing line
      {
            double temp = a[out];     // remove marked item
             in = out;                // start shifts at out
```

```
        while (in>0 && a[in-1] >= temp) // until one is smaller,
        {
          a[in] = a[in-1];        // shift item to right
        --in;                  // go left one position
        }
         a[in] = temp;              // insert marked item
      } // end for
    } // end insertionSort()
```

How many comparisons and exchanges does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of n-1 comparisons on the last pass. Then,

$$1+ 2 + 3 + \ldots + n = n(n-1)/2$$

However, because on each pass an average of only half of the maximum number of items are actually compared before the insertion point found, we can divide by 2, which gives n(n-1)/4.

The number of exchanges is approximately the same as the number of comparisons. However, a copy isn't as time-consuming as a swap in bubble sort, so for random data this algorithm runs twice as fast as the bubble sort and faster than the selection sort. Thus in any case, the insertion sort runs in O (n^2) time for random data.

For data that is already sorted, the insertion sort does much better. When data is in order which we are considering, the condition in the *while* loop is never true, so it becomes a simple statement in the outer loop, which executes n-1 times. In this case the algorithm runs in O(n) time, which makes it a simple and efficient way to order a file that is only slightly out of order.

Summary

In this lesson you have learnt about simple sorting algorithms. You got know different sorting algorithms such as selection sort, exchange sort (bubble sort) and insertion sort. Further you learnt about how efficient they are. During the next lesson you will be able to learn about advanced sorting algorithms including Quick Sort and Merge Sort.